

The Smalltalk Report

The International Newsletter for Smalltalk Programmers

March/April 1992

Volume 1 Number 6

REIMPLEMENTING MODEL-VIEW- CONTROLLER

by David J. Leibs and
Kenneth S. Rubin

Contents:

Features/Articles

- 1 Reimplementing Model-View-Controller
by David J. Leibs & Kenneth S. Rubin

Columns

- 8 Object-oriented design: Becoming more predictable
by Rebecca Wirfs-Brock
- 11 Smalltalk with style: Tips for improved Smalltalk reuse and reliability
by Ed Klimas & Suzanne Skublics
- 15 GUIs: Paint palettes (taking control in Smalltalk/VPM 1.3)
by Greg Hendley & Eric Smith
- 17 Best of comp.lang.smalltalk

Departments

- 20 Lab Review: Smalltalk at the University of Washington
reviewed by Bjorn Freeman-Benson
- 22 Product Review: Coopers & Lybrand's AM/ST, Version 3.5
reviewed by Jim Salmons & Timlynn Babitsky
- 28 Product Announcements
- 29 What They're Saying About Smalltalk



The Model-View-Controller (MVC) architecture was first conceived at the Xerox Palo Alto Research Center in the late 1970s and early 1980s. Over the past 10 years, the MVC architecture was used to develop many sophisticated Smalltalk graphical user interface applications. Engineering of these applications illustrated both the strengths and weaknesses of the MVC concept and implementation. Much of the experience gleaned from developing these applications influenced the changes made to the MVC facilities of the Objectworks\Smalltalk Release 4 system*. These changes were made to make the MVC implementation more understandable, reusable, and efficient.

This column describes the reimplementation of MVC in Release 4, and is intended for individuals who are interested in understanding and using the revised MVC architecture. It is not intended to be a tutorial on the subject of MVC and therefore assumes that the reader is already familiar with basic MVC concepts. Other articles^{1,2} provide a quick introduction to MVC and some examples of its usage. A more comprehensive overview of the pre-Release 4 implementation is also available.³

In this column, we first discuss aspects of the classic MVC implementation that motivated a redesign and reimplementation in Release 4. Next, we introduce some of the architectural features of Release 4 MVC, and discuss how these features are used to develop sophisticated applications, giving special focus to the new concepts of wrappers and invalidation-based redisplay.

CLASSIC MVC IMPLEMENTATION

The classical MVC implementation served application developers well for the past 15 years. However, there were several deficiencies that often made the implementation task more difficult than necessary. Specifically, much of the MVC-related functionality was vested in a few components that were large and difficult to understand. In addition, a fundamental distinction was made between display objects, which could render themselves on an arbitrary display medium, and views, which could only render themselves on the display screen.

TOO FEW AND TOO LARGE

In classic MVC, too many of the features required to develop interesting graphical user interfaces were located in too few components. In particular, the base class `View` embodied far too many features. This is a typical object-oriented design mistake, creating a few heavyweight components that serve multiple functions and can be difficult to understand and extend. Applications that use these components are forced to deal with a larger set of features than may be required to perform the desired task. This results in applications that are unnecessarily complex and inefficient.

continued on page 3...

*Objectworks is a registered trademark of ParcPlace Systems Inc



John Pugh



Paul White

EDITORS' CORNER

In a recent retrospective in the February issue of the *Hotline on Object-Oriented Technology*, prominent industry watcher Tom Love gave his synopsis of the most significant O-O related events of 1991. It was interesting to see how many of the events that made his list were Smalltalk related—the introduction of the Momenta pen-based computer, which was developed using Smalltalk technology; the partnership of Sequent, Versant, and ParcPlace to develop a parallelized version of Smalltalk; the development of a batch version of Smalltalk that runs on MVS; and, perhaps the most improbable sight of the year, IBM salespeople peddling Smalltalk/V! What will 1992 hold for Smalltalk?

Another topic mentioned by Tom is one that is near and dear to our own hearts—the dearth of object-oriented education and training in educational institutions. Tom is right: “most university graduates have not heard of objects.” At Carleton, for the last three years we have been doing as Tom suggests, teaching Smalltalk as the first programming language for CS majors. It’s been a great success. We’ve some good news for Tom. Things are changing, if only slowly. Many universities have introduced OOP into the curriculum if only at the graduate or senior undergraduate levels. There are some big obstacles to overcome if the situation is to improve—the current lack of knowledge among instructors about OOP technology, the lack of sample curricula and good texts, and the politics involved in making significant curriculum changes, to name but a few. For the first time, OOPSLA '92 in Vancouver, Canada will feature a special one-day Educators' Symposium—a forum for educators to share their experiences and ideas with those contemplating introducing object-oriented technology into the curriculum. Let's hope lots of educators attend.

In this month's lead article, David Leibs and Kenny Rubin from ParcPlace Systems give us the inside story on the reimplementing of the Model-View-Controller (MVC) paradigm in Release 4. They review the original implementation of MVC in Smalltalk-80 and discuss its deficiencies. By unifying display objects and views into a hierarchy of visual components built on a structured graphics foundation, the new implementation aims to provide a more coherent framework for producing sophisticated graphical user interfaces using Objectworks \Smalltalk. This article provides the rationale behind the introduction of concepts such as wrappers, SPIM, and composite parts in Release 4 and is a must read for all Smalltalk programmers.

We welcome Alan Knight as a regular contributor to *The Smalltalk Report*. Alan will be monitoring the USENET bulletin boards for items of interest to Smalltalkers. In his first column, he provides answers to some frequently asked questions and, in particular, reports on a subject that troubles many beginning and experienced Smalltalk programmers—how to deal with lost instances.

Ed Klimas and Suzanne Skublics return with their popular Smalltalk with Style column. This month they provide tips on the use of global, pool, and class variables as well as describing the use of multiple dispatching to eliminate the use of non-object-oriented case statement-like code. We hear a lot about the benefits of using an incremental, iterative approach to software development in Smalltalk. The process is not without its dangers and pitfalls but, as Rebecca Wirfs-Brock illustrates, proper planning and management can go a long way to ensuring success and removing unpredictability. In the GUI column, Greg Hendley and Eric Smith describe how to subclass the OS/2 ValueSet control to produce a palette view for selecting paint colors.

Following our recent spate of articles on team-oriented development in Smalltalk, Jim Salmons and Timlynn Babitsky provide a comprehensive review of the AM/ST application manager product for Smalltalk/V. And finally, in this issue's lab review, Bjorn Freeman-Benson reports on the research in constraint-based systems at the University of Washington.

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 Inc, 588 Broadway, New York, NY 10012 phone: (212) 274-0640; fax: (212) 274-0646. © Copyright 1992 by SIGS Publications, Inc. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publishers. Mailed First Class. Subscription rates 1 year, (9 issues) domestic, \$65. Foreign and Canada, \$90. Single copy price, \$8.00. POSTMASTER: Send address changes and subscription orders to: THE SMALLTALK REPORT, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada.

continued from page 1...

DISPLAY OBJECT VS. VIEW

In classic MVC two separate hierarchies of classes were used to construct applications with visual presentations—the *Display-Object* and the *View* hierarchies. Both hierarchies had the same intended function of rendering visual presentation, but each supported different protocols with incompatible implementations. In particular, *DisplayObjects* such as *Form*, *Circle*, and *Line* could render themselves on an arbitrary medium at an arbitrary location, whereas *Views* could only render themselves on the display screen at a fixed location. These differences made it impossible to use *DisplayObjects* and *Views* interchangeably.

CLASS VIEW

The classic MVC version of class *View* had so many different features that it forced application developers to make generalizing assumptions that were often untrue and led to certain conceptual as well as operational inefficiencies. These assumptions involved visual, composite, layout, and control properties.

View's visual properties included both edge decorating, which described border thickness and color, and the inside color of the view. In addition, each view assumed that it was a composite and might contain subviews—an assumption that was certainly incorrect for leaf views in a view structure tree.

As for layout, each view knew its position both relative to the containing view and the containing screen. This support was provided by a complex, redundant set of transformations that were difficult to understand and use. In addition, each subview assumed that it scaled (occupied a relative percentage of the area) with respect to its containing view. This made it very difficult to have fixed-sized views. Furthermore, all views were assumed to be aligned in a tiled fashion. If subviews overlapped, it was not possible to update an occluded view without damaging the visible view. (Worse yet, no notification was provided to the damaged view.) The same scenario was true for windows that overlapped on the screen.

Finally, every view had to be prepared to handle control when it received it via control delegation. This required every view in a view structure tree to have a controller associated with it.

Oversized by all these features, and overconstrained by underlying assumptions, *View* instances were quite monolithic (564-byte space overhead inherited from class *View*) and often difficult to use as desired.

DISTINCTION BETWEEN APPLICATION VIEWS AND STRUCTURED GRAPHICS

Previous Smalltalk implementations provided the MVC framework as a means of creating application views. However, if applications required structured pictures it was the responsibility of the developer to create the graphics framework to support these applications.

Structured graphics frameworks typically embody a collection of lightweight, displayable objects that can be composed

and can overlap. Such objects display themselves on a medium by communicating with an object that carries translation, clipping, and visual properties—an object that the literature typically refers to as a “graphics context.” Good structured graphics frameworks carry forward ideas from MVC such as the use of the dependency mechanism to initiate a change in a displayed object. Having initiated the change, these systems then employ the concept of propagating a damage rectangle up the containing hierarchy to effect a redisplay. This is a simple yet powerful mechanism for dealing with overlapping displayed objects.

Most applications in classic MVC required both application views, as provided by the system, and structured graphics, which had to be provided by the programmer. Such applica-

“

A major focus of Release 4 was the development of a framework that incorporated the best ideas of structured graphics and MVC.

”

tions were required to deal with the presence of two parallel frameworks that overlapped in intent and functionality. Consequently, these applications suffered from the union of the inadequacies. For example, class *View* could only display at absolute positions on the actual display screen (*Display*). In the context of dependency-driven redisplay, *View* ignored the possibility of overlapping views and windows—the updating window had to pop to the top to *displaySafe*. As for the structured graphics, they had no inherent notion of control since controllers very much expected to be connected to views.

In addition, it was impossible to mix the two ideas together. For example, the structured graphics would reside inside a view, but there was no way of putting a view inside a structured graphic picture, e.g., making a *TextEditor* part of a structured picture). As for views, they could only contain other views. As such, views could draw structured pictures inside of themselves, but they did not contain the pictures as part of their inherent structure. This meant that displaying, interacting, and updating of pictures inside of views was handled in a separate and different manner than displaying, interacting, and updating of the views.

A major focus of Release 4 was the development of a framework that incorporated the best ideas of structured graphics and MVC. From structured graphics the ideas include lightweight components, fine-grain control over layout, display on arbitrary media, and invalidation-based redisplay.

From MVC the ideas include dependency-based redisplay and separate control for user interaction.

RELEASE 4 VISUAL PRESENTATION FEATURES

Two somewhat independent major functionality changes in Release 4 directly affect the MVC redesign—host window integration and the new imaging model. While a full description of these is beyond the scope of this article, this section summarizes their most significant impacts on the MVC framework.

HOST WINDOW INTEGRATION

Under host window integration, Smalltalk windows are host windows. As such, the responsibilities of the classic Smalltalk window manager have been relegated to the host window manager. Within this framework, Smalltalk is provided with event notifications when windows are manipulated using the host window manager. This event notification paradigm required substantial modifications to the classic `StandardSystemView`. In particular, the Release 4 version of this class, called `ScheduledWindow`, must now be prepared to deal with invalidation-based redisplay, a topic to be discussed later. Furthermore, in Release 4 every window has its own sensor, which is its channel to receive host events. This sets the stage for complete event-driven input, although, for backward compatibility reasons, Release 4 continues to use a polling scheme.⁴

SMALLTALK PORTABLE IMAGING MODEL

While classic Smalltalk had monochrome `BitBlt` drawing capabilities, Release 4 provides a full-color, platform independent, high-level graphics imaging model known as the Smalltalk Portable Imaging Model (SPIM).⁴ The class `GraphicsContext` defines the SPIM application programmers interface. Messages are sent to a `GraphicsContext` to invoke host graphic capabilities (drawing circles, lines, rectangle, polygons) on a display surface (`Window`, `PixelFormat`). All visual presentations in Release 4 are achieved using this approach. The invalidation-based redisplay capability of MVC uses `GraphicsContext` as a means of carrying translation, clipping, and certain visual properties of visual components.

VISUAL COMPONENTS

In the Release 4 system, the basic presentation metaphor is that of a component that is capable of rendering a visual presentation of itself on a `DisplaySurface` using a `GraphicsContext`. Each such component can answer a preferred bounds, which indicates the area of space that the component would prefer to occupy. In addition, a visual component can respond with an object that is capable of handling user interaction, if such interaction is desired.

Visual components can also be composed into hierarchical presentations by adding them to composite visual components. Any visual component structure, singular or composite, can be placed inside a host window.

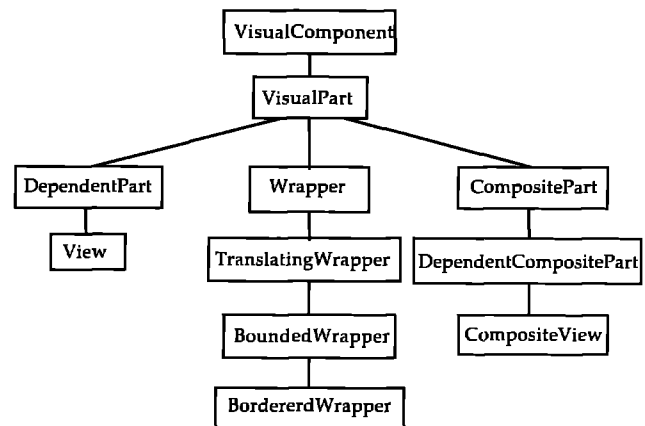


Figure 1. A partial VisualComponent hierarchy.

Figure 1 shows a partial hierarchy of the Release 4 classes that support visual components. The root class of the hierarchy is `VisualComponent`, representing a lightweight, stateless, abstract object that can create a visual presentation. This class specifies the important abstract protocol of `displayOn: aGraphicsContext and preferredBounds`. In addition, it specifies the default implementation of `objectWantingControl`, which answers `nil` indicating no control is desired. Subclasses that wish to take control must redefine this behavior to respond with the object to which control should be passed, e.g., a view would respond with a particular controller.)

`VisualPart`, a subclass of `VisualComponent`, is used to make up a structured picture by providing a pointer to some containing visual component. Some visual components need to know about a containing visual component, others do not. Those that do need to know are interested in interacting with their surrounding environment.

