The Smalltalk Report

The International Newsletter for Smalltalk Programmers

May 1993 **Volume 2 Number 7**

TOWARD A SMALLTALK STANDARD:

TECHNICAL ASPECTS OF THE COMMON BASE

By R.J. DeNatale & Y.P. Shan

Contents:

Recognizing Smalltalk's increasing importance as a mainstream programming language and acting as a large user of the language, IBM recently proposed the formation of a standards effort within ANSI to define a Smalltalk language standard and offered a "common base" strawman to start such an effort. At this time the proposal has been accepted by the ANSI SPARC committee, and the formation of an ANSI Smalltalk committee has begun.

This article focuses on technical issues regarding the common base. We have written a companion article that will appear in OBJECT MAGAZINE, which outlines the history of the development of the common base.

WHAT IS THE COMMON BASE?

As part of the proposal for an ANSI Smalltalk standards effort, we have contributed a "strawman" as the starting point for standardization. That strawman is contained in the IBM document entitled Smalltalk Portability: A Common Base and comprises chapters 3–5 and appendices A and B from that document.*

This proposal is not our work entirely. It is the result of an 18-month-long collaboration among five companies: IBM, Digitalk, KSC, OTI, and ParcPlace.

A purely syntactic description of Smalltalk results in a language specification that is incomplete when compared to those for languages such as C, COBOL, and FORTRAN. When studying the specification for a language one expects to learn things, such as how to do arithmetic, how to code conditional logic, and so forth. Smalltalk syntax does not address these issues. To bring the description of Smalltalk up to the expected degree of completeness we must specify a number of classes, such as numbers, booleans, blocks, and so on. The purpose of the common base is to provide a semantic description that is common to both Smalltalk-80 and Smalltalk/V. We wanted to produce a specification of Smalltalk that covers the variety of existing implementations. This led us to specifying the external behavior of classes without prescribing implementation. Detail differences between the two implementations were left out of the common base, although we have kept careful note of these differences in the review process, and they will no doubt be important items of discussion as the standardization effort proceeds.

Currently, the common base covers the following areas. (This scope might be changed during the standardization process):

- Language syntax
- Common object behavior
- Common class behavior

continued on page 4...

** The document can be ordered from your local IBM branch office or by credit card through the IBM publications ordering number (800.879.2755). The publication number is GG24-3903. The price is $2.75 per copy for printing and handling.*

John Pugh

Paul White

EDITORS' CORNER

This month's hot topic is standards. After many years of discussion, an ANSI Smalltalk language standard is now much nearer to becoming a reality. There is little doubt that languages achieve an extra measure of respectability when an ANSI standard is defined for them. Many in the Smalltalk community have long recognized this, but how do you standardize Smalltalk? The language itself is very small, but standardization alone—though valuable—does not produce a very useful result. We must standardize the class library. The Smalltalk class library can be thought of as an extension of the language; for example, even control structures are captured via message passing rather than hard-wired syntactic constructs. However, now we run into further trouble. There are two major dialects of Smalltalk: Smalltalk-80 and Smalltalk/V. Enfin might be included as yet a third dialect, and by the time you read this article there may be a fourth, SmalltalkAgents for the Macintosh. Each has classes and frameworks unique to itself particularly in the domain of user interface classes. Even when we restrict ourselves to magnitudes and collections we are not out of the woods. Smalltalk-80 and Smalltalk/V have distinct differences both in the organization of the class hierarchy and in the classes themselves. How have all these issues been addressed? Well, read the lead article written by Rick DeNatale and Y.P. Shan and you will find out. For our part, we applaud the initiative taken by IBM to promote the standards effort and the participating vendors for putting their competitive instincts to one side for the benefit of the Smalltalk community as a whole. We'll keep you informed as the standardization effort proceeds and hope that as many of you as possible will play a part in the process.

But there's even more news on standards. Digitalk has announced that it will make its Smalltalk products interoperable with SOM, IBM's System Object Model for OS/2 2.0 and that it will also develop client-server database and development tools adhering to the data access portions of Apple's Virtually Integrated Technical Architecture Lifecycle (VITAL).

In our second feature article this month, Ralph Johnson provides us with a list of classic Smalltalk bugs. He has compiled his list from the collective experiences of many experienced Smalltalk programmers. The list will be particularly useful to beginning Smalltalk programmers. If you are aware of other bugs you think should be accorded "classic" status, please forward them to Ralph. His address is given at the end of the article.

