The Smalltalk Report

The International Newsletter for Smalltalk Programmers

January 1992 Volume 1 Number 4

SHOULD CLASSES HAVE OWNERS? PERSPECTIVES FROM EXPERIENCE

By S. Sridhar

Contents:

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

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

TERMINOLOGY PRELUDE

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

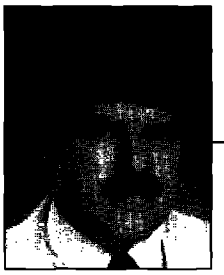
The granularity of a software component can be varied: a method, class, set of classes, set of sets of classes, etc. ENVY also supports an additional programming environment structure called an *application*. An application is a collection of defined and extended classes that together accomplish a well-defined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

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

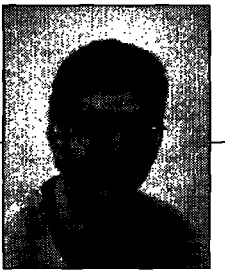
WHO IS BEST QUALIFIED TO FIX THE BUG OR WRITE THE NEW METHOD?

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

continued on page 4...



John Pugh



Paul White

EDITORS' CORNER

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

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

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

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

John Pugh Paul White

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by COOT, Inc., a member of the SIGS Publications Group, 588 Broadway, New York, NY 10012 (212)274-0640. Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: THE SMALLTALK REPORT, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada.

Putting Smalltalk To Work!

1980	Smalltalk Leaves The Lab.	We were there.
1984	First Commercial Version Of Smalltalk.	We were there.
1985	First Industrial Quality Smalltalk Training Course.	We were there.
1987	First Fully Integrated Color Smalltalk System.	We were there.
1988	Responsibility-Driven Design Approach Developed.	We were there.
1991	Smalltalk Mainstreamed in Fortune 100 Applications.	WE ARE THERE.
NEW!	First multi-repository, group programming environment.	*NEW!*

Smalltalk Technology Adoption Services

Technology Fit Assessment
Expert Technical Consulting
Object-Oriented System Design/Review
Proof-of-Concept Prototypes
Custom Engineering Services & Support

Smalltalk Training & Team Building

Smalltalk Programming Classes:

Objectworks Smalltalk Release 4
Smalltalk V/Windows V/PM V/Mac
Building Applications Using Smalltalk

Object-Oriented Design Classes:

Designing Object-Oriented Software: An Introduction
Designing Object-Oriented Systems Using Smalltalk

Mentoring:

Project-focused team and individual learning experiences.

Smalltalk Development Tools

NEW! Convergence/Team Engineering Environment™

Multi-user/shared repository development environment for teams creating production-quality Smalltalk applications.

Convergence/Application Organizer Plus™

Version management, development tools, and improved code modularity for individual Smalltalk developers.

Instantiations, Inc.

1.800.888.6892

continued from page 1...

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

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

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

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

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

ADDING CLASS EXTENSIONS

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

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

REWARDING REUSE

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

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

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

FLEXIBLE PROGRAMMING ENVIRONMENT

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

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

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

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

CONCLUSION

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

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

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

Rebecca Wirfs-Brock

Determining object roles and responsibilities

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

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

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

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

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

SUBDIVIDING LARGE CONCEPTS

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

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

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

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

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

COMPLETING A MODEL OF OBJECT INTERACTIONS

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

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

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

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

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

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

OBJECTS THAT DON'T FIT THE MODEL

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

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

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

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

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

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

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

REFERENCES

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

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

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

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

Juanita Ewing

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

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

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

WHAT ARE CLASS VARIABLES?

Classes can have:

- class variables
- class instances variables

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

The methods in ListInterface would look like this:

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

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

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

        ^ListMenu notNil

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

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

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

HOW ARE CLASS VARIABLES INHERITED?

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

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

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

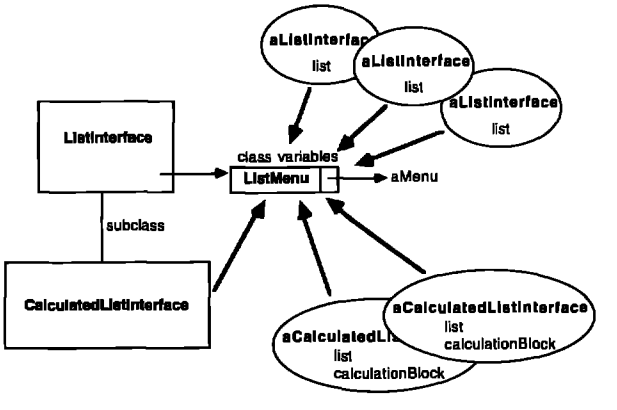


Figure 1. Class variables are refernced by subclasses and all instances.