At this point the hierarchy splits into visual components that contain other visual components, and those that do not. The class `CompositePart` captures the idea that a visual component can be a collection of other visual components. With the introduction of composition, it is necessary to consider where each component resides in the coordinate system of the composite. Rather than storing this information in the visual component itself (as with the old class `View's insetDisplayBox`) or in some form of parallel record structure in the composite, the information is held in a special type of visual component called a *wrapper*. A wrapper is a visual component that contains exactly one other visual component.

MVC IN RELEASE 4

The classes `VisualComponent`, `VisualPart`, and `CompositePart` provide the base structured graphics framework upon which the MVC facilities are implemented. Purposefully absent from the framework thus far is the notion of a model and a controller.

Class `DependentPart`, a subclass of `VisualPart`, provides a model for dependency relationships. As a subclass of `VisualPart`, it has a direct way of turning an update: message from a

model into an invalidation message that is propagated up the containment hierarchy to effectuate a dynamic redisplay consistent with the surrounding environment.

The class *View* refines the dynamic redisplay behavior of *DependentPart* by adding a controller. This enables users to interact with a view and model in the traditional manner dictated by MVC. *View* provides the abstraction for components such as *TextViews* and *ListViews*.

For completeness, there are also *DependentCompositePart* and *CompositeView* to support model and controller behavior for composite visual components.

Notice, however, how low in the hierarchy the class *View* is located. There are numerous lightweight abstractions from which it is derived (in contrast to previous versions where *View* was the heavyweight abstraction).

Even this low, *View* is lighter weight than in classic MVC. Many features of the old class *View* are no longer present. In particular, there is no mention of information related to border width, border color, or inside color. This information has been removed and is now the responsibility of wrappers. Likewise, there is no information regarding layout and current position—tracking this information has also been relegated to wrappers.

The resulting framework provides the classic MVC functionality within a structured graphics framework. It does so in a singular, unified manner by which all visual components are created, decorated, positioned, manipulated, and controlled.

WRAPPERS

The idea of a wrapper (or wrapping) is a generic design technique. It is a form of composition and delegation that can be thought of as selective message forwarding. That is, given the desire to enhance the capabilities of an existing object, we might choose to subclass the object to add the additional capabilities, or we might construct a new object that is not necessarily hierarchically related to the existing object, but contains the new required features. The idea of wrapping is to combine the two objects by placing the original object inside of the wrapper, which is used to intercept messages destined for the original object. When intercepting messages, wrappers may perform their own special behavior, and then forward the message on to the original object.

As an example of how wrappers are used, examine the application shown in Figure 2. This represents a classic struc-

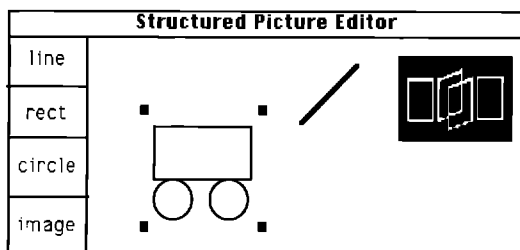


Figure 2. A typical application.

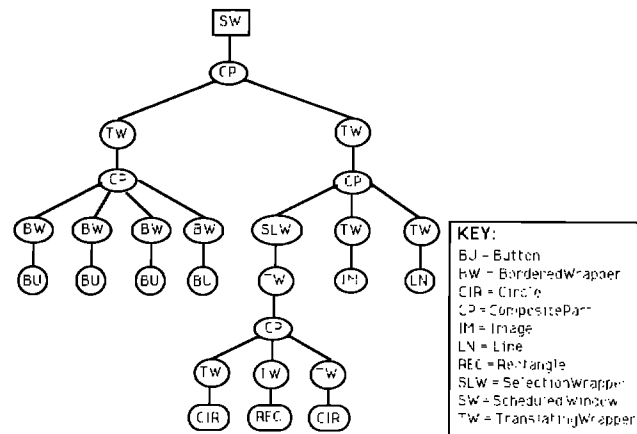


Figure 3. Visual components of the structured picture editor.

ured picture editor where the user is able to manipulate a variety of visual objects on a drawing surface. In particular, the user is able to select, drag, group, and delete objects. In the application window there are four buttons and several other visual objects. The image and line are single visual objects. The rectangle and two circles have been grouped together into a composite object, which is currently selected (denoted by the selection handles). The visual component-level structure of the current state of this application is illustrated in Figure 3.

Wrappers are visual components designed to extend the features of other visual components. As such, wrappers respond to a particular part of the visual component protocol. To understand this, it helps to consider how wrappers interact with other visual components. To begin with, the leaf visual components of any hierarchy are always surrounded by wrappers. In Figure 3, notice that all leaves in the tree are suitably wrapped. Specifically, Circle, Rectangle, Image, and Line are all contained within *TranslatingWrappers*. The buttons are contained within *BorderedWrappers*.

The components of *CompositeParts* are always wrappers, each of which can contain another wrapper or visual part transitively, until a leaf visual component is reached. Hence, at least one wrapper will always intervene between a visual component and a composite. Under this scheme, wrappers forward messages sent by the composite toward the leaf and, conversely, channel messages sent by the leaf toward the composite. Messages moving in either direction maybe filtered or modified by the intervening wrappers.

Downward-moving messages that are interpreted/forwarded by wrappers include messages used for:

- instructing a component to display itself on a *GraphicsContext*
- offering control to a component
- querying and asserting the bounds of a component, i.e., its size and position

Upward-moving messages include:

- invalidation of all or part of the component's area, i.e., requesting redisplay
- asserting that a component has a new bounds, i.e., size or location
- requesting information such as how big a component should be, as well as a variety of visual attributes

The Release 4 implementation comes with a few standard wrappers such as `TranslatingWrapper`, `BoundedWrapper`, `BorderedWrapper`, and `EdgeWidgetWrapper`.

`TranslatingWrapper` is used to position a visual component at a particular origin. In Figure 3, the structured graphic entities are all in `TranslatingWrappers`. `BoundedWrapper` provides for a bounds used in both the translation and clipping of a visual component. A `BoundedWrapper` also provides a layout to determine the bounds of the visual component. This layout typically is an instance of `LayoutFrame` that permits developers to specify the layout of a visual component with respect to another visual component in either an absolute, relative, or combination absolute/relative manner.

A `BorderedWrapper` provides for a border around a visual component, as well as an inside color. In Figure 3, the buttons all reside within `BorderedWrappers`. An `EdgeWidgetWrapper` provides a convenient means of decorating a component with desired widgets. This wrapper provides for the placement of horizontal and vertical scrollbars as well as a menu bar around the edges of a visual component.

There are many more uses for wrappers. Some of the more interesting examples that have been built include:

- double buffering for smooth animation
- read-only to limit interaction
- visibility controlling to determine whether a component is visible, invisible, or somewhere in between (alpha)
- selection indication to illustrate when a component is selected
- wrappers for arbitrary visual property manipulation

In Figure 2, the rectangle and the circles are contained within a `CompositePart`. In Figure 3, we see that the `CompositePart` is contained within a `TranslatingWrapper` that positions the composite relative to the other visual components. The `TranslatingWrapper` is itself contained within a `SelectionWrapper` that provides the visual presentation of the selection handles that are seen in Figure 2. This example illustrates that wrappers may be composed to create a desired effect, i.e., translation and selection. In addition, it should be noted that any visual component in this application can be displayed with selection handles simply by wrapping it in a `SelectionWrapper` (i.e., inserting a `SelectionWrapper` at the proper location in the structure hierarchy).

The Release 4 system supports transparent use of the standard wrappers by providing protocol in `CompositePart` that sup-

ports the automatic wrapping of constituent visual components. For example:

```
CompositePart add: aVisualComponent at: aPoint
```

places a `TranslatingWrapper` around `aVisualComponent`,

```
CompositePart add: aVisualComponent in: aLayout
```

places a `BoundedWrapper` around `aVisualComponent`, and

```
CompositePart add: aVisualComponent borderedIn: aLayout
```

places a `BorderedWrapper` around `aVisualComponent`.

In summary, the use of wrappers in the visual component framework facilitates the handling of bookkeeping information required by many visual components. In addition, it provides a flexible technique for creating specific behavior once, and quickly adding this behavior to existing objects via composition/delegation (i.e., wrapping).

INVALIDATION-BASED REDISPLAY

As part of unifying structured graphics, MVC, and host windows, it was necessary to choose a singular mechanism for dynamic redisplay. Based on the operation of host window managers and past experience with structured graphics, invalidation-based redisplay was chosen. The idea was briefly described earlier and is elaborated on here.

When a visual component wants to change its presentation, it assumes that it does not know enough about the environment in which it resides. For example, it might not know if other components are overlapping it, or there might be a collection of damage rectangles from the host window that need to be considered at the same time to avoid multiple displaying. As such, a visual component simply does not know enough to redisplay itself at the time it wants to effect a change.

To achieve a proper redisplay within its environment, the component invalidates its bounds, either in whole or in part, by sending a message up the enclosing structure environment, beginning with its container. The container, and anything above it up to the top of the tree (the host window), can modify the invalidated regions in any manner.

Once the invalidated region reaches the window, it can be merged with any damage queued by the host. Then the invalidated areas may be queued for later handling, or a message may start down the structured picture requesting visual components to redisplay themselves on a particular medium using a graphics context that has been preclipped with the invalidated regions. In this manner, all visual components achieve a proper redisplay within their environments. The idea is simple and uniform. Any complexity that arises is due to the application and not to the concept or the mechanism.

CONCLUSIONS

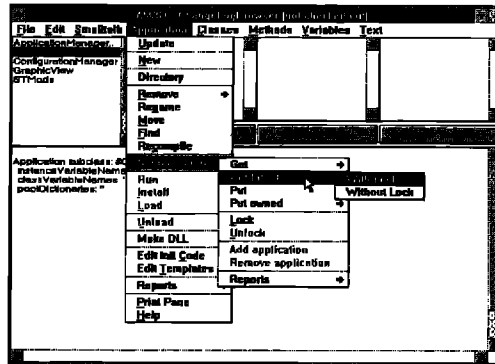
Prior to Release 4, much of the MVC-related functionality was

ImageSoft

AM/ST™

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

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



Key Features of AM/ST Include:

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

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

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

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



1-800/245-8840
ImageSoft™
The World's Leading Publisher of Development Tools

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

vested in a few components that were large and difficult to understand. In addition, a fundamental distinction was made between display objects, which could render themselves on an arbitrary display medium, and views, which could only render themselves on the display screen. In Release 4, both of these problems are handled by unifying display objects and views into a hierarchy of visual components based on a structured graphics foundation. In addition, many of the features that were previously associated with views have been relegated to wrapper visual components. This permits greater flexibility in achieving the desired results, and promotes a high level of reuse of the specialized features contained within wrappers.

Preliminary feedback from Objectworks\Smalltalk Release 4 users indicates that the new MVC facilities are powerful and permit the development of sophisticated graphical user interface applications in a more coherent manner. However, people have noted that the facilities embody different and somewhat more sophisticated principles, and thus require more time to digest. The purpose of this article is to further the understanding of the facilities and of how to successfully employ them. ■

ACKNOWLEDGEMENTS

We would like to thank all of the members of the ParcPlace Smalltalk team whose creativity and dedication made Objectworks\Smalltalk Release 4 a reality. We would also like to thank Glenn Krasner, Adele Goldberg, Brian Alexander, and Frank

Jackson for their input into this paper. However, any inaccuracies or ambiguities in this paper are solely our responsibility.

REFERENCES

1. Krasner, G. E. and S.T. Pope. A cookbook for using the Model-View-Controller user interface in Smalltalk-80, *Journal of Object-Oriented Programming* 1(3):26-49 1988.
2. Goldberg, A. Information models, views and controllers, *Dr. Dobb's Journal*, July, 1990.
3. LaLonde, W.R. and J.R. Pugh. *Inside Smalltalk Volume II*, Prentice Hall, Englewood Cliffs, NJ, 1991.
4. ParcPlace Systems. *Objectworks\Smalltalk Release Notes*, ParcPlace Systems, Inc., Mountain View, CA, 1991.

David J. Leibs is a Computer Scientist at ParcPlace Systems where he was the architect of the Release 4 MVC reimplemention and currently works on Smalltalk special projects. He studied mathematics at North Texas State University. He has been working with Smalltalk-80 since joining Xerox in 1984. He joined ParcPlace in 1986 as their first Smalltalk programmer.

Kenneth S. Rubin is Manager of Professional and Educational Services at ParcPlace Systems where he is coauthoring a book on managing object-oriented software development projects and codeveloping a rigorous object-oriented analysis and design methodology. He has been affiliated with the Center for Human-Machine Systems Research at Georgia Tech, Advanced Decision Systems, and the Hughes Aircraft Company. He received an M.S. in Computer Science from Stanford University.

Becoming more predictable

Smalltalk provides an excellent platform for incremental development. Prototypes can be built to clarify poorly understood requirements. Design can complement prototyping efforts to produce production-quality code. The benefits of incremental, iterative software development are enormous. Large, complex systems have a good chance of meeting customer requirements when they ship. Concepts can be validated before major resources are committed and schedules finalized. Functionality can be routinely added to an existing application base without things grinding to a halt. Object-oriented designs and programming languages are a real aid to incremental, iterative development. The modularity, flexibility, and encapsulation that objects provide makes incremental development practical.

Managing an incremental development can be extremely challenging. This is particularly true when the development team is new to both object technology and an incremental development process. I know of no magic formula that guarantees success, but life can be a lot easier if some checks and balances are applied. It is possible to design (and redesign) in an iterative, incrementally developed object-oriented application. Predictability (and taking action to become more predictable over time) is the key to making it work.

LIFE IN THE FAST LANE

Iterative, incremental development projects are typically started for several valid reasons:

1. Requirements for parts of the software are unclear. The plan is to develop prototype code, get feedback, assess what needs to be done, and then do it.
2. New hardware or complex processes need validation. Given long lead times, it may be necessary to write code that exercises new system functionality long before a detailed design is finished.
3. In even moderately complex applications, it is difficult to complete all subsystems at the same time. In fact, planning for a single massive integration phase is risky business. Confidence can often be gained if the system is brought alive in planned phases. With phased development, temporary functionality needs to be provided to make the things work. These interim parts are replaced

over time (if things go according to plan) with well-designed, production-quality code.

These are sound reasons. There are also situations where projects slide into an iterative whirlpool due to lack of leadership and planning, neglect, or inexperience. One danger to avoid (regardless of how you embarked on an iterative development) is wandering for too long without making significant progress, jeopardizing the entire project. Another risk to avoid is continual backtracking to fix things up when premature decisions don't pan out.

Odds can be improved by committing to and sticking with a process that encourages open communications. People need to consider the consequences of their actions or inaction. Even if you have to tinker and adjust the process over the course of the project, it's easier to put the appropriate force fields in place early than to inject change into an organization that has been lumbering along for a while. The objective is to encourage communication and thoughtful behavior, making the entire team function more smoothly.