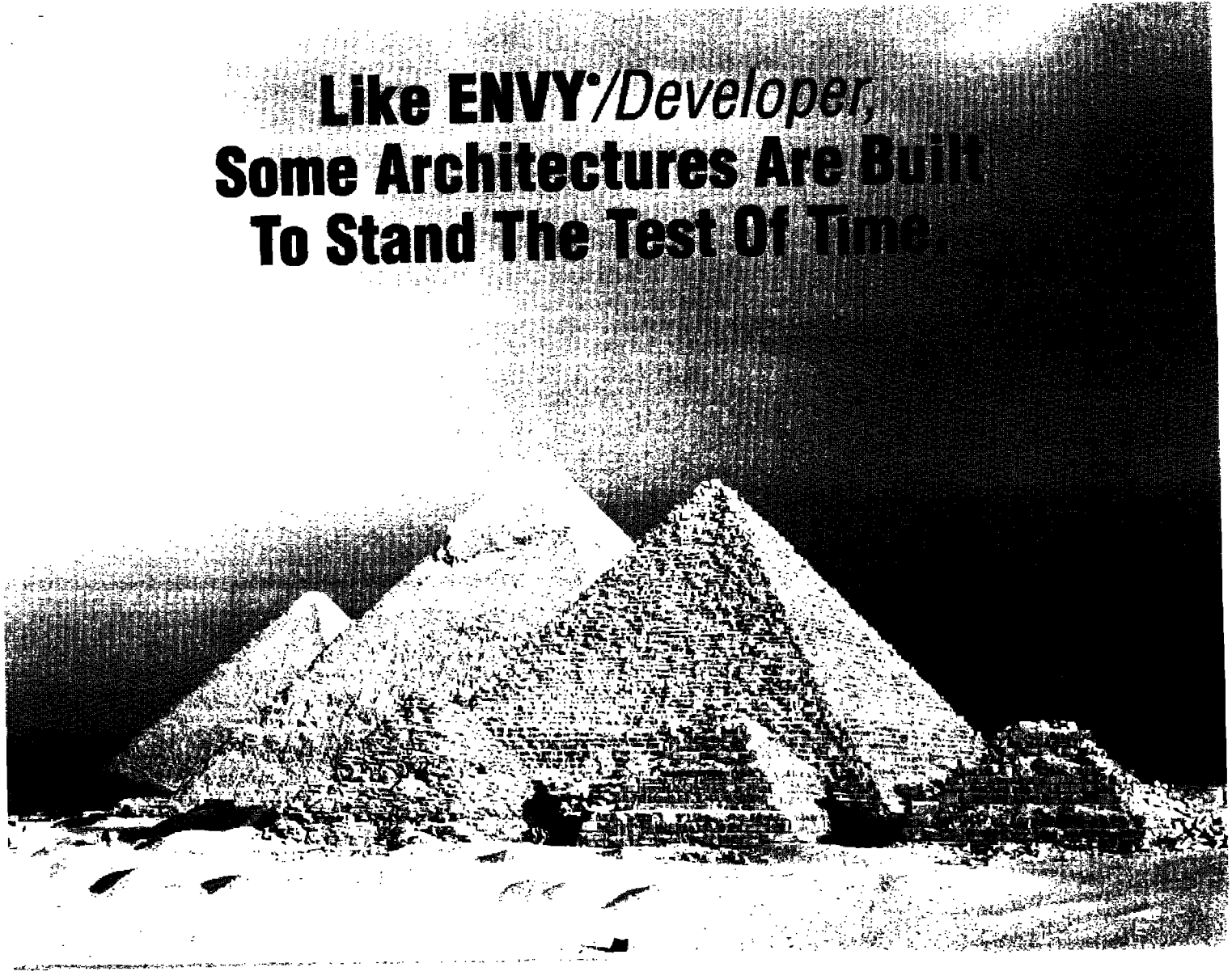
In her column, Rebecca Wirfs-Brock passes on some more of her nine years of experience designing, implementing, and managing software projects. In this issue, she discusses the incremental nature of design and what distinguishes incremental design from rapid prototyping. In this issue's Getting Real column, Juanita Ewing discusses the inappropriate use of arrays and how their misuse affects reusability. Kent Beck continues his discussion on instance-specific behavior, where methods can be attached to individual instances, as opposed to being attached only to the class. This month, Kent explores the implementation of instance specialization in Digitalk's Smalltalk/V for OS/2 and contrasts it with the ParcPlace implementation of the same concept.