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

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

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

HOW DO SUBCLASSES MODIFY CLASS VARIABLES?

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

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

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

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

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

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

        ^CalculatedMenu notNil

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

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

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

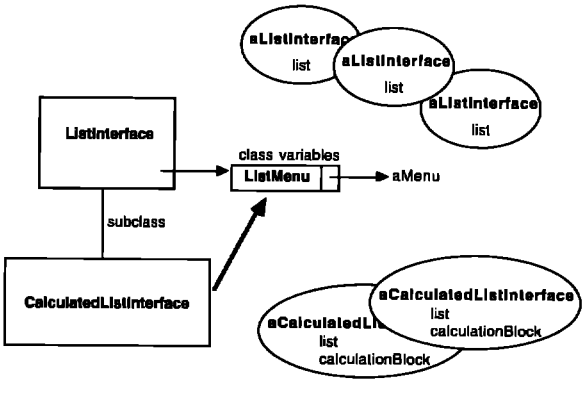


Figure 2. Coding conventions increase the reusability of classes implemented with class variables.

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

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

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

    menu
        "Return the list menu."

        ^ListMenu

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

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

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

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

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

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

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

    menu
        "Return the list menu."

        ^CalculatedListMenu
```

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

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

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

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

SMALLTALK COMES TO THE MAINFRAME, PART 2

Glenn J. Reid

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
( x:Array )
```

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

```
x at: 1
```

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

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

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

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

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

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

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

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

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

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

```
Number
    Integer
        SmallInteger
```

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

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

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

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

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

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

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

Listing 1.

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

ConstrainedAssociation class methods

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

ConstrainedAssociation methods

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

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

REFERENCES

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

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

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

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

PRODUCT REVIEW

Reviewed by Jon Hylands

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

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

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

HOW TO USE PROFILE/V

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

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

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

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

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

BUT MY CODE IS ALREADY FAST...

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

USING PROFILER TO OPTIMIZE A SAMPLE APPLICATION

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

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

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

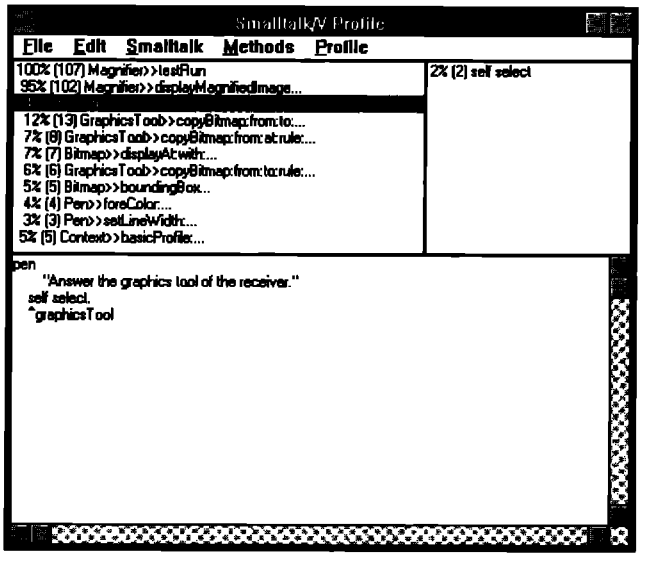


Figure 1. Initial profile.