SOME TYPICAL SCENARIOS

In the remainder of this month's column I want to discuss several tasks that are part of most incremental developments and offer advice on how to perform them effectively. Although the tasks undertaken by most object-oriented development teams are roughly equivalent, outcomes vary widely. What's startling to consider, however, is just how big an influence *anticipating* and *preparing* for change can have on the final outcome. It's crucial to think things through before reacting, and to provide enough information to allow others that same opportunity. I've learned some strategies for improving the outcome through direct personal experience. Observing the habits of people and organizations that meet or exceed their objectives has been another rich source of inspiration.

MAKING PROGRESS

Knowing precisely what's left to do and how long it will take is difficult to ascertain in a phased development. It's important to be as open and honest as possible when assessing status. Trust and teamwork play a big part. On one project, we lived and breathed the creed of incremental, iterative development. Not every project team will be so dynamic, have such

exciting chemistry, or be so committed to the project. But, I learned a lot from that project that I've found extremely useful on many other occasions.

We recognized our plans were estimates. They needed to be living, changing documents. We knew we couldn't do it otherwise. Team members didn't feel persecuted when they were behind on their initial estimates. If someone honestly didn't know where they were, the last thing they did was hide the fact. We described our designs and implementations as either being throwaway, experimental, temporary, works well enough, or of finished quality. We had guidelines for communicating project status that stated both our progress and confidence in our decisions and achievements.

When things didn't go as planned, we brainstormed about what it might take to get back on track. We set a new date for achieving measurable results, generated a list of action items, and kept moving. We acknowledged our situation, and actively sought help and advice from others when necessary. We were team players. It was important to do assigned tasks, but making the team succeed was the primary objective. We freely debated the impacts of change (and our progress) not only with the design and development team, but also with marketing, manufacturing, and project management.

Management made it very clear that it was OK to say, "I don't know." You were expected take action to find answers, but you didn't have to bear the burden alone. The team helped develop alternatives. And most importantly, a messenger of unexpected news was *never* punished.

“

The most effective prototyping effort I have seen was pulled off by a team that felt certain that they'd require several prototyping cycles before finalizing their design.

”

DECIDING WHAT TO PROTOTYPE

Determining what you want to prototype before starting is important. Sloppiness and unpredictability on the part of some prototyping efforts has gained prototyping an undeservedly bad reputation. If you have a clear set of objectives and a plan of attack, it will be much easier to get management to buy in. It also improves your chances for developing a meaningful prototype. It's vital to explore options before committing to any serious prototyping. What's serious prototyping? Spending more than a couple of weeks.

Take enough time to collect your thoughts and set objectives. One way to ensure that you've done enough homework is

VOSS

Virtual Object Storage System for *Smalltalk/V*

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

- Transparent access to Smalltalk objects on disk
- Transaction commit/rollback
- Access to individual elements of virtual collections and dictionaries
- Multi-key and multi-value virtual dictionaries with query by key range and set intersection
- Class restructure editor for renaming classes and adding or removing instance variables allows incremental application development
- Shared access to named virtual object spaces
- Source code supplied

Some comments we have received about VOSS:

"...clean ...elegant. Works like a charm."

—Hal Hildebrand, Anamet Laboratories

"Works absolutely beautifully; excellent performance and applicability."

—Raul Duran, Microgenics Instruments

logic
ARTS

VOSS/286 \$595 (\$375 to end of February 1992) + \$15 shipping.
VOSS/Windows \$750 (\$475 to end of February 1992) + \$15 shipping.
Quantity discounts available. Visa, MasterCard and EuroCard accepted.
Logic Arts Ltd. 75 Hemingford Road, Cambridge, England, CB1 3BY
TEL: +44 223 212392 FAX: +44 223 245171

to write things down. If you have trouble summarizing objectives, you aren't ready to launch into prototyping. You need to be able to state both what you hope to accomplish and how you intend to do so. Summarize any burning issues or questions. Communicate what you know and what you want to find out.

To clarify objectives, discuss the prototype with people whose perspectives differ from your own. It is fair to state a preferred course of action, but be willing to listen to other ideas. It's best if you can bounce ideas off someone who is both receptive to your ideas and a good critic. Brainstorm alternatives. Listen carefully to others' comments and criticisms. Don't shoot down new ideas. Mull things over.

Do a paper design of the parts you think you understand. This preliminary design probably won't be worked out in much detail. But be prepared to discuss key objects and architectural strategies. Talk to those who are reviewing your ideas in their own language. If your audience understands objects, talk about objects. If they don't, you can either spend time educating them, or speak to them in their own terms. I have had several meaningful discussions with people with an electronic engineering background. I found it useful when presenting my prototyping ideas to draw analogies with phased hardware development.

Setting objectives doesn't require a lot of time. What's appropriate obviously depends on the duration of the prototype. It's perfectly reasonable to spend a few days setting goals before a month-long prototyping effort. A six-month effort

might require a few weeks to set clear objectives. Without such forethought, it's difficult to know if you are even working on the right problem.

BUILDING A SUCCESSFUL PROTOTYPE

The most effective prototyping effort I have seen was pulled off by a team that felt certain that they'd require several prototyping cycles before finalizing their design. Knowing that, they planned for incremental, iterative success. I wouldn't characterize their efforts as random prototyping. They didn't just build something and keep tinkering with it until they had a final product. They documented what they expected to accomplish (and what issues they wouldn't address). They made sure others bought into their concept of prototyping. They made it clear to management that they wanted to design and implement prototypes to gain understanding. They didn't hope or expect a final result would "pop out" if they were lucky. They set milestones, and measured results along the way. They spent as much time assessing their prototypes as they did building them. They actively solicited advice and expertise when they felt uncertain. They spent a lot of time analyzing whether their design would be flexible enough and whether it would scale to accommodate future system requirements.

They went through several prototyping cycles on a major part of a large, complex system without ever being on the critical path. Were these people more brilliant or harder working than their teammates? They were skilled and had over 10 years of experience. But they weren't the only bright stars on the team. Instead, I'd characterize them as being in the habit of thinking before doing. They also believed very strongly in working smarter, not harder. They weren't constantly programming. Coding was a by-product of designing and reasoning about the problem and not its only manifestation, even during prototyping.

SOLIDIFYING SUBSYSTEM INTERFACES

In incremental development, the objective is to accommodate change without leaving interfaces too ill defined or soft and squishy. Working out interfaces is naturally an iterative process. Expect to refine them to match both class users' and class developers' needs. The key is to agree upon initial interfaces, and agree to *renegotiate* change. Changes should be made in context of their impacts on the overall system. Although responsibility for making change ultimately rests with the developers, system concerns need to be injected into the process of deciding what (and how) to change.

As subsystem designers work out the details of how services provided by their subsystem will be supported, they collect ideas for changes. Others using their subsystem will also find room for improvement. Evolving an interface requires teamwork. One way to foster teamwork is to publish *proposed* changes rather than to notify others after the fact. If no one responds to proposed changes, don't assume they agree by default. Things go more smoothly if people are given a chance to

understand and comment on proposed changes. Make sure people have enough time to absorb the impacts of a proposed change. What may seem minor to subsystem implementors can cause major repercussions elsewhere. Expect debate on alternatives before making any major change.

Smalltalk team programming environments make it easier to propose changes. Developers can pass back and forth workable alternatives without affecting others. They aren't a substitute for thinking things through. Effectively evolving interfaces requires adopting and promoting "systems think." On larger projects, one way to promote system thinking is to designate a system architect. The architect's initial role is to determine the initial structure and organization of the application, including subsystem interfaces. Throughout the project, the architect keeps on top of proposed changes, and actively works to mediate needs of the subsystems' developers and users.

Avoid the two ends of the spectrum: standardizing too early, or being so flexible that nothing can ever be agreed upon. Both extremes cause problems. If an interface is frozen, other parts of the application contort to fit. It isn't always appropriate that the first one "done" defines what is expected of others. On the other hand, if nothing is ever agreed upon, people are constantly in a reactionary mode, consuming lots of time adjusting and readjusting to shifting interfaces.

AN ALTERNATIVE TO JUST LIVING WITH THE CONSEQUENCES

There are many factors that go into a project's success. There's no substitute for persistence, intelligence, and commitment. Iterative and incremental development require extra attention to planning and designing and careful consideration of consequences. Individuals can make a difference by planning for change, rather than just letting it happen. Things won't work well if individuals go off and "do their own thing" ignoring the rest of the project. Successful iteration is fostered by teamwork and a willingness to accept and solicit constructive criticism. Improving predictability is a constant, ongoing process. ■

Rebecca Wirfs-Brock is the Director of Object Technology Services at Digitalk and coauthor of Designing Object-Oriented Software. She is the program chair for OOPSLA '92. She has over 16 years of experience designing, implementing, and managing software products. During the last eight years she has focused on object-oriented software. She managed the development of Tektronix Color Smalltalk and has been immersed in developing, teaching, and lecturing on object-oriented software. Comments, further insights, or wild speculations are greatly appreciated by the author. Rebecca can be reached via email at rebecca@instance.com. Her US mail address is Digitalk, 921 S.W. Washington, Suite 312, Portland, Oregon 97205.

Tips for improved Smalltalk reuse and reliability

“DON’T POUND NAILS WITH A CHAINSAW.”

Every programming language has certain features that can be sources of trouble when misused. These features are typically included in the language to solve specific issues but can, if misapplied, result in code that is bug ridden, difficult to reuse, and expensive to maintain. This month’s column covers a few of these Smalltalk features to help developers understand some of the issues associated with their proper use and the potential problems associated with their misuse.

GLOBAL, POOL, AND CLASS VARIABLES

Smalltalk provides mechanisms for sharing information via global variables, pool dictionaries/pool variables, class variables, and instance variables. Although each mechanism has a specific intended use, some of these mechanisms can be misused in ways that can impact code quality and reusability.

GLOBAL VARIABLES

Global variables represent data that are directly accessible to all of the classes in a program. Global variables are useful for holding the temporary code used in debugging, recording the count of window events that cannot be interrupted, and the rapid testing of prototype code segments. However, they typically have a negative effect on commercial software quality. The problems with global variables in conventional software development have been well documented by William Wulf and Mary Shaw¹ during the “X considered harmful” wave of papers. The problems with globals in object-oriented programs were succinctly summarized by Bertrand Meyer² as follows:

As different modules share global variables, they make each of these modules more difficult to understand, read, and maintain.

Global variables form a hidden dependency between modules. They are a major obstacle to software evolution because they make it harder to modify a module without having an impact on others.

The use of global variables violates encapsulation and the protective software “fire walls” that result. It is much easier

to make stand-alone, portable applications and classes without global variables.

They are a major source of errors. An error in one module may propagate to many others. As a result, the manifestation of the error may be remote from its cause, making it difficult to trace errors and correct them.

A less fundamental problem is that a global variable does not typically belong to one module in particular. Thus, it is not clear where to initialize the global.

Guideline—Avoid using global variables. Use class variables instead of global variables if the value is to be shared in only one class.

POOL VARIABLES/POOL DICTIONARIES

Pool variables or pool dictionaries are mechanisms for sharing the same information between instances of several classes. Pool dictionaries are usually used to hold dictionaries of related constants for a given application, e.g., ColorConstants and CharacterConstants.

Guideline—To avoid creating hidden dependencies, set the values of the class variables that are replacing pool dictionaries only in one common initialization method.

Without this practice, it is easy to change a number in one method and miss it in another. Class variable names can also provide semantic information about the use of a constant. For example, ButtonDown equals one and ButtonUp equals zero.

Guideline—Use symbolic constants and constant expressions to allow multiple dependencies to link to one or a small number of symbols in pool dictionaries.

In some OOP circles, using pool variables—as ubiquitous as they are in the base Smalltalk image—is considered a bad programming practice because they permit the violation of encapsulation. Therefore, in some situations, pool variables may not meet with the strictest guidelines of good encapsulation and OOP practice. Also, be aware that various dialects of Smalltalk handle pool variable inheritance differently. For example, in the Smalltalk/V implementations, pool dictionaries are not inherited while in Smalltalk-80, pool variables are inherited by subclasses.

Guideline—Use class variables with accessor methods instead of pool dictionaries/variables. If an object must be shared across several classes, create a separate new class to hold the pool dictionary object in a class variable of the newly created class.

Use class methods to tag constants instead of repeating them in many methods:

“Instead of declaring MyWindowControlKeys a pool dictionary of MyWindow use the following technique”

```
Object subclass: #MyWindowControlKeys
  instanceVariableNames: “
  classVariableNames: “
  ‘MyWindowControlKeys ’
  poolDictionaries: “!
```

!MyWindowControlKeys class methods !

```
myWindowControlKeys
“Use the lazy initialization technique to create and initialize the
dictionary if necessary in this get accessor method”
```

```
MyWindowControlKeys isNil
  ifTrue: [self initializeMyWindowControlKeys].
^MyWindowControlKeys.!
```

```
myWindowControlKeys: aDictionary
“Set the value of the instance variable using this setter accessor
method”
```

```
MyWindowControlKeys := aDictionary.!
```

```
initializeMyWindowControlKeys
“Use a common initialization method to add a key constant to the
MyWindowControlKeys dictionary .”
^(self myWindowControlKeys: (Dictionary new)) at: #F9Key put: 120.!
```

```
at: aSymbol
^self myWindowControlKeys
  at: aSymbol
  ifAbsent: [nil].!
```

```
at: aSymbol ifAbsent: aBlock
^(self includes: aSymbol)
  ifTrue: [self myWindowControlKeys at: aSymbol]
  ifFalse: [aBlock value].!
```

```
includes: aSymbol
^(self myWindowControlKeys includes: aSymbol).! !
```

```
Object subclass: #MyWindow
  instanceVariableNames: “
  classVariableNames: “
  poolDictionaries: “!
```

!MyWindow methods !

```
keyInput: anInteger
“Private - Process a key input to check if the pane should be cycled.”
anInteger = (MyWindowControlKeys at: #F9Key)
  ifTrue: [Notifier cycle].! !
```

CLASS VARIABLES

Class variables are shared only by the instances of the single class in which they are declared. Class variables contain the same value for all instances of the assigned class.

A class variable is often a good replacement for a global variable. The class should include the protocol necessary to initialize the global, access it, and modify it as necessary.

For example, suppose there is a class called User representing users of a system. Instead of using a global variable to store all the users, define a class variable called Users. The class protocol added to class User might include:

```
addUser:
deleteUser:
deleteUser:ifAbsent:
checkForUser:
users      “to return the collection of users”
```

Guideline—Use class variables for shared components between all instances of a class and its subclasses, and as a public interface for all classes.