Finally, Alan Knight focuses on the thread of discussion generated on comp.lang.smalltalk by the following question: "I [have] always found a way to avoid this, but I would like to know how to break away from inside a loop and return [to] the immediate upper level context?"

We hope you enjoy this issue. John Pugh P.X. 29

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by SIGS Publications Group, 588 Broadway, New York, NY 10012 (212)274-0640. Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: THE SMALLTALK REPORT, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada. For service on current subscriptions call 800.783.4903. Printed in the United States.

Like ENVY/*Developer*, Some Architectures Are Built To Stand The Test Of Time.



ENVY/*Developer*: The Proven Standard For Smalltalk Development

An Architecture You Can Build On

ENVY/*Developer* is a multi-user environment designed for serious Smalltalk development. From team programming to corporate reuse strategies, ENVY/*Developer* provides a flexible framework that can grow with you to meet the needs of tomorrow. Here are some of the features that have made ENVY/*Developer* the industry's standard Smalltalk development environment:

Allows Concurrent Developers

Multiple developers access a shared repository to concurrently develop applications. Changes and enhancements are immediately available to all members of the development team. This enables constant unit and system integration and test – removing the requirement for costly error-prone load builds.

Enables Corporate Software Reuse

ENVY/*Developer*'s object-oriented architecture actually encourages code reuse. Using this framework, the developer creates new applications by assembling existing components or by creating new components. This process can reduce development costs and time, while increasing application reliability.

Offers A Complete Version Control And Configuration Management System

ENVY/*Developer* allows an individual to version and release as much or as little of a project as required. This automatically creates a project management chain that simplifies tracking and maintaining projects. In addition, these tools also make ENVY/*Developer* ideal for multi-stream development.

Provides 'Real' Multi-Platform Development

With ENVY/*Developer*, platform-specific code can be isolated from the generic application code. As a result, application development can parallel platform-specific development, without wasted effort or code replication.

Supports Different Smalltalk Vendors

ENVY/*Developer* supports both Objectworks"\Smalltalk and Smalltalk/V". And that means you can enjoy the benefits of ENVY/*Developer* regardless of the Smalltalk you choose.

For the last 3 years, Fortune 500 customers have been using ENVY/*Developer* to deliver Smalltalk applications. For more information, call either Object Technology International or our U.S. distributor, Knowledge Systems Corporation today!

Object Technology International Inc
2670 Queensview Drive
Ottawa, Ontario K2B 8K1

Ottawa Office
Phone: (613) 820-1200
Fax: (613) 820-1202
E-mail: info@oti.on.ca

Phoenix Office
Phone: (602) 222-9519
Fax: (602) 222-8503

Knowledge Systems Corporation
114 MacKenan Drive, Suite 100
Cary, North Carolina 27511
Phone: (919) 481-4000
Fax: (919) 460-9044

...continued from page 1

- Magnitude
- Collections
- Streams
- Basic geometry
- File in/out format

THE TECHNICAL APPROACH

We wish the common base to describe the behavior of Smalltalk classes without prescribing implementation. To this end we have:

1. Documented only the public protocols of the classes
2. Avoided the specification of inheritance hierarchies

We will describe how we approached the specification of the collection classes without the prescription of a particular inheritance hierarchy.

COLLECTIONS

Collections are an important part of the Smalltalk class library, and present an interesting challenge given the desire to describe behavior without recourse to describing implementation inheritance.

A major inspiration for this work was the early publication on the internet by William Cook, currently with Apple, of his investigation of the relationship between the implementation and type hierarchies of the Smalltalk collection classes.[1] Following this work, we described each collection class individually without recourse to inheritance, in terms of combinations of the following protocols:

- **Expandable.** Contains the messages for adding elements to a collection. Set, SortedCollection, and OrderedCollection support this protocol.
- **Ordered.** Contains the messages that pertain to collections which maintain their contents in a specific order. SortedCollection, OrderedCollection, Interval, Array, and String support this protocol.
- **Copy-Replaceable.** Contains the #copyReplaceFrom:to:with: message. Interval, Array, OrderedCollection, and String support this protocol.
- **Array-Like.** Contains messages for changing the collection based on a collection or range of indices. Array, OrderedCollection, and String support this protocol.
- **Indexable.** Contains the #at: message used to access an element of the collection based on an index or key. SortedCollection, OrderedCollection, Interval, Array, String, Dictionary, and IdentityDictionary support this protocol.
- **Updatable.** Contains the #at:put: message used to replace an element of the collection based on an index or key. OrderedCollection, Array, String, Dictionary, and IdentityDictionary support this protocol.
- **Contractable.** Contains messages for removing an element or collection of elements from the collection. Set, SortedCollection, and OrderedCollection support this protocol.
- **Insertable-From-Ends.** Contains messages for adding elements at the beginning or end of the collection. OrderedCollection supports this protocol.
- **Removable-From-Ends.** Contains messages for removing elements from the beginning or end of the collection. SortedCollection and OrderedCollection support this protocol.