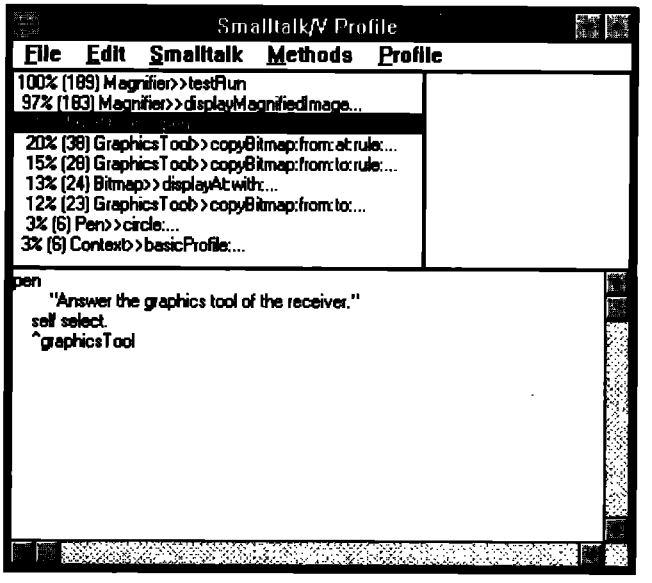


Figure 3. Final profile, with boundingBox message removed.

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

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

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

PROBLEMS WITH PROFILE/V

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

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

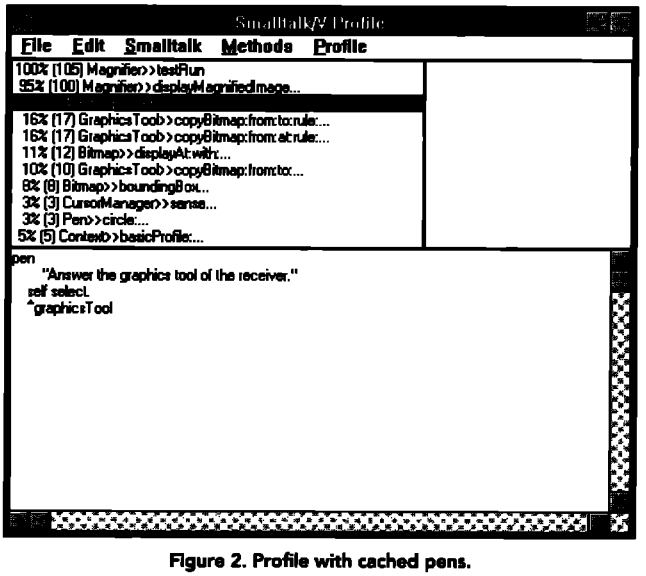


Figure 2. Profile with cached pens.

FINAL WORD

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

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

REFERENCES

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

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

BOOK REVIEW

Reviewed by Dan Lesage

OBJECT-ORIENTED MODELING AND DESIGN

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

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

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

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

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

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

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

continued on page 18 ...

WHAT THEY'RE SAYING ABOUT SMALLTALK

Excerpts from industry publications

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

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

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

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

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

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

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

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

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

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

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

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

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

... *continued from p.17*

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

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

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

PRODUCT ANNOUNCEMENTS

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

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

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

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

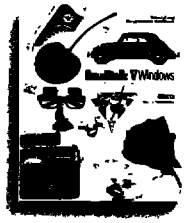
For more information, contact Vlastimil Adamovsky, 66 rue de Bourgogne, L-1272 Luxembourg; phone 352 420884.

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

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

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

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



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

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

With Smalltalk/V, you don't.

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

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

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

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

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

Smalltalk/V

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

LOOK WHO'S TALKING

HEWLETT-PACKARD

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

NCR

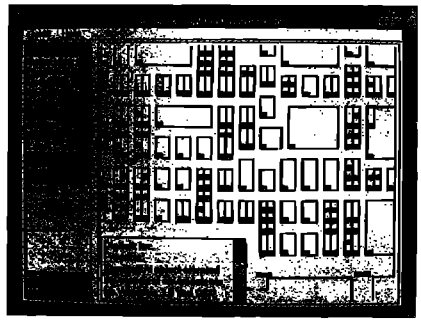
NCR has an integrated test program development environment for digital, analog and mixed mode printed circuit board testing.

MIDLAND BANK

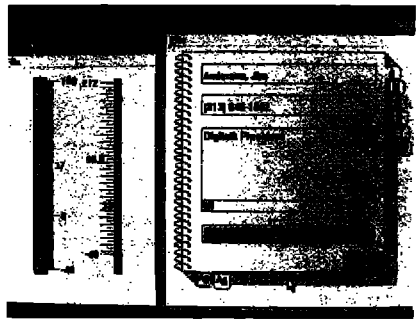
Midland Bank built a Windowed Technical Trading Environment for currency, futures and stock traders using Smalltalk V.

KEY FEATURES

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



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



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