ELIMINATE CASE ANALYSIS

In OOP, case analysis is the practice of testing an object for some criteria to determine what kind of action needs to be taken:

```
anObject isMemberOf: Rectangle
  ifTrue: [anObject drawRectangle].
anObject isMemberOf: Circle
  ifTrue: [anObject drawCircle].
anObject isMemberOf: Line
  ifTrue: [anObject drawLine].
```

The problem with this kind of programming technique is that it leads to a combinatorial explosion for large systems where many different objects need to be tested and appropriate methods need to be dispatched. *The practice of case analysis in large systems generally results in code that is more difficult to reliably maintain. Subsequent developers must find every relevant case statement in the system and make sure that it is properly updated.* Another negative side effect of this practice is that case analysis greatly reduces the reusability of the code. Using case statements, a new developer will be required to modify all the existing case statements to accommodate a new object.

The concept of polymorphism was designed to provide a simple and efficient solution to this problem. Briefly, with polymorphism there are many objects that respond to the same command, each taking the necessary steps to complete the actions of the command. The above problem can be greatly simplified if each object has a common command such as draw that it implements so that wherever an object needs to be drawn, the following message is sent:

```
anObject draw.
```

A draw method needs to be defined only once for each object, as follows:

```
<Rectangle> draw
    "Code to draw a rectangle"
```

```
<Circle> draw
    "Code to draw a Circle"
```

```
<Line> draw
    "Code to draw a Line"
```

Different but appropriate actions will be taken by each object to implement its own drawing. When properly implemented, any previously existing code that dealt with displaying objects can now be readily reused under this scheme. The developer of a new object need only take advantage of the existing code to create a draw method for the new object. In addition to greatly simplified programming, polymorphic programming uses the message dispatching scheme for achieving the desired result, which is usually a much quicker mechanism than any case statement. Although it may take more effort initially, *case statements can always be reworked into polymorphic messages.*

CHECKING FOR CLASS MEMBERSHIP

In object-oriented languages, polymorphism can also replace checking the class of an object. Different classes handle the same message differently. Using code with multiple calls to `isKindOf:`, `isMemberOf:`, or code of the form:

```
"class Window instance method"

resizeToMaxScreen
    (self class == FixedWindow)
        ifTrue: [self resizeNotAllowed]
        iffFalse: [self resize]
```


is often an indication of a function being in the wrong class. Replace these statements with a message to the object whose class is being checked. Create methods in the various classes of the object that respond to the message. Each method should contain one clause of the cases. An example of avoiding case analysis for the above problem might be to have the following code instead:

```
"class Window instance method"
    resizeToMaxScreen
        self resize


"class FixedWindow instance method"
    resizeToMaxScreen
        self resizeNotAllowed
```

Sending the message `resizeToMaxScreen` to an instance of either window class will result in the receiver window sending the correct message to resize itself or not.

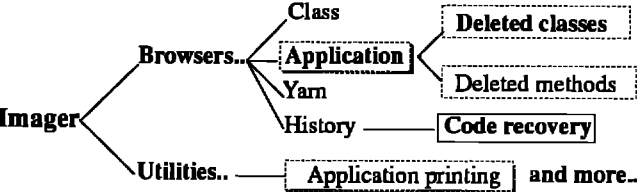
Guideline—Avoid explicitly checking the class of an object. Using these case statements is almost always incorrect.




**Smalltalk/V users: the tool
for maximum productivity**



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



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



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

MULTIPLE DISPATCHING

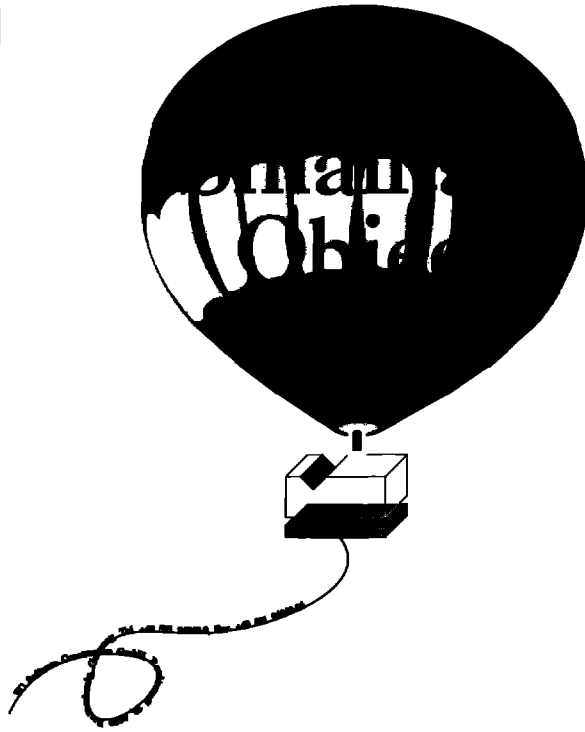
Even messages composed of more than one argument can be simplified using the double dispatching technique described by Dan Ingalls.³ This problem can be illustrated by the following summary of Ingalls' examples where the object to be displayed might be routed to a screen, a printer, or a clipboard. In this case, a programmer might be tempted to write the following case statements:

```
<Rectangle>displayOn: aPort
    aPort isMemberOf: Screen
        ifTrue: ["code for displaying on the screen"]
    aPort isMemberOf: Printer
        ifTrue: ["code for displaying on the printer"]
    aPort isMemberOf: ClipBoard
        ifTrue: ["code for displaying on the clip board"]
```

Although the above code is local to the specific object to be displayed, it will still be difficult to extend and to maintain. The solution to these "doubly polymorphic" situations is to use a relay method in each object to be displayed to further dispatch on the port as follows:

```
<Rectangle>displayOn: aPort
    aPort displayRectangle: self
<Circle>displayOn: aPort
    aPort displayCircle: self
<Line>displayOn: aPort
    aPort displayLine: self
.
.
.
```

ODBMS



ODBMS
The Objectoriented Database for Windows 3
and OS/2.

ORDER NOW !

- The ODBMS - Complete Version
- The ODBMS - Programmer's Version
- The DSSDe - Distributed Smalltalk
Software Development environment

More applications using ODBMS including
the exiting combination of ODBMS and
SQL are available.

VC Software Construction
Petritorwall 28
3300 Braunschweig
Germany
Tel. +49 531 24240-0
Fax. +49 531 24240-24

To complete the dispatching, one now only needs to define methods for each of the display port classes as follows:

```
<Screen>displayRectangle: aRect
    "Code to display a rectangle on a screen"
<Screen>displayCircle: aCircle
    "Code to display a circle on a screen"
<Screen>displayLine: aLine
    "Code to display a line on a screen"
```

and similarly for the other objects to be displayed:

```
<Printer>displayRectangle: aRect
    "Code to display a rectangle on a Printer"
<Printer>displayCircle: aCircle
    "Code to display a circle on a Printer"
<Printer>displayLine: aLine
    "Code to display a line on a Printer"
```

By following this approach, one can add new objects to the system without having to tamper with the existing code by only defining the relay message in the new class and the corresponding display method in each port class.

Guideline—To obtain the intended benefits of reusability, developers should avoid case statements from the start and avoid using case analysis to check the values of variables.

CONCLUSION

Smalltalk has many powerful features that can result in the efficient development of commercial-quality code. New developers should take a few minutes early on in their Smalltalk experiences to understand these issues so that they can significantly improve the reusability and reliability of their code from the start. The sooner one develops good programming style and techniques, the more valuable the effort will be to the programmer and to others in the future. ■

REFERENCES

1. Wulf, W. and M. Shaw. Global variable considered harmful, *ACM SIGPLAN Notices* 8, 1973.
2. Meyer, B. Bidding farewell to globals, *Journal of Object-Oriented Programming* 1(3): 73-76, 1988.
3. Ingalls, D. A simple technique for handling multiple polymorphism, *Proceedings of OOPSLA '86*, Portland, OR, 1986.

Ed Klimas is managing director of Linea Engineering, Inc., a supplier of custom object-oriented based solutions for industrial applications. He can be reached at (216) 381-8493. Suzanne Skublics is education manager at Object Technology International. Suzanne can be reached at (613) 228-3535. Ed and Suzanne, along with Dave Thomas, are coauthors of an upcoming Addison-Wesley book titled Smalltalk with Style that covers these and many other issues associated with commercial Smalltalk-based code development.

Paint palettes (taking control in Smalltalk/VPM 1.3)

This installment of GUI Smalltalk will take you on a journey. In the process of creating a palette, we will make use of an OS/2 2.0 control, take advantage of VPM's easy PM-subclassing feature, try out a feature of ViewManager, and override several PM behaviors. As an example use of the palette we will use it to enhance the old example application *FreeDrawing*. The palette will be used as an alternative to the menu for selecting drawing colors.

PRELIMINARIES

We will start off easy by filing in *FreeDrawing*. The necessary files can be found in the subdirectory *EXAMPLES\FREDRWNG*. Follow the instructions in the file *FREDRWNG.TXT*. To be safe, create *FreeDrawingWithPalette* as a subclass of *FreeDrawing*. This way we can experiment without losing the original *FreeDrawing*. Remember to include *PMConstants* as a pool dictionary for the new subclass.

Next install the the new control *ValueSet*. The necessary files can be found in the subdirectory *EXTRAS\VALUESET*. Follow the instructions in the file *VALUESET.TXT*. *ValueSet* is documented in the file and in the *Smalltalk/V PM Programming Handbook Supplement*.¹ Briefly, *ValueSet* is an array of rectangles where only one rectangle may be selected at a time. The rectangles may contain bitmaps, icons, colors defined in either of two ways, or text. *ValueSet* seems ideally suited for use in palettes.

A FIRST PASS

We will add the palette as a view of *FreeDrawingWithPalette*. The new view will contain one subpane, a *ValueSet*. The code is straightforward with one exception, the initialization of the contents of the *ValueSet*. To set the color of row 1 column 1 to red, one would normally write `aSubPane colorItem: ClrRed row: 1 column: 1`. But in this example, the colors are kept in a two-dimensional array (an array of arrays). So the loading is done in a two-level loop:

```
createPaletteView
    "Create and add a palette to my list of views.
    | aTopPane aSubPane |
    aTopPane := PaletteTopPane new.
    aTopPane framingBlock: [:box | 10@200 extent: 100@120].
    aSubPane := PaletteValueSet new.
    aSubPane
        framingRatio: (0@0 corner: 1@0.8);
```

```
when: #select perform: #colorSelectedFromPalette: ;
owner: self;
rows: 2 columns: 2.
1 to: 2 do: [:aRow |
    1 to: 2 do: [:aColumn |
        aSubPane
            colorItem: ((self paletteColorArray at: aRow)
                at: aColumn)
            row: aRow
            column: aColumn] ].
aTopPane addSubpane: aSubPane.
self addView: aTopPane
```

The supporting methods are:

```
paletteColorArray
    "Answer an array with the colors for the palette."
    ^Array
        with: (Array with: ClrRed with: ClrBlue)
        with: (Array with: ClrGreen with: ClrBlack)
colorSelectedFromPalette: aPane
    "A selection was made in the palette view.set
    the drawing color to the selection in the palette."
    | index color |
    index := aPane selection.
    color := (self paletteColorArray at: index y) at: index x.
    self colorSelected: color.!
```

The quick and dirty way to add the palette view to open is to copy it from *FreeDrawing* to *FreeDrawingWithPalette*. Add the line `createPaletteView`; right before `openWindow`. Now try it out by doing `FreeDrawingWithPalette new openOn: 'temp'`.

One nice feature you will notice is when the main "Smalltalk/V Paint" view is closed the palette view also closes. One slightly annoying characteristic is that once a color is selected from the palette you can not immediately draw on the main view. You first have to click on the main view to get its attention (make it active). Then you can continue drawing.

A SECOND PASS

The quick fix is to set the global Smalltalk variable, *CUA*, to true. Now there is no delay in drawing on the main view, but there is also no delay in going between any two Smalltalk windows. And we still have one view becoming active, then the other. Some people find this distracting. What we would like to do is modify the palette so it does not take control. This way the title bars do not change colors when the palette

```
super initialize.
self receiveAllWindowsMessages.
```

Now we have control of `button1down`. Clicking on the palette no longer deactivates the main drawing window. It also no longer changes the selection in the palette. Along with control comes responsibility. To determine and set the selection, add the following method:

```
button1Down: aPoint
    "The user pressed button at aPoint within me.
    Determine which value (rectangle) aPoint is
    in. Set my selection to that value. And send
    myself the select event. This last part keeps
    me consistent with my superclass.
    | selectedRow selectedColumn selectedCell |
    selectedColumn := aPoint x // (self rectangle width // columns) + 1.
    selectedRow := rows - (aPoint y // (self rectangle height // rows) ).
    selectedCell := selectedColumn @ selectedRow.
    self selection: selectedCell.
```

The last line returns control to PM. It tells the PM control `ValueSet` to select the cell the cursor was over when `button1` was pressed. PM responds by marking the selected item and sending a `wmControl` event. Smalltalk responds with a `syncControl`, which responds with an `asyncControl`, which generates the Smalltalk event `#select`. This puts us back into familiar territory. The next action is whatever we told the palette pane to do through the `when:perform:` when we created the pane.

AN EXERCISE FOR THE READER

There are several details you may want to clean up yourself. These include handling the `wmButton1dbclk:with:` event and getting rid of the menu bar. You may also want to give the dialog behavior of always floating above the main drawing view. The dialog behavior was covered in this column in the October 1991 issue. ■

REFERENCES

1. *Smalltalk/V PM Programming Handbook Supplement*, Digitalk, Inc., Los Angeles, CA.
2. *Smalltalk/V PM Programming Handbook Supplement*, Digitalk, Inc., Los Angeles, CA, p.469.

is used, and there is no delay going back and forth between the main drawing view and the palette.

The palette view becomes active upon processing the `wmButton1Down:with: message`. This is done under PM control. To keep the view from becoming active we need to take control from PM.

First make a new class, `PaletteValueSet`, as a subclass of `ValueSet`. Remember to give it the same pool dictionaries as its superclass. Copy the method `wmButton1Down:with:` from the class `Window` to the class `PaletteValueSet`. Remove the line beginning with `CUA`. (We want to respond especially when the palette is not active.) Change the return value from `nil` to `0`. (This keeps the default `winProc` from processing the message, which keeps the palette from becoming active.)

So far, so good. Except, the palette does not get the `wmButton1Down:with:` because the control has not been PM-subclassed. Subclassing in PM is not the same as subclassing in Smalltalk. When you PM-subclass a control you get messages before the control's `winProc` does. Returning `nil` tells the `winProc` to do its standard processing after your method is done. Returning `0` tells the `winProc` not to bother. (See Ref. 2 for more information.) Digitalk provides a neat way to take care of this with the message `receiveAllWindowsMessages`. This ensures the control is PM-subclassed. Make use of this by adding the following method to `PaletteValueSet`:

```
initialize
    "Private - Initialize the receiver."
    "This lets me take control from PM."
```

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 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, NC 27511, or by phone at (919) 481-4000.

The Best of comp.lang.smalltalk

Welcome to a new feature of *The Smalltalk Report*, a column summarizing the best Smalltalk-relevant discussions from USENET's comp.lang.smalltalk bulletin board.