By specifying each collection class in terms of a set of these protocols we can describe the capabilities of each class without requiring a particular implementation hierarchy.

> “Smalltalk is more than ten years old. A standard is needed, and the time is now.”

FUTURE STANDARDS ACTIVITY

The common base represents an attempt to document what is common between the two major Smalltalk implementations. So, it leaves out what is not common. This points the way for future standards activities.

As additional implementations appear, they need to be compared to the common base. Decisions have to be made concerning what to do about existing incompatibilities. Many questions will be outside the scope of standardization, but some will need to be addressed. The impact and importance to users should be the determining factor.

The primary goal is to produce a language standard. The problem with doing this with Smalltalk is that it's not particularly clear where the language ends and class libraries take over. With the common base we made some conscious decisions:

1. We purposely avoided attempting to standardize user interface classes. The pragmatic reason is that this is where most of the differences lie between existing implementations. On the other hand, other language standards do not address user interface libraries. Smalltalk should not be penalized because it does not standardize areas not addressed by other language standards.
2. We purposely tackled higher-level language features, such as the collection classes, and some aspects of class objects, because these features make Smalltalk what it is.

Starting a standards effort inevitably triggers the desire to

continued on page 9...

CLASSIC SMALLTALK BUGS

Ralph Johnson

Every programming system is prone to certain kinds of bugs. A good programmer learns these bugs and how to avoid them. Smalltalk is no exception. Although Smalltalk eliminates many bugs that are common in other languages, such as bugs in linear search algorithms (just use do:), it has its own set of classic bugs, which most new Smalltalk programmers learn the hard way.

There are several reasons to collect classic bugs. First, it will help experienced programmers test and debug programs, and it can help us design better programs. Second, if we teach these bugs to novice Smalltalk programmers, they should learn to be good programmers faster. Third, perhaps we can redesign the system to eliminate some of these bugs, or we can write checking tools to spot them automatically.

I started the following list and posted it to comp.lang.smalltalk. Lots of people responded with more bugs, instructions on how to fix the bugs, and comments about my bugs. The result is the following list.

BUG 1: VARIABLE-SIZED CLASSES

Set, Dictionary, and OrderedCollection are variable-sized classes that grow. They grow by making a copy of themselves and "becoming" the copy. If you add new instance variables to a subclass, you have to make sure these instance variables get copied, too, or you will mysteriously lose the values of the instance variables at random points in time.

Smalltalk-80 R4.0 (and probably some earlier versions) has a #copyEmpty: method in Collection that you are supposed to override if you make a subclass of Collection that adds instance variables. The solution to this bug is to write a version of #copyEmpty: for your class.

It would be easy to write a tool that checked that every new subclass of Collection that added instance variables also defined a method for #copyEmpty:.

BUG 2: #ADD: RETURNS ITS ARGUMENT

Most collections that grow implement the #add: method, which returns its argument. Most new Smalltalk programmers assume that #add: returns its receiver, which leads to problems. Thus, they write "(c add: x) add: y" when they should really write "c add: x; add: y" or "c add: x. c add: y". This is one of the good uses for #yourself. For example, you can write:

```
(Set new
    add: x;
    add: y;
    ...;
    yourself)
```

to make sure that you have the new Set.

#add: returns its arguments for serveral good reasons. Making #add: return its argument often keeps you from resorting to temporary variables, because you can create the argument to #add: on the fly and use the argument afterward. If you want to access the collection, you can do it with #yourself and cascaded messages, as described above.

Nevertheless, after years of explaining how #add: works to students, I wish that it had been defined to return its receiver. It is too late to change now without confusing every Smalltalk programmer on the planet, so it is a problem we have to live with.

BUG 3: CHANGING COLLECTION WHILE ITERATING OVER IT

Never, never, never iterate over a collection the iteration loop modifies. Elements of the collection will be moved during the iteration, and elements might be missed or handled twice. Instead, make a copy of the collection you are iterating over. That is, aCollection copy do: [:each | aCollection remove: each] is a good program, but if you leave out the copy it isn't.