WHAT IS USENET NEWS?

USENET is an informal computer network that carries electronic mail and discussion groups from all over the world. Although the discussion groups are referred to as "news," they are really open forums, similar to those available through electronic "bulletin boards" or commercial online services such as CompuServe or BIX.

The most significant difference between USENET and commercial systems is that USENET is based on a distributed (some would say anarchic) architecture. Rather than a few large, centralized machines, news is carried by an enormous number of smaller machines, without any central organization at all. Messages are distributed primarily through the Internet, a term that loosely describes the high-speed wide area networks linking universities, government institutions, and many large corporations. Machines that are not on the Internet usually get their news and email services by connecting (perhaps indirectly) to a machine that is.

A common feature of commercial online services are the "vendor forums," in which users can ask questions of company representatives and get authoritative answers. Because many of the networks over which USENET operates are government funded, they are not permitted to carry commercial traffic of this nature. Groups for discussion of particular products do exist, but these are primarily user groups, with only an occasional word from company representatives.

This is an important distinction. Although these user discussions can be very valuable, with good advice, ideas, and even snippets of code, one should take what is said there with at least one grain of salt. These are open forums, in which messages don't last very long, and the amount of thought that goes into the messages varies. *Just because someone speaks confidently on a subject does not mean they know what they're talking about.*

Of course a particularly outrageous statement will likely be corrected by someone more knowledgeable. It's even possible that the mistake will be politely pointed out, the original poster will see and admit the error, and everyone will come away having learned something. It doesn't usually work that way. Instead we find a message beginning with:

BZZZZT! Wrong! No points for effort. The correct answer is...

This is not designed to inspire calm and rational discussion. A typical response would be:

I always suspected everyone from <insert appropriate affiliation> had wallpaper paste for brains, but this surpasses even their usual standard of idiocy. My view is clearly supported by the following definitive sources...

“

Although these user discussions can be very valuable, with good advice, ideas, and even snippets of code, one should take what is said there with at least one grain of salt.

”

This soon becomes a "flame war," an endurance contest for electronic abuse which lasts until no one involved can remember the original point of debate and no one else is reading the messages.

FREQUENTLY ASKED QUESTIONS

One of the purposes of a user group is to answer questions. Often, to save answering the same questions repeatedly, the members of the group will put together a list of standard answers. In this spirit, I will lead off the column by providing answers to a few such questions.

WHERE CAN I GET FREE STUFF?

This is by far the most common question in any user group. We will omit the common variations of "Where can I get stuff even if I have to pay for it?" and "Should I buy this product?", because these are better dealt with by the ads, product announcements, and reviews elsewhere in this publication.

When people ask about free stuff on a computer network, they often want it to be accessible through the network. There are several ways of going about this:

- *ftp*. File Transfer Protocol (*ftp*) is a fast and easy way of transferring files over the Internet. A local systems administrator should be able to tell you if you can use *ftp* from your site.
- *E-mail*. Sites not on the Internet but with electronic mail access can often receive files by email. A message in a special format is decoded by the receiving computer, which automatically translates the files into a mailable form and forwards them.
- *Modem*. Other machines may allow you to call directly by modem and transfer files.

WHERE CAN I GET A FREE VERSION OF SMALLTALK FOR MY MACHINE?

I have only seen two freely distributable implementations of Smalltalk mentioned, and unfortunately neither of them is even close to competitive with the commercial versions. They are as follows:

- A Little Smalltalk is Timothy Budd's implementation of a portable Smalltalk system for UNIX, though it has been ported to MS-DOS, and probably to other machines. It has no graphical user interface. It is available for *ftp* from Oregon State University, cs.orst.edu.
- GNU Smalltalk is produced by the Free Software Foundation. It is another text-only Smalltalk, with a rudimentary user interface using the EMACS editor. It is also for UNIX, but runs on Atari ST computers as well. The standard *ftp* source is prep.ai.mit.edu, but like most GNU software it is available from many other places. For \$200 US you can get the source for it and whatever else happens to be on the same tape from the Free Software Foundation.

WHERE CAN I GET SMALLTALK CODE?

Because we all think code reuse is important, it's good to know what code is out there begging to be reused. Fortunately there's quite a bit available and it's beginning to be organized for easy retrieval, with centralized archive sites in Europe and North America. Ralph Johnson described the North American archive in the first issue of this magazine, but I'll mention it again for the sake of completeness. There's far too much code in these archives to describe, but in future columns I may highlight some items of interest. Here are some sources of Smalltalk code:

- The University of Manchester maintains an archive of Smalltalk-80 code, accessible through an email server. To get instructions and an index, send a message of the form:

To: goodies-lib@cs.man.ac.uk
 Subject: help; index

- In North America, the University of Illinois maintains an archive accessible by *ftp* at st.cs.uiuc.edu (an alias for 128.174.241.10). This contains a copy of the Manchester Smalltalk-80 archives along with lots of Smalltalk/V code, postscript versions of journal papers, and numerous other goodies. This archive can also be reached through an e-mail server by sending a message of the form:

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

If you have neither *ftp* nor email access, and can't find anyone who does to help you out, the archive is available by mail in various formats by sending \$200 US to William Voss at the Department of Computer Science, University of Illinois, 1304 W. Springfield, Urbana, IL, USA 61801.

- Commercial online services such as CompuServe and BIX often have Smalltalk discussion forums and accompanying file download areas. These are only semi-free, as you have to pay for access.
- ParcPlace runs a bulletin board called ParcBench (in California) with discussion areas and files available. This has the advantage of featuring vendor-supplied fixes and enhancements, but you have to be a paid customer of ParcPlace technical support to access many of the areas. The number is (415)691-6716, and is answered by a 2400 baud modem. There should be information on the service with your documentation. I haven't seen anything similar for Digitalk customers mentioned, but check the documentation.

LOST INSTANCES

A familiar cry on USENET is that of the Smalltalker who has gotten into a bad situation and can't find a way out. One of the common problems is "lost instances."

As everybody knows, Smalltalk is garbage collected. Objects that are no longer needed disappear quietly, without any intervention on the part of the programmer. This is true most of the time, but even the best programmers sometimes have difficulty convincing Smalltalk that certain objects are no longer needed.

The problem is more serious in Smalltalk/V than Smalltalk-80. In ST/V the definition of a class with instances cannot be changed. Even a few stray instances can make it impossible to work until they are tracked down and eliminated.

In ST-80 existing instances are changed to conform to the new definition when a class is recompiled. Lost instances take up space, but don't usually cause any further trouble. Usually, *Often, though, lost instances are a symptom of something more*

fundamentally wrong with the image. This is particularly true of instances of system classes like `ScheduledWindow` or `Process`. These deeper problems are often due to errors in user-interface code, e.g., windows that did not properly finish opening or closing. They can have bizarre effects, and be very difficult to fix. I've had the delete key stop working due to these kind of problems, and I still don't understand how.

INSTANCE EXTERMINATION

Given that lost instances may be or signal a problem, you need to know how to get rid of them as quickly and easily as possible. First, you need to find out if you have them. If you want to check for instances of a particular class, evaluate `aClass allInstances inspect`. If you have the problem, here are a few standard techniques for trying to solve it:

1. Check the obvious places. The most likely, and least troublesome, reason for lost instances is that you have a reference you've overlooked. Check global or class variables, as well as instances of other classes that might reference them. Be sure you don't have any inspectors or custom windows open. To minimize problems with globals, I avoid their use except for scratch storage, and I always start their name with a distinctive prefix. This makes it very easy to find all of my globals using `Smalltalk inspect`.
2. Start again. Go back to an old image that doesn't have this problem and file in your changes. If necessary, start from a clean image and file in all of your changes. Naturally, all Smalltalk programmers keep numerous backup images of different ages and file out their changes regularly, so this shouldn't be much trouble. Unfortunately, if the problem has been around for a while without you noticing you may have to back up quite a long way.
3. Use `become:`. If you're reluctant to abandon an image so soon, you can try forcibly eliminating the instances. Be warned that this may not solve an underlying system problem, and it can introduce new ones. Be prepared to start over unexpectedly. For classes that should have no instances, try evaluating something like:

```
ProblemClass allInstances do: [:eachLostInstance |
    eachLostInstance become: somethingInnocuous].
```

The central operation here is `become:`. This finds all references anywhere in the system to the block argument `eachLostInstance` and changes each one into a reference to `somethingInnocuous`. A good choice for `somethingInnocuous` is `String new`. This works in any dialect of Smalltalk. You're left with a number of empty strings, which still take up space, but nothing more serious.

A potentially bad choice for `somethingInnocuous` is `nil`, and unfortunately this is often what is suggested on USENET.

The danger is that this works perfectly well in Digitalk implementations, and in fact it doesn't even leave you with empty strings that take up space. In ParcPlace implementations, on the other hand, this will instantly kill your image. This is due to the different semantics of the `become:` operation in these implementations.

Digitalk uses a one-way `become:`, which works as described above, effectively changing one object into another. ParcPlace uses a two-way `become:`, which swaps the references, effectively interchanging the two objects. Think for a moment about the likely effects of changing every reference to `nil` in the system this way and you'll see why your image crashes.

4. Check the non-obvious places. If you are really determined to find and solve the problem with your image, be prepared for a mind-bending and time-consuming odyssey into Smalltalk's subconscious. Your tools are the methods `allReferences (ST/V)` or `allOwners (ST-80)`. Explaining how to do this is far beyond the scope of a frequently asked question, but here are a few words of advice.
 - Be prepared to go through several layers of indirection to find the real owner.
 - Realize that you will be creating references as you go. Try to minimize them. Don't keep inspectors open on objects you're trying to track down references to. Use long expressions instead, e.g., `MyClass allInstances first owners first owners`. Because the order in which instances and references occur is not fixed, this can be confusing.
 - References to associations indicate a dictionary entry. If the key is a symbol it may well be the Smalltalk dictionary. References in medium-sized, odd-looking collections may be compiled code or block contexts. Blocks (sort blocks are often culprits) sometimes hold onto their last arguments. Also, it's possible to get "lost processes," which then hold onto data referenced by the code they are supposed to be executing. I generally try setting everything to `nil` and hoping for the best.

Good luck. Are you sure you wouldn't just rather file those changes in? ■

Alan Knight is a researcher in the Department of Mechanical and Aerospace Engineering at Carleton University, Ottawa, Canada, K1S 5B6. He currently works on problems related to finite element analysis in ParcPlace Smalltalk, and has worked in most Smalltalk dialects at one time or another. He can be reached at (613)788-2600 x5783 or by email as knight@mrc0.carleton.ca.

Smalltalk at the University of Washington

In the Weird Languages Group at the University of Washington, we have used ParcPlace's Smalltalk-80 for a number of years. Not only have we used it as our personal computing environment, but we have also used it to develop a series of constraint-based systems: ThingLab-87, ThingLab II, and Kaleidoscope'90. However, while the overall goal of our research has been to provide constraint-based technology to users and programmers, each of the three projects has delivered this technology in different packages. The goal of group director Alan Borning's original ThingLab system was to provide a constraint-based simulation environment. However, after almost a decade of growth by accumulation, the ThingLab system no longer met our needs. Thus, we undertook redesigning and reimplementing it as ThingLab-87. In the process, we reexamined our ideas about constraints and produced constraint hierarchies. A constraint is a multidirectional, system-maintained assertion about the state of a system. Constraints are useful in programming languages, user-interface toolkits, simulation packages, and other systems because they allow programmers and users to state declaratively a relation that is to be maintained, rather than requiring them to write procedures to maintain the relation themselves. In general, there may be many interrelated constraints in a given application; it is left up to the system to sort out how they interact and to keep them all satisfied. In a constraint hierarchy, each constraint has a strength such that stronger constraints dominate weaker ones. Constraint hierarchies are especially useful in graphical layout or user interface applications because they allow the user or programmer to state preferences as well as requirements. For example, in a ThingLab physics simulation required constraints express the connectivity of rods and levers, Hooke's Law of spring force, and so on; weaker constraints attach the mouse and keyboard to the simulation. These user constraints are merely preferences because we don't want the user to be able to break the simulation—just to manipulate it.

In our second system, ThingLab II (also known as Minstrel), our research emphasis shifted from providing a tool for simulations to providing a tool for building constraint-based user interfaces. Thus the ThingLab II system concentrated on three specific issues: providing a very efficient constraint solver; providing a library of "Things" (objects) for user interfaces; and integrating the constraint system into Smalltalk-80.

This later integration turned out to be very difficult to do correctly. The two fundamental problems were that Smalltalk is an imperative object-oriented language whereas constraints are a declarative language; and imperative assignment and explicit control flow do not merge with multidirectional relations. Fortunately, we were able to design and implement a very efficient constraint-solving algorithm named DeltaBlue. Supported by the rapid prototyping features of Smalltalk, we were able to examine numerous variations of the algorithm and choose the best one for average base user interface. In addition, we developed a constraint compiler to find solutions to a fixed subset of constraints at compile time, rather than at runtime. This compiler uses Smalltalk's built-in compiler and reflective capabilities to translate constraints into executable byte-codes.

“

...the goal of our third project, Kaleidoscope'90, was to develop a constraint-imperative extension of Smalltalk-80.

”

The ThingLab II system was successfully used to prototype a number of user interfaces including a statistical visualization tool and a multimedia presentation control panel. Furthermore, the underlying constraint engine, DeltaBlue, has been ported from Smalltalk to other object-oriented languages and is being used around the world by both commercial software developers and other research labs.

During our experience with ThingLab II, it became clear that ad-hoc constraint-imperative integration techniques were not sufficient. We decided that proper support of constraints could only be accomplished in a language in which the imperative and constraint constructs were equal partners in the semantics of the language. In other words, we could not implement constraints on top of Smalltalk and hope for a safe, robust integration. Thus the goal of our third project, Kaleidoscope'90, was to develop a constraint-imperative extension of Smalltalk-80. After designing the language, we planned to im-

Coopers & Lybrand's AM/ST, Version 3.5

The SoftPert Division of Coopers & Lybrand, the worldwide accounting and management consulting firm, has been in the Smalltalk/V enhancement business about as long as anyone. Its Application Manager was an early entry in the application-based project manager browsers. The most recent incarnation of this product, AM/ST Version 3.5 is currently available for the OS/2 Presentation Manager and DOS-based Windows versions of Smalltalk/V. (The Macintosh and DOS implementations of Smalltalk/V are still being served by the older, less feature-rich AM/ST Version 3.0 and are not the subject of this review.)

In the years since its initial introduction, AM/ST has matured and evolved into a solid product. Where it once stood alone, however, AM/ST now stands in an increasingly crowded market segment—products that wrap real-world software engineering features around the early vision of Smalltalk as the tool of the lone, exploratory hacker/developer.