Mario Wolczko suggested a solution that catches this problem the instant it occurs (at some performance penalty, of course). The solution is to change the collection classes. Each iteration method enters that collection into a set of collections being iterated over (IteratedCollections), executes the block, and then removes the collection from the set. Collections are usually modified using #at:put: or #basicAt:put:, so these are overridden to check that the collection is not in IteratedCollections. If it is, an error is signaled. You can either use this technique all the time or just install these classes when you are testing and debugging your program. The changes are packaged in a file called Iterator-check.st that is available on the Manchester and Illinois servers. On the Illinois server, it is in pub/MANCHESTER/manchester/4.0/Iterator-check.st.

BUG 4: MODIFYING COPIES OF COLLECTIONS

It is common for an object to have an accessing method that returns a collection of objects you can modify. However, sometimes an object will return a copy of this collection to keep you from modifying it. Instead, you are probably supposed to use messages that will change the collection for you. The problem is that this is often poorly documented, and anyone who likes to modify collections directly will run into problems. See "ScheduledControllers scheduledControllers" for an example.

The solution is to provide better documentation, to claim that nobody is allowed to modify copies of collections returned from other objects, or to have objects that don't want their collections modified to return immutable versions of the collections that will give an error if you try to modify them.

BUG 5: MISSING ^

It is very easy to leave off a return caret on an expression. If there is no return at the end of a method, Smalltalk returns the receiver of the method. It only takes one missing return to mess up a long chain of method invocations.

BUG 6: CLASS INSTANCE CREATION METHODS

Writing a correct instance creation method is apparently non-trivial. The correct way to do it is to have something like:

```
new
    ^super new init
```

where each class redefines #init to initialize its instance variables. In turn, #init is defined as an instance method init:

```
super init "to initialize inherited instance variables"
"initialize variables that I define"
```

> “It only takes one missing return to mess up a long chain of method invocations.”

There are lots of ways to do this wrong. Perhaps the most common is to forget the return, that is, to write:

```
super new init
```

As a result, you have the class where you want the instance of the class. This is a special case of bug number 5.

Another error is to make an infinite loop by writing:

```
^self new init
```

If Smalltalk doesn't respond when you think it should, press ^C to get the debugger. If the debugger shows a stack of #new messages, you know you made this mistake.

Finally, you should define #new only once for each class hierarchy and let subclasses inherit the method. If you redefine it in each class, you will reinitialize the new object many times, wasting time and perhaps memory.

One way to keep this from happening is to make the #new method in Object send #init, and have the #init method in Object do nothing. Of course sometimes the version of #init that you define has arguments, and this wouldn't help those cases. It is probably better to rely on education to eliminate this kind of error.

BUG 7: ASSIGNING TO CLASSES

OrderedCollection := 2 is perfectly legal Smalltalk, but does dreadful things to your image.

This bug could be eliminated if the compiler gave a warning when you assign to a global variable that contained a class.

BUG 8: BECOME:

#become: is a very powerful operation. It is easy to destroy your image with it. Its main use is in growing collections (see bug number 1), since it can make every reference to the old version of a collection become a reference to the new, larger version. It has slightly different semantics in Smalltalk/V and Smalltalk-80, since x becomes: y causes every reference to x and y to be interchanged in Smalltalk-80, but does not change any of the references to y in Smalltalk/V.

Suppose you want to eliminate all references to an object x. Saying x becomes: nil works fine in Smalltalk/V, but will cause every reference to nil to become a reference to x in Smalltalk-80. This is a sure calamity. You want x to become a new object with no references, such as in x becomes: String new.

BUG 9: RECOMPILING BUGS IN SMALLTALK/V

It is easy to have references to obsolete objects in Smalltalk/V if you change code without cleaning things up carefully. For example, the associations whose keys are the referenced names in the Pool Dictionary are stored in the CompiledMethods at compile time. If you create a new version of the Pool Dictionary and install it by simple assignment, the compiled methods still refer to the old associations.

If you substitute a new instance of Dictionary or replace, rather than update an association in a pool dictionary, you have to recompile all methods using variables scoped to that Pool.

This is is also annoying when using ENVY, where the methods are under strict control. Perhaps Pool Dictionaries should be be first-class versioned prerequisites of classes, just like the class definition.