As object technology has continued to push into mainstream software engineering projects, the Smalltalk development environment has often stood hat in hand with no means to support real-world, team-based application development. Version control, performance testing, application delivery, and source documentation standards were missing for early Smalltalk teams of developers.

Nancy Martin of SoftPert, needing team development tools for her own division's Smalltalk/V work at Coopers & Lybrand, saw the market opportunity and began to work on early versions of AM/ST. The product has evolved into a multimodule system placed squarely, price- and feature-wise, in the middle ground of Smalltalk/V team-oriented application managers.

PRODUCTS AND PRICING

The AM/ST product suite currently consists of three interrelated modules: AM/ST, which is the base Application Manager product; the AM/ST Source Control extension; and the AM/ST Change Browser. The cost per user license for either the PM or Windows version of the AM/ST base module is \$475. The network file server-based Source Control can be added to this base product for \$1,595 for the first user and \$595 per each additional user. (Site licenses, at \$3,400 for AM/ST and \$7,495 for the AM/ST Source Control extension, are also available.)

The AM/ST Change Browser (single copies at \$195 and site licenses at \$1,170) is a full-featured Change Log browser that can be used independently of AM/ST. This tool gives you complete freedom in viewing and selectively restoring classes, methods, and evaluations from the Smalltalk Change Log or any file in the Smalltalk/V "chunk" format. While it makes an affordable and well-integrated enhancement to AM/ST, it is not strictly a component of AM/ST and thus is not a further subject of this review.

THE APPLICATION ARCHITECTURE

The traditional Smalltalk Class Hierarchy Browser and Class Browser are, as their names imply, designed with the Smalltalk *class* in mind as their organizing "molecule" with method and instance variable "atoms." AM/ST is designed with a higher-level "molecular" abstraction of the *application*.

An AM/ST application is a functionally based, logical group of classes and methods that are accessible and manipulable as a single unit. Under a traditional Smalltalk environment, applications are loose abstractions in the minds of the developer. Their implementation is sprinkled throughout the class hierarchy, in objects that interact to provide the attributes and behaviors required.

Under AM/ST, an application is explicit. An application consists of a group of contributing classes. The AM/ST Application Browser makes this application-based organization clearer. In Figure 1, the supplied `NetworkApp` application is shown to consist of three contributing classes: `Network`, `NetworkNode`, and `NetworkTest`. Actually, the implementation is found in the `Network` and `NetworkNode` classes. The presence of `NetworkTest` demonstrates a strength of application-based class access. That is, the implementation classes are logically organized and accessible with a class whose sole purpose during development is to encapsulate case-testing methods that exercise and verify the implementation.

Anyone familiar with the *Digitaltalk Handbook's* Network of Nodes example, from which this AM/ST application is derived, would recognize that the `NetworkNode` class is not limited to the name and position methods. This highlights another feature of AM/ST applications. A class within an application need only contribute the methods relevant to that application.

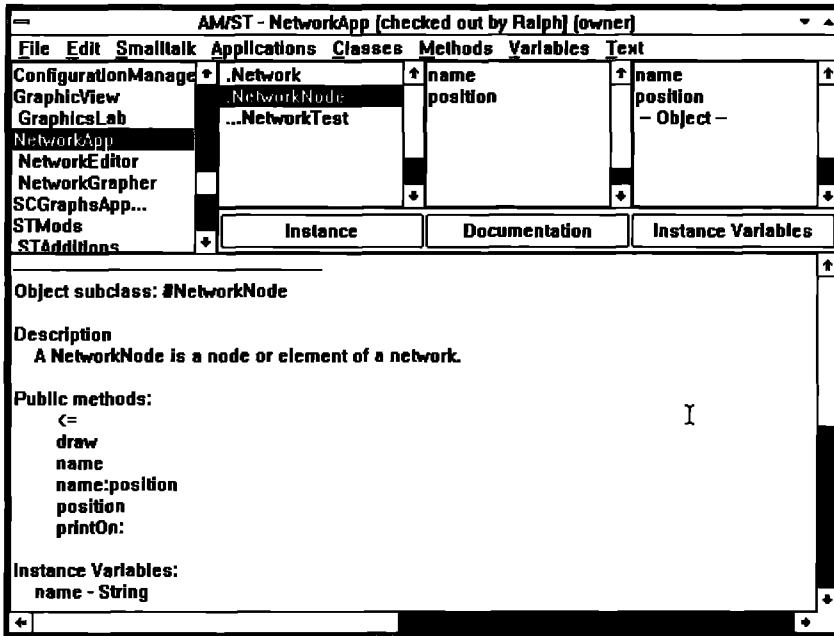


Figure 1. The AM/ST Application Manager Browser showing a documentation pane.

Further, an application can be functionally defined as a collection of *subapplications*. The *NetworkApp* example consists of the *NetworkEditor* and *NetworkGrapher* subapplications which each also use the *Network* and *NetworkNode* classes. The methods of these classes relevant to network construction are found under the *NetworkEditor* subapplication, and the graphics rendering methods under the *NetworkGrapher* subapplication.

A class may be included in any number of related or unrelated applications. You may, however, designate an application as an owner of a class. Owned classes may only have their class definitions modified within the owning application, although methods may be edited or added to the class from any application which uses the owned class.

As useful as it is to access the vast Smalltalk class hierarchy from an application browsing perspective, a real benefit of the AM/ST architecture is its ability to load and unload applications and to create application-specific Dynamic Link Libraries (DLLs). Under the AM/ST Source Control extension, this ability to load and unload applications is extended to network-based getting and putting operations with lock and unlock capabilities.

When you define or load an application into AM/ST, it creates and maintains a Smalltalk class with a name derived from the application name. AM/ST transparently creates and maintains class and instance "bookkeeping" methods in this application class that keep track of inclusion and ownership of classes and methods in the application.

AM/ST PRODUCT OVERVIEW

AM/ST is delivered on a single diskette, with whichever of its optional modules you may have purchased. It comes with a

140-page manual covering the base product and the source control extension. If you purchase the optional Change Log Browser, a separate 13-page manual is provided.

Not surprisingly, the AM/ST product was developed and is delivered as a collection of AM/ST applications:

1. *ApplicationManager*, the main application, which itself consists of the following subapplications:

- *AppManBrowser*, the application browser, with each of its panes implemented as subapplications.
- *AppManDLLizer*, an exciting facility for creating DLLs from AM/ST applications.
- *AppManInstaller*, which implements the load and unload facilities.
- *Apropos*, which adds flexible and powerful string searching features to the Smalltalk environment.

- *DynamicAnalysis*, which provides the method and block counting features.
- *FinderApp*, which implements the very useful Finder dialog used strategically throughout the Application Browser.
- *StaticAnalysis*, which implements variable and method cross-referencing.

2. *ConfigurationManager*, the Source Control extension (sold separately).

3. *ChangeLogBrowser* (sold separately).

4. *GraphicView*, which implements the graphical tree drawing utility.

5. *STMods*, a wide-ranging collection of changes and additions to the Smalltalk/V base classes.

THE APPLICATION MANAGER BROWSER

Your primary interaction with the features supplied by AM/ST is through the Application Manager Browser. As shown in Figure 1, the Application Manager Browser is essentially a Smalltalk browser with its application-oriented extra level of presentation organization. If all AM/ST did was deliver a well-designed application browser, it would be enough for consideration as a helpful extension to an unadorned Smalltalk/V environment. What the figure does not capture are some of the features that make the AM/ST Application Browser even more useful.

In concert with the ability to designate application owners of classes and, under source control, to check out and lock ap-

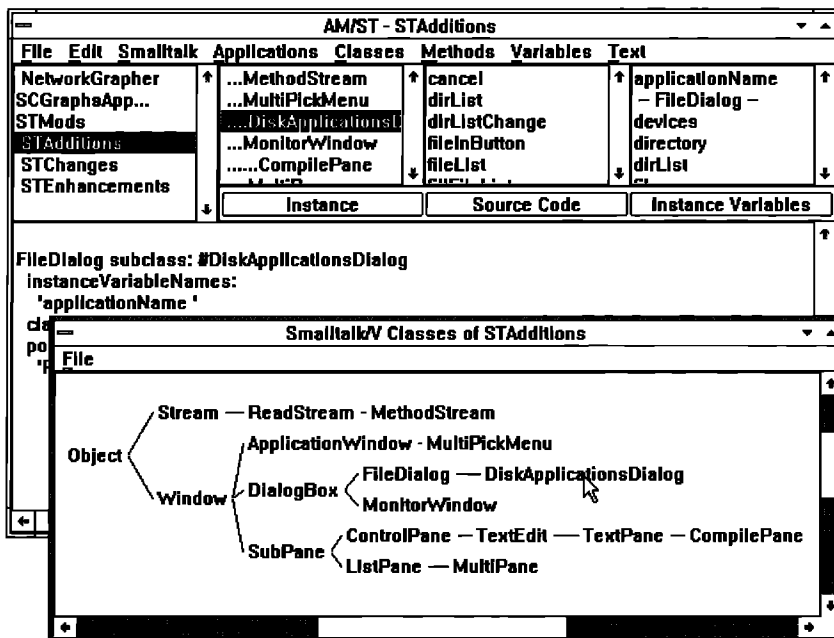


Figure 2. The AM/ST Graphic View of classes in an application.

plications, the Application Browser textpane intelligently sets itself to either editable or read-only capabilities. So while you may view all manner of class definitions, method source and related documentation, you will only be able to make changes if you are in the appropriate application, or if you are the currently active developer who has checked out and locked an application. Read-only text is displayed in a different color than editable, black text.

A nice feature of the Application Manager Browser is the Graphic View, which presents a hierarchical tree diagram of all Smalltalk classes or the classes in a selected application, as shown in Figure 2. Not only does this graphical view show you the inheritance relationship among classes, it is color coded and interactive. Classes included in the selected application are shown in blue type, their superclass parents in black, and the currently selected application class is shown in red. Clicking on an application class name will cause the Application Manager Browser to display the selected class.

The AM/ST Profile dialog lets you set a user id, default project directory, server/workstation time zone offset, and select to implement hierarchical or flat project directory structures as well as turn on and off the optional Source Control module. The time zone offset feature is particularly important in the case of remote developers working on a source-controlled application getting and putting through telecommunication connections.

The loading and unloading facilities make it quick and easy to add, update, or remove the collection of classes and methods grouped according to application-based functionality. You are given the option of loading classes and methods together or methods only to quickly update an application in which the classes required are already in your image. AM/ST

also has a convenient means to add pre- and post-load initialization code for applications and classes. This gives you the ability to perform preconditional processing as well as post-loading housekeeping.

DOCUMENTATION AND EDITING FEATURES

AM/ST strongly values and encourages full documentation. This emphasis is facilitated by the Source/Documentation toggle pushbutton in the Application Manager Browser. While you are encouraged to make traditional inline comments within the source code of your methods, AM/ST facilitates a standards-meeting implementation of full application, class, and method comment documentation. By toggling the Source button to Documentation, the lower textpane is devoted to editing and viewing documentary comments.

AM/ST uses user-modifiable templates to simplify documentation and to encourage consistent, full commentary. These templates are automatically inserted as you add or create classes and methods in your application. It is particularly helpful that structured documentation comments are maintained for classes as well as methods. Class comments are maintained in class methods named `classHeader`. (AM/ST maintains its application classes and all these `classHeader` and related "bookkeeping" methods transparently—the Application browser filters them from view unless you explicitly ask to see all classes and methods.)

Because the documentation templates are user definable, the extent to which you want to implement comments is fully under your control. Also, AM/ST implements its template-inserting features in documented methods defined in the Application class. This means that the templates may be defined locally to an application, allowing application-specific commenting standards.

Fully-customizable coding templates are also supported. Templates supplied include conditional, while True/False, block, do, collect, select, reject, popup menu specification, and dictionary lookup expressions.

The Apropos facility is a powerful, pattern-matching, GREP-like search tool for text strings in source code.

For the "hunt, cut, and paste" crowd of rapid prototypers, AM/ST provides a "method scrap" similar to but separate from the PM and Windows clipboard. This scrap can capture copies of multiple methods, simultaneously holding any number of class and instance variables. After gathering a number of methods from other applications' classes, you can return to your current application and, in one fell swoop, paste all the methods in the scrap into the currently selected class.

Every time you create or edit a class or method, a simple dialog pops up asking you what change was made. These brief comments are added with a time and date stamp to a revision history maintained in the class or method's documentation.

REPORTING FEATURES

AM/ST includes a wide-ranging collection of informative reports easily accessible through menu selections. The Applications, Classes, and Methods menus each have a hierarchical menu item that opens to one or more levels of specific reporting topics.

1. Applications menu reports

- class and global variable dependencies of a selected application
- listing of application directories
- class and method documentation for a selected application
- class ownership and class inclusion listing by application
- method indexes showing methods and the classes in which they appear
- application content inconsistencies, including classes owned by more than one application, classes neither owned nor part of an application, methods in more than one application, and methods in an application that no longer exist

2. Classes menu reports

- ancestor classes of a selected class
- dependency listing of classes and global variables directly referenced in the selected class
- various cross-referenced listings of all references to a selected class, all classes, or classes in a selected application

3. Methods menu reports include:

- applications containing a designated method
- methods in no applications
- methods in multiple applications
- methods not in a selection application
- methods with no senders
- various method cross-reference listings of the instance, class, shared, pool, and global variables used by a class

Variables menu reports include a global variables and cross-reference listing.

All reports are time and date stamped and generated into a Smalltalk/V Workspace window where they may be printed or saved to disk.

PERFORMANCE TUNING FEATURES

The dynamic performance tuning features are as easily accessed as reports. Both method and block counting are implemented. AM/ST adds instrumentation code to the compiled code of the profiled methods or blocks. This instrumentation code exists only in the compiled code and is not added to the visible source code. Instrumented methods are marked with a “#” character prefix and instrumented blocks are marked with a “&” prefix.

To add profiling instrumentation, you simply highlight a class or specific method and select Add Counters from a submenu of the Counters menu: either the Classes or Methods menus. Once instrumented, you need only turn on the counters and run your application. You then select Report from a submenu of the Counters menu item to view the method and block execution counts.

While execution counts is a rather crude means of method performance tuning, it is particularly well suited to testing coverage and code use measurements. You simply develop test suite methods that ensure all instrumented methods return a non-zero counter. In the event you are satisfied that all user interaction conditions are covered and you still return zero values, you have identified unnecessary methods or blocks that can be removed without affecting your application.