If you prune and graft a subtree of your class structure, you have to make sure that all referencing methods are recompiled. Otherwise, you (or your customer, because this is only detected at runtime) will run into a Deleted class error message. Thomas Muhr posted a "bite" a while ago to handle this problem for Smalltalk/V 286.

BUG 10: OPENING WINDOWS

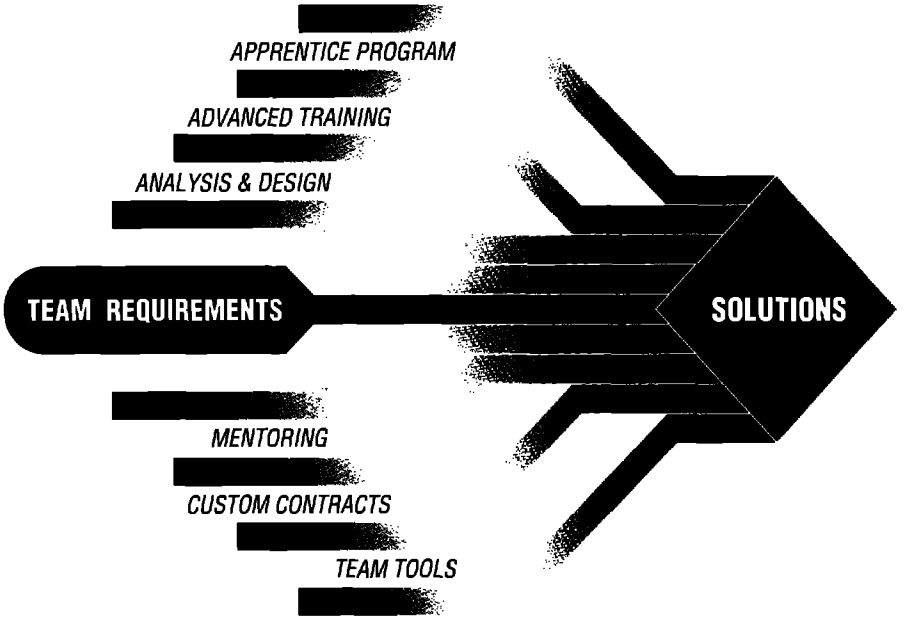
Older versions of Smalltalk/V and Smalltalk-80 do not return to the sender when a new window is opened. Thus, any code after a message to open a window will never be executed. This is the cause of much frustration. For example, if you try to open two windows at once, that is:

```
TextPane new open.
TextPane new open
```

in Smalltalk/V 286 and

```
aScheduledWindow1 open.
aScheduledWindow2 open
```

Object Transition by Design



Object Technology Potential

Object Technology can provide a company with significant benefits:

- Quality Software
- Rapid Development
- Reusable Code
- Model Business Rules

But the transition is a process that must be designed for success.

Transition Solution

Since 1985, Knowledge Systems Corporation (KSC) has helped hundreds of companies such as AMS, First Union, Hewlett-Packard, IBM, Northern Telecom, Southern California Edison and Texas Instruments to successfully transition to Object Technology.

KSC Transition Services

KSC offers a complete training curriculum and expert consulting services. Our multi-step program is designed to allow a client to ultimately attain self-sufficiency and produce deliverable solutions. KSC accelerates group learning and development. The learning curve is measured in weeks rather than months. The process includes:

- Introductory to Advanced Programming in Smalltalk
- STAP™ (Smalltalk Apprentice Program) Project Focus at KSC
- OO Analysis and Design
- Mentoring: Process Support

KSC Development Environment

KSC provides an integrated application development environment consisting of "Best of Breed" third party tools and KSC value-added software. Together KSC tools and services empower development teams to build object-oriented applications for a client-server environment.

Design your Transition

Begin *your* successful "Object Transition by Design". For more information on KSC's products and services, call us at 919-481-4000 today . Ask for a FREE copy of KSC's informative management report: *Software Assets by Design.*

Knowledge Systems Corporation

OBJECT TRANSITION BY DESIGN

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

in Smalltalk-80, then you will get one open window and one forgotten piece of code. This problem has been fixed in Objectworks\Smalltalk R 4.1 and later releases of Smalltalk/V, so the above code will create two windows as you would expect.

The fix for earlier versions of Smalltalk-80 is to use the openNoTerminate method to open the window, which does not transfer control to it. A useful trick is to store the new window in a global variable so you can test it.