CREATING APPLICATION-BASED DLLS

The Smalltalk/V Windows and Presentation Manager (Version 1.3) have an exciting enhancement in which libraries of dynamically bindable objects are accessible through Dynamic Link Libraries. This object library DLL feature significantly enhances and simplifies the development and delivery of Smalltalk/V applications and development extensions. Dig-

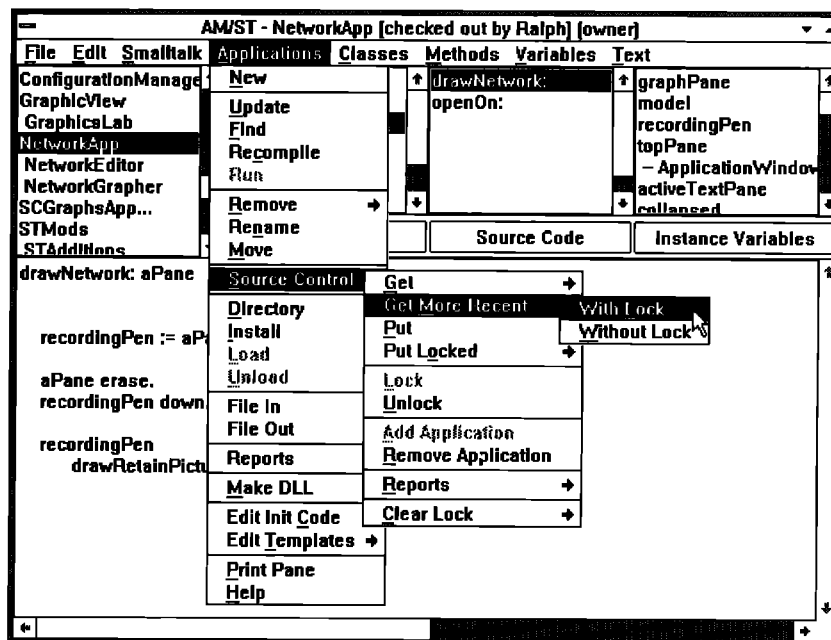


Figure 3. The AM/ST Application Manager Browser Source Control menus.

italk provides an Object Library Builder application, which is used to specify and generate the object library DLLs. Creating a DLL can be time and energy consuming. In particular, you need to fully understand what classes and methods must be included in your DLL to encapsulate the functionality you have in mind.

The application architecture of AM/ST is a perfect complement to the creation of object library DLLs. What goes into your DLL is the currently selected application. A simple **Make DLL** menu item selection transparently runs the Object Library Builder application, passing it all the required information from the Application Manager Browser. You may indicate a Development version, with source code included, or a Delivery version of your DLL.

In most cases, your application-based DLL is generated automatically. Occasionally, you will be prompted to resolve pointers.

SOURCE CONTROL FEATURES

Slipped easily into this already bulging feature set is the optional Source Code extension. When included in your image, the Source Code extension shows up as a *Source Control* menu item in the *Applications* menu, as shown in Figure 3.

To use AM/ST Source Control, you simply access a network server as a logical drive on your local workstation. You can then add an application from your image into the project-based directory structure of the server's application library. Once an application is entered into Source Control, it can be checked out with or without a lock. Locked applications can be retrieved by other team developers, but only the developer who set the lock can modify and put the application back for subsequent checkout.

AM/ST maintains a time and date stamp record of all applications under Source Control. You can ask Source Control to update all applications in your image older than the versions on the Source Control server.

For an application under Source Control, the load and unload facilities are deactivated when that application is selected. You are not, however, deprived from loading and unloading applications not under Source Control management.

PERSONAL EXPERIENCE

Our personal experience using AM/ST over several days was easily partitioned into two phases: resistance and appreciation. We used AM/ST to develop a variation on the Network of Nodes application under Source Control on a two-station, LanManager-based OS/2 network.

For the first two days, we were resistant. The application architecture adds a different perspective to the typical "I have the Universe at my command" feel of the traditional class hierarchy browser. With so many interactive features, the Application Manager Browser is pane and menu intensive. The shift in organizing perspective and the elaborate array-interactive features initially combine to intimidate.

Our resistance was maintained during initial, non-goal-directed exploration of the Application Manager Browser. Then we got to work on our test project. Three features combined to shake us out of our reticence: the method scrap, the DLL generator, and the get/load and put/unload facilities of the Application Manager Browser. While AM/ST has not yet achieved "how'd we ever live without it" status, it has garnered sufficient respect to be given continued, hopeful exploration. The following summarizes our reactions.

Among the features we like about AM/ST:

- The Method Scrap is a prototyper's dream. The efficiency of your "hunting parties" is increased tenfold. This feature alone is worth 30% of the cost of the basic AM/ST product.
- The application architecture combined with the DLL builder is tremendous. The application perspective neatly prepares you for generating modular, functional DLL object libraries. This feature is worth another 30% of the cost of the basic AM/ST product.
- The application-based load and unload facilities are a real-world project-based developer's salvation. By setting different base directories and using the hierarchical project directories setting, archiving project-specific versions of an application is easy. This feature would be especially valuable to any consultant or corporate developer working simultaneously on multiple, independent projects. This feature justifies another 30% of the cost of the basic AM/ST product.
- The long list of additional features of this product easily justify the remaining 10% of the cost of the base product. High on the list is coverage testing instrumentation; the flexible implementation of coding and documentation templates, together with the browser's separation of source code and header comments; the wide range of informative reports; and the class hierarchy graphical view.

Among the features of AM/ST we did not like are:

- You can easily find but not prevent inconsistencies in multi-application use of methods. A change in a method used by multiple applications is a change to all those dependent applications.
- Class ownership is by application, not by developer. This model recognizes the application developer, but ignores the contribution of class developers who can contribute reusable classes for read-only consumption by application developers.
- User profile sign-ons are not password protected, nor can application access be restricted by user id. Neglecting password protection and restricted access assumes an idyllic world of noncompetitive, nonprivate group development.
- AM/ST implements its own menu-specific, nonstandard help system. Each Browser menu includes a Help item as

its last item. While there is much valuable information in each of these Help items, this design does not follow the Windows or PM user interface standards.

OUR AM/ST WISH LIST

While some features of AM/ST that we did not like are matters of personal preference, the following three areas should be high on a list for future improvements:

- The Source Control module should be improved to support password user id protection and application access restriction via user id. Some form of a version resolution system is needed to allow multiple developers to work on the same application at the same time. It is hard to imagine a large project for which the one-developer-per-application granularity is sufficient.
- Application-specific method versions would be a significant improvement. This single enhancement would do much to alleviate the "tyranny" of the application-as-owner model of the current AM/ST. By allowing applications to overshadow methods, the potential for bringing the "team model" of class developers and application developers into the picture would be greatly improved.
- While the template-based documentation features of AM/ST are a positive contribution to real-world project development, they still depend heavily on the time, energy, and proper attitude of the individual developer. We would really like to see more automated documentation features. Automatic updating of class documentation by self-examination of its methods and instance variables to extract key documentation into the class header would alleviate the need for the developer to enter and maintain this information.

AM/ST IN THE COMPETITIVE MARKET

To determine the real value and quality of a commercial product it must be lined up against its competitors in the commercial marketplace. AM/ST competes in three product categories; application/project browsers, performance tuning tools, and source control.

APPLICATION/PROJECT BROWSERS

AM/ST falls in the middle range of price here. Third-party products claiming to add application- and project-based organization to Smalltalk/V are popping up in every OOP-related publication. Some of these new products allege to provide the same range of tools provided by AM/ST, but for a fraction of the cost. A number of these application/project browsers come with a change log browsing feature, which must be purchased as an extra-cost option with AM/ST.

Given the expanding field of possible contenders, a roundup review is certainly in order. However, if you just can't wait to make your purchase, our belief is that AM/ST has maturity and corporate development resources behind it

that will most likely be invested in increased product performance and stability over the long term. The price may come down; the quality will most likely go up.

PERFORMANCE TUNING TOOLS

AM/ST is again in the middle ground of a growing pack of products. In its favor, the AM/ST test coverage and identification of unnecessary code is sufficient and convenient. The method and block counting instrumentation is certainly easier to use than the Profiler supplied with both the Windows and PM versions of Smalltalk/V. However, the information you get in the AM/ST count reports does not approach the detailed information presented in the Digitalk profiler report.

“

AM/ST competes in three product categories; application/project browsers, performance tuning tools, and source control.

”

The AM/ST features cannot compete with a dedicated performance tuning tool such as First Class Software's Profiler. Profiler's ability to collapse and expand its performance measures based on recursive and other conditional forms is extremely important in assessing bottlenecks and optimization potential.

SOURCE CONTROL

Source Control is serious business. Based on the quality of the product and the utility of its features for your particular project, what you use can support or abuse. For large-scale projects involving team development with Smalltalk/V there are two main competitors: AM/ST and OTI's ENVY/Developer. Both of these products come in at the higher-price end of the spectrum, with ENVY/Developer topping the two.

ENVY/Developer is a strong competitor in team-based development. It supports the desirable application-specific method versioning but it also provides a built-in e-mail capability to facilitate communication between class-owning and application-owning developers. At the base configuration (requires 3 nodes at \$4,000 each), ENVY/Developer costs a lot. But the cost may be justifiable if your project requires multiple developers working on single applications or if class ownership by individual developers suits your engineering model.

If not, AM/ST would seem to be well suited to an organization with a non-real-time networked team, where remote ac-

...continued on page 28

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.

silence

Digamma Solutions is now offering **silence**, a collection of tools for project management and code delivery. **silence** provides a full-function professional environment at a low cost, without the need for expensive add-ons and upgrades, and drastically reduces the work involved in developing efficient, well-organized, and maintainable Smalltalk/V Windows code. **silence** for Smalltalk/V Windows is available immediately. **silence** for Smalltalk/V PM will be shipping the beginning of the second quarter of 1992. Information about **silence** for Smalltalk/V Mac is available.

For further information, contact Digamma Solutions, Spadina Ave., Unit 6, Toronto, Ontario, Canada M5T 2G5, (416)351-8833, fax (416)408-2850.

Smalltalk/V PM Relational Database Interface

Digitalk announced availability of its **Smalltalk/V PM Relational Database Interface**. This new extension to Smalltalk/V PM, Digitalk's object-oriented programming system for OS/2 Presentation Manager, provides an easy-to-understand interface to the OS/2 Extended Edition (EE) Database Manager and the Microsoft-Sybase SQL Server.

The new database classes implement the application programming interface (API) of EE Database Manager and SQL Server. As with Digitalk's other products, this allows the programmer access to these database interfaces by creating objects that understand how the underlying database functionality works. The programmer can then send messages to these objects to perform database functions, and the objects handle the underlying API. The classes include protocols to perform standard SQL operations, send SQL expressions to the database, and create such database objects a **SQLDatabase**, **SQLTable**, and **SQLRow**.

For further information, contact Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045, (310)645-1082, fax (310)645-1306.

continued from page 27...

cess is used to upload and download applications from a central repository.

Inching into the source control market are a few of those low-cost application/project browser products that claim to also have source control features. If cost is important, you'll have to lay out the features of these products side by side with those of AM/ST before you make your purchase decision. Does the less expensive product provide enough of the source control features needed on your projects to be a contender for your Smalltalk source control dollar?

THE BOTTOM LINE

The current Windows and Presentation Manager incarnation of Coopers & Lybrand's AM/ST is a definite forward step in the development of this product. At \$475, the basic module presents a cost-justifiable grab-bag of powerful extensions to the basic Smalltalk/V development environment. AM/ST must, however, be prepared to assess and respond to competition from the low end of application/project browsers.

As far as the source control extension is concerned, we see room for growth and improvement. Venturing into source

control is new territory for the otherwise mature AM/ST product. Given this company's history of commitment and efforts to improve its Smalltalk/V enhancement products, the Source Control module probably will mature and evolve over the next couple of years.

Finally, we encourage Coopers & Lybrand to bring their Smalltalk/V Mac, Smalltalk/V DOS and Smalltalk/V 286 versions of AM/ST up to the Version 3.5 standard. Smalltalk/V developers on these platforms deserve to have the same feature set currently available to their Windows and Presentation Manager colleagues. ■

Jim Salmons and Timlynn Babitsky are principals in JFS Consulting, offering corporate and peer consulting services in object technology. JFS Consulting specializes in user interface version control systems and technical documentation. Jim and Timlynn are coeditors of the annual International OOP Directory, A Guide to Object-Oriented Products and Services. They are the Exhibits Cochairs for OOPSLA'92 and will serve as the Conference Cochairs for OOPSLA'93.

WHAT THEY'RE SAYING ABOUT SMALLTALK

Excerpts from industry publications

SPECIFICALLY SMALLTALK

...“Object-oriented programming (OOP) and ODBMS are among the most important leading edge technologies today,” says [Patrick] Arnone [president of PRC Open Technology Inc.]. “It’s still a very small part of the industry, but we believe the market for OOP and ODBMS will grow throughout this decade. Right now we have to educate the community on the benefits of ODBMS, sort of like how the relational DBMS vendors did in the late ‘70s. It is not going to replace RDBMS technology, however, but rather will complement it very nicely.” Advantages of object-oriented over relational database systems include better integration of text, sound, images and graphics, as well as features suited for highly interactive application. As such, ODBMS is expected to be at the core of tomorrow’s multimedia and image processing software. “The early adopters, in both federal and commercial markets, will be in the engineering and scientific areas for computer-aided design and computer-aided engineering, certain segments of the MIS market, office automation system, multi-media network management, CASE, financial modeling and compound document imaging,” Arnone offers as examples....

...The ability to isolate and minimize platform-specific code within a program is one of the most important advantages of OOP. Only 2 percent of the code for ObjectStore for example, addresses the hardware platform it runs on. This characteristic of OOP and ODBMS simplifies porting efforts and makes the technology eminently suitable for client-server and distributed applications. “Portability is crucial,” says Ron Suarez, president of Arbor Intelligence Systems Inc., Ann Arbor, Mich., “especially for corporations that have already bought into different hardware platforms. Where a group of developers may be sitting at Unix workstations and people in accounting are using PCs and the graphics department is using Macs, you may have certain applications that they should all be able to use.” That portability brings important benefits to VARs and developers building products with OOP as well. “For a small company like us, working with these kinds of tools means that we can reach a much wider market,” points out Suarez. “We can sit at our machine of choice, which is the Mac, and develop applications that will run on Unix machines and on Microsoft Windows.”...“Our product, Smalltalk Nexpert Bridge, allows one to imbed a knowledge base produced using Nexpert within an application created with Smalltalk,” says Suarez. “The programs that you create are instantly portable from the Mac operating system to Microsoft Windows to Unix workstations. So a knowledge base developed with Nexpert can be used across all those platforms.”....

Brave new VARS, Jim Waddell, VAR Business, 12/91

...Virtually all object-oriented systems rely on relatively strict hierarchies. (Multiple inheritance, as implemented in C++, bends the rules, but only a bit.) Inheritance is useful..., but more useful still would be a paradigm that allowed programmers to assemble new classes from a network of smaller components — in a sense, “modular” objects. With more structural options at hand, we’d be less likely to suffer the pain and inconvenience of ripping up an existing hierarchy by the roots — or turning it inside out — to implement a new concept or feature...