Aad Nales says that the fix for Smalltalk/V286 is to fork the creation of the new window:

```
[Textpane open] fork.
```

If this is not what the programmer wants, it is probably necessary to hack the dispatcher code and remove the dropSenderChain message, which is the ultimate cause of the problem.

BUG 11: BLOCKS

Blocks are powerful, and it isn't hard for programmers to get into trouble trying to be too tricky. To compound problems, the two versions of Smalltalk have slightly different semantics for blocks, and one of them often leads to problems.

Originally blocks did not have truly local variables. The block parameters were really local variables in the enclosing method. Thus:

```
| x y |
    x := 0.
    (1 to: 100) do: [:z | x := x + z]
```

actually had three temporaries, x, y, and z. This leads to bugs such as the following:

```
someMethod

    | a b |
    a := #(4 3 2 1).
    b := SortedCollection sortBlock: [:a :b | a someOperation: b].
    b addAll: a.
    Transcript show: a.
```

When elements are added to b, the sortBlock is used to tell where to put them. What gets displayed on the transcript will be an integer, not an array.

Early versions of Smalltalk-80 (2.4 and before) implemented blocks like this, and Smalltalk/V still does. However, in current ParcPlace implementations, blocks are close to being closures. You can declare variables local to a block, and the names of the block parameters are local to the block. Most people agree that this is a much better definition of blocks than the original one. Nevertheless, people planning to use Smalltalk/V should realize that it has a different semantics for blocks.

This difference can lead to some amusing problems. For example, here is some code written by someone who had obviously learned Scheme:

```
| anotherArray aBlockArray |

aBlockArray := Array new: 4.
anotherArray := #(1 2 4 8).
1 to: 4 do: [ :anIndex |
    aBlockArray at: anIndex put: [ (anotherArray at: anIndex) * 2 ]].
```

The programmer expected each block to be stored in the array along with its own value of anIndex. If anIndex were just a local variable of the method, this will not work. It assumes that each execution of the block gets its own version of anIndex, and Smalltalk/V and old Smalltalk-80 actually make each execution share the same version.

So, if you are using Smalltalk/V, be careful not to reuse the names of arguments of blocks unless you know that the blocks are not going to have their lives overlap. Thus:

```
aCollect do: [:i | ...].
bCollect do: [:i | ...].
```

is probably OK because #do: does not store its argument, so the blocks will be garbage by the time the method is finished. However, if the first block were stored in a variable somewhere and evaluated during the execution of the second block then problems would probably occur.

BUG 12: CACHED MENUS

Menus are often defined in a class method, where they are created and stored in a class variable or a class instance variable. The method will look something like this:

```
initializeMenu
    ...
```

Note that accepting the method does *not* change the menu. You have to execute the method to change the class variable or class instance variable. Often the #initializeMenu method is invoked by the class method #initialize. This can lead to the strange effect that you can initialize the menu by deleting the class and filing it in again, but otherwise you don't seem to be able to change the menu (because you haven't figured out that you should really be executing the #initializeMenu method).

To make matters worse, it is possible that each instance of the controller, or model, or whatever has the menu, stores its own copy of the menu in an instance variable. If that is the case, it is not enough to execute #initializeMenu, you must also cause each object to reinitialize its own copy of the menu. It is often easier to delete the objects and recreate them.

Often a class will have a #flushMenus method to clear out all menus. Typically the method that fetches the menu will check to see if it is nil and invoke #initializeMenu if it is. So, #flushMenus will just "nil out" the variable holding the menu. The best way to figure out what is happening is to look at all uses of the variable. Smalltalk experts rarely have problems with this bug, but it often confuses novices.

Caching is a very common technique in Smalltalk for making programs more efficient in both time and space. Caching of menus is one of the simplest uses of caches, and other uses can create more subtle bugs.

BUG 13: SINGLETON OBJECTS IMPLEMENTED WITH CLASS METHODS

Sometimes you need to make a globally known object that is the only member of its class. These singleton objects are sometimes implemented as class methods and class variables. This works fine in the short term, but does not work in the long term because the time inevitably comes when you need to make more instances of the class. If you have implemented an object with class methods, you will have to rewrite the class or try to implement a second object by making a subclass of the first.

> Blocks are powerful, and it isn't hard for programmers to get into trouble trying to be too tricky.

The correct way to implement a globally known singleton object is to make a normal class for it, to define a class instance variable to hold the singleton object (in Smalltalk-80 this is done in the definition pane of the browser when the "meta" button is pressed), and to have a class method (I like the name #default) return the value of the variable, initializing it if it is nil. This is like a cache, and nearly eliminates the possibility of an initialization error.