...Too often, when you want to use a class that’s part of a rel-

atively deep hierarchy, you’ll find the program awash in code pertaining to its superclasses. Think of it this way: imagine you’ve invited a friend over for dinner. At the appointed time, he arrives — along with all his forebears: parents, grandparents, even the spirits of a few ancestors who are now, as it were, virtual. These ancestors stampede across the threshold and proceed to eat you out of house and home. Your friend explains, “Well, Ma taught me how to hold a spoon, so I need her here to help me eat. And Gramps taught me how to read, so he needs to be here in case I want to read anything. And Great-grandma...” The thought of anyone dragging his ancestors with him everywhere he goes is comical, but having to put up with relatives of a class you import from a large toolkit isn’t funny. Even with “smart linking,” which cuts down on uninvited “sibling” classes, the overhead can be daunting....

...One of the touted advantages of OOP is the ability to use “toolkits” of objects from which you can select classes and create subclasses according to your needs. But have you ever tried to combine pieces of two toolkits, with two different hierarchies arranged two different ways? Here, the ability to assemble objects from smaller components in customized, structured networks would free programmers from the tyranny of monolithic hierarchies that must be imported wholesale, bag and baggage...

...Unfortunately, few of the current crop of object-oriented languages are dynamic enough to comprehend such environments [as software which is flexible at runtime as well as at compile time]. Xerox’s Smalltalk can rebuild the innermost parts of the operating system as it is running (most current implementations share this flexibility to some degree), but the most popular object-oriented language, C++, is statically compiled and does not allow classes to be created or destroyed at runtime. A C++ program that deals with a dynamic object-oriented environment must support two kinds of classes that have two kinds of semantics: C++ classes, which are manipulated via the fundamental constructs of the language, and the environment’s classes, manipulated by the C-like, non-object-oriented constructs of the language. It’s a shame when the powerful, object-oriented features of a language can’t extend to the environment in which programs run. and it’s a poor idea, as William of Occam once noted, to multiply entities beyond necessity — to deal with two kinds of objects that abide by different sets of rules....

...Before “strictly static” OOP languages proliferate, like the IBM PC architecture, to the extent that there’s no turning back, we need to send our language designers back to the drawing board, exhorting them to seek new language models that gracefully accommodate both [the static and dynamic] worlds. Otherwise, we will be left with tools that are largely extensions of static, non-object-oriented languages and that don’t address our real needs in increasingly dynamic environments...

Trouble in Object City, Brett Glass, Programmer’s Journal, 12/91

...Most OOP advocates, like Object Management Group’s [John] Slitz, believe that OOP, usually written with C++ or Smalltalk programming languages, is “revolutionizing the industry.” Others, like Bill Moritz, owner of IQ Computer in Boulder [Colorado], disagree. “Everyone is jumping on the bandwagon. It’s the marketing buzzword of the ‘90s, like “user-friendly” in

the '70s and "intuitive" in the '80s. OOP has lost its initial, rigorous definition. It was a computer science and engineering tool; now it's just hype," he contends....

...Mark Hatch, president of the Alpha Marketing Group, which specializes in helping start-up companies bring OOP to market, cautions, "Reusability can be a curse if you don't find inherent function problems during testing. These problems are then carried through to other programs, like a virus. You don't always know how the objects will behave in different environments."....

...[Roger] Loeb [chief executive officer of Boulder's MarTech Group], among others, emphasizes that to write programs with reusable objects entails an entirely new way of thinking and looking at problems. "You must start the process with the right pioneers. Programmers can't often see through their specific task. They write procedures enabling them to solve problems in the way the problems are described, but they don't see the whole picture. You need people with a vision to understand how someone else may want to use their design, or object, in another situation. They must have anticipatory skills, the ability to generalize a problem. Systems are always changing; they must be able to deal with exceptions." Loeb adds that others can learn once the right people start out designing the initial objects. Since OOP is a relatively new concept, the effects of training programmers to shift their way of thinking to "a tolerance for ambiguity," as Loeb says, presently isn't known...

...The Object Paradigm Research Group, headed up by Dr. David Monarchi, a professor in the School of Business at the University of Colorado in Boulder, is researching the effects that OOP may have on programmers and on business. "We study how this technology can be transferred into businesses, how to implement these concepts. We're looking at the process of building object-oriented systems, how to perform object-oriented analysis and design, how to judge the quality of the design, what additional problems need to be solved to make this a commercially viable technology." Monarchi adds that it's difficult to quantify benefits because there haven't been many large commercial systems using OOP. "We don't have the experience to know, for example, what the long-term maintenance effects will be." Although he says that OOP will prevail in this decade, "there aren't many skilled in it or in teaching it. You can't just suddenly grasp the concept of OOP without using it."....

Trying to make software simple, Hilary Lane, The Boulder County Business Report, 12/91

OF GENERAL INTEREST

The five leading object-oriented DBMS vendors recently outlined their efforts to create object-oriented database standards for approval by the Object Management Group (OMG). The new group, called the Object Database Management Group (ODMG), consist of Objectivity Inc., Object Design Inc., Ontos Inc., Servio Corp., and Versant Object Technology Corp. For expedience ODMG is currently not inviting other vendors to join in its work, which includes drafting technical proposals that respond to the recently drafted OMG Object Services request for information...ODMG has defined a common object database perspective, which was used to help shape the efforts of the OMG's Object Model, which OMG has been working on since last spring...ODMG is now addressing database-specific technology areas beyond the scope of the OMG's Object Model. ODMG is considering an object-oriented query language, perhaps based on some of its members' own languages...Another task is to define an object-oriented database

interface with the OMG Object Request broker and distributed data management.

Five DBMS vendors agree to create OODB standard, Scott Mace, InfoWorld, 12/23/91

...Any menu or window on the desktop can be hooked into and subclassed. This means that anything is fair game, whether it's the listbox in your application that you need to enhance, or the menu bar in Aldus Pagemaker that needs an extra command or two. I have to admit, at first this made me think a little about the legal ramifications. However, as long as you've actually purchased the application, nobody can really complain of any wrongdoing. After all, you haven't actually modified anybody's code—just the way it interacts with Windows and other objects...

Subclassing Applications, Mike Klein, Dr. Dobb's Journal, 12/91

...But as we head into the 1990s, there is a new category of computer specialist that is about to emerge, people I call paraprogrammers. This new type of programmer could create some interesting new job opportunities for a lot of people who do not have specific computer backgrounds, but are willing to learn some new skills in order to get into the world of computers and business. Other business fields have specialists like paralegals and paramedics, and now the computer industry is about to get its own: paraprogrammers. The role of a paraprogrammer is linked to a significant new software programming concept called object-oriented programming (OOP). With this approach to programming, much of the programming code is actually pre-written by the programming language vendor, and the person working with this code draws on the objects, or object libraries, to write the bulk of the specific program...We actually have a type of paraprogrammer today. These individuals normally reside within an MIS department and are given the task of customizing a database or spreadsheet... Many times they create macros for repetitive tasks. Some of these are actually professionally trained programmers, while many of them have just learned a particular database language and are trained to create this custom layer of the program...Trained programmers will continue to play a very important role, especially in the creation of code that will be used for the commercial market. They will also become an important part of the management of these paraprogrammers, in the same way doctors or lawyers oversee the work of their paraprofessionals...

Industry Insight: new category of computer specialist, Tim Bajarin, Computer Currents: San Francisco Bay Area, 12/3/91

[Lou Mazzucchelli of Cadre]:...Traditionally, people have tended to concentrate on the functional behavior or the process behavior. I mean, what happens when this data turns into this data and then it goes over here and gets turned into something else. What object-oriented analysis does is borrow a lot of ideas from database design and information engineering. Before you do that, let's understand what the data space looks like. What are the things that the system is manipulating? Why do those things exist? How do those things relate to each other? It turns out that if you do a good job of that, you're actually beginning to identify candidates for objects that you might design into a system to support this space. So there's been a lot of work in the last year or two on ways to represent the design of an object-oriented program and then ways to transform that design into a reasonable implementation: and you're beginning to see a few products that do things like that — you'll see more....

Q & A: Lou Mazzucchelli, Cadre's vice president, Gregg Wendorf, The Sun Observer, 12/91

...Driving the shift to object-oriented programming is a change in the IS environment itself. "The desktop operating system is fragmenting: The old DOS standard is splintered into Windows and OS/2," says Eugene Wang, director of the languages group at Borland International Inc., the Scotts Valley, Calif., software vendor. "IS is nervous about choosing the wrong operating system, and object-oriented architectures are much more portable."...

...another reason for the slow acceptance of object-oriented programming is that, initially at least, it has been most available for languages that are not widely used for state-of-the-art programming. "Object-oriented code started in C++ and Pascal, which are not the IS languages of choice," says Rob Dickerson, VP and general manager of Borland's database business unit. "Object-oriented programming has to become more available in fourth-generation languages that have database-modeling semantics built in."....

A Reusable Revolution, John Parker, Information Week, 1/6/92

...[Joseph] Firmage's [president of Serius Corporation] current goal is much broader than creating a lone business application. He says he would like "to raise the level of software development away from code so the end user can construct application software." A more distant goal is to redefine the way software is sold, which he predicts will take about a decade. "What happens if mom-and-pop consultants can build Microsoft Excel in seven days?" Firmage asks. "What we're basically talking about here is customized software that can compete fiscally with packaged software." Adds Firmage: "We're proposing that you move power out of the hands of those large companies and closer to the users."....

Joseph Firmage Means Business — "Seriously," Dawn Smith, MARKETING Computers, 12/91

...[Peter] Meng [co-founder of Drochelmann/Meng & Associates, a multimedia integration company based in St. Louis] commented on a recent TIME Magazine article about multimedia which expressed the opinion that what multimedia lacks is the so-called "killer application" — something like the electronic spreadsheets or word processors that triggered the personal computer revolution. Meng thinks that this "killer application" will turn out to be not an application at all but an object-oriented operating system that gives users the capability of creating their own customized applications. One example he gives is that a user could create an individual application such as a spreadsheet with the numbers animated, along with sound accompaniment. Meng predicts that the recently announced alliance of Apple and IBM will do much to enhance and move forward the whole field of interactive multimedia, particularly in the matter of standards and system compatibility. Potential uses of interactive multimedia are virtually limitless, especially in the field of education and corporate training. Meng says, "In our fast-moving society, where it's vital to educate and re-educate people quickly, multi-media offers itself as an incredibly powerful training tool."...As for the future of multimedia,[co-founder Peter] Drochelmann says that even though we still live in basically a print world, he thinks that's changing. "I see multimedia replacing print. Television has been an interim step in this process. Eventually, I see the computer becoming an all-encompassing technology, integrated in the totality of our daily lives."

Drochelmann/Meng & Associates: a media resource integration firm, Carol Ellerman, St. Louis Computing, 12/91

...Analysis and design methods are already changing to accom-

modate objects. They will need to change even more to accommodate organic systems development. For objects, the trend is to move away from phased development towards task oriented development. In a method such as Coad/Yourdon, e.g., several layers are created, but it is not necessary to construct them sequentially. A distinction is still made between the requirements and the design of a component, but all the requirements need not be known before design may begin. Organic systems will evolve, as the name implies, over time. It will still be necessary to understand the requirements of an enhancement, but the realization of those requirements in software may be automated to the extent that there will be no formal design phase for an individual component....

Methodology: Developing organic systems, Adrian Bowles, Object Magazine, 1-2/92

...As a result of controllers and views, the application object is much less in control of its own destiny. It will only handle a subset of the events that it otherwise might. When it does respond to an event, the response is likely to be restricted to the application data. The days when the application ruled the interface are over. Losing autonomy isn't bad in itself. Given the enormous complexity of GUIs, overall control is best left to the application-independent GUI classes. By reusing view and controller classes, designers and programmers can largely limit their attentions to the design of application classes. However, reuse means that a programmer no longer has the control that comes from creating a program from scratch. As major league control freaks, programmers may find this pill a bit hard to swallow...

...The secrecy and lack of attention associated with GUI issues is ironic in the extreme. OO was largely created as a reaction to command line-based — DOS, for example — user-computer interfaces. OO pioneers like Alan Day wanted user interfaces that provided direct manipulation capability. This means that the objects the user manipulates in the "real world" are used in the computer system's interface. (Flight simulators are good examples of this concept. Virtual reality is the ultimate application of direct manipulation.) Object orientation is more than the communicating objects the programmer sees. It is a whole new way for most users to interact with computer system. Dynamic binding is a mind bender. But, there is also a significant paradigm shift when command line or character-based hierarchical interfaces are converted to GUIs. Unlike dynamic binding, however, the GUI part of the paradigm shift affects everyone — user, software developers, and methods specialists. The universal impact of the shift to GUIs is the greatest dirty little secret. No one will be spared the conceptual devastation of the OO paradigm shift, not even the users. How long before we hear the sound of shattered glass because users are trying to click their mice on their windows?

Antihype: OO's dirty little GUI secrets, John Palmer, Object Magazine, 1-2/92

... "Thirty percent of federal agencies are already using some kind of object-oriented programming (OOP) software," says Robin Rather, of Vienna, Va. - based Information Strategies Group, who recently completed a study of OOP in the federal government. She also discovered that another 25 percent of the agencies said they would be buying OOP technology in the next three years....

Company courts Federal market with object-oriented programming, Washington Technology 12/19/91



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

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

With Smalltalk/V, you don't.

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

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

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

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

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

Smalltalk/V

Smalltalk/V is a registered trademark of Digitaltalk, Inc. Other product names are trademarks or registered trademarks of their respective holders.

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

LOOK WHO'S TALKING

HEWLETT-PACKARD

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

NCR

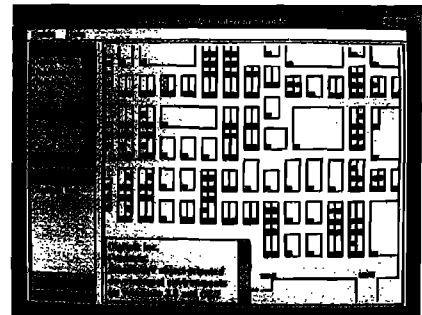
NCR has an integrated test program development environment for digital, analog and mixed mode printed circuit board testing.

MIDLAND BANK

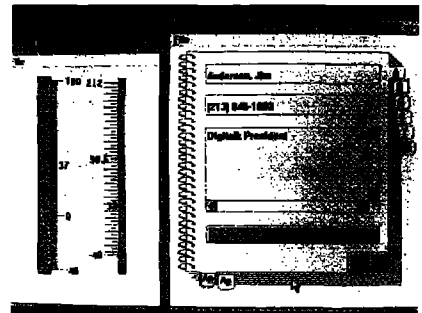
Midland Bank built a Windowed Technical Trading Environment for currency, futures and stock traders using Smalltalk V.

KEY FEATURES

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



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



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