Another alternative is to make a singleton object be the value of a global variable. There is no other proper use of global variables. Storing an object in a global variable is proper when there are instances of the class used for other purposes. For example, the global variable Undeclared in Smalltalk-80 is just a regular Dictionary. However, it is probably not a good way to implement a singleton class, because making sure that a global variable is initialized is a common source of problems.

CONCLUSION

I would like to thank the many people who contributed bugs or solutions to bugs to the list: Amir Bakhtiar, Hubert Baumeister, Naci Dai, Marten Feldtmann, Peter Goodall, Alan Knight, Simon Lewis, Eliot Miranda, Thomas Muhr, Aad Nales, Kurt Piersol, Jan Steinman, Mario Wolczko, Mike Smith, Terry Raymond, Dave Robbins, Randy Stafford, Michael Sullivan, Brent Sterner, Nicole Tedesco, Rik Fischer Smoody, and Markus Stumptner.

If you would like to bring bugs to my attention, please post them to comp.lang.smalltalk, email them to me at johnson@cs.uiuc.edu, or write me at Department of Computer Science, 1304 W. Springfield Ave, Urbana, IL 16801. ■

Ralph Johnson is affiliated with the University of Illinois at Urbana-Champaign.

...continued from page 4

"improve" the language. Although this desire is good, we think that the overriding goal must be to achieve a common specification that is supported by available implementations. While this is likely to require some compromise between the various Smalltalk implementors and the constituents of the user community, we believe the ultimate arbiter should be the Smalltalk user community. The users are the ultimate audience for Smalltalk and the standard.

CONCLUSION

Smalltalk is more than 10 years old. It has come a long way in overcoming the perception of being a research language and has entered the realm of commercial application development. We believe a standard is needed, and the time is now. If you agree, please encourage your organization to join us in ANSI to define the standard. Together, as Smalltalk users, we can ensure our success and contribute to the acceptance of Smalltalk by the software development community at large. ■

Acknowledgments

We would like to thank Digitalk, KSC, OTI, and ParcPlace for their contributions to and support for the project. We would also like to thank all the IBM internal reviewers, the legal and contract team, ITSC and its editors, and our management for supporting this effort.

Reference

1. Cook, W. Interfaces and specifications for the Smalltalk-80 collection classes, PROCEEDINGS OF OOPSLA '92, pp.1–15.

Rick DeNatale is a Senior Programmer with the IBM Systems Laboratory in Cary, NC. In 1993, he headed a team that designed and implemented a hybrid O-O language called ClassC. He is a co-author of the Smalltalk Common Base document. He can be reached by email at denatale@carvm3.vnet.ibm.com.

Y.P. Shan is a Development Staff Member at the IBM Systems Laboratory in Cary, NC. He has been active in researching and developing object-oriented technology since 1986. He can be reached by phone at 919.469.6571, fax at 919.469.6948, or email at shan@carvm3.vnet.ibm.com.

PUTTING IT IN PERSPECTIVE

Rebecca Wirfs-Brock

The incremental nature of design

It is good to have an end to journey towards;
but it is the journey that matters in the end.
—Ursula K. LeGuin

Design requires effort, review, reflection, and rework. I don't know of anyone who has built an application right the first time. Objects always need rework and redefinition. Solutions should remain fluid throughout an incremental design and implementation. In this column, I want to reflect on when a design starts and when it is finished. I also want to touch on some major differences between incremental design and implementation cycles and rapid prototyping.

HOW DESIGN REALLY WORKS

Designing object software means creating an executable model of interacting objects. One fundamental difference between software design and software analysis is that designs have to be translated into working programs. Analysis results need to reflect an accurate statement of the problem and constrain possible solutions, but they don't have to work. We designers still have to solve the problem. Solving even a well-defined problem is not always straightforward or easy.

I find software design to be inherently messy and fraught with mistakes. It involves top-down, bottom-up and sideways building and rebuilding of a solution. I try to teach this to my design students while giving them a strong foundation for building object designs. Designers and implementers appreciate this honest exposure to the way things really work and are eager to pick up some immediately useful skills they can apply to object design.

I've had managers sit in on design sessions (or even worse, in classrooms) and get very concerned that designers aren't honing in quickly enough on the "right" solution. Besides hindering progress, this can be demoralizing to teams new to object design. I've also worked with managers who entrust teams from the start to solve problems and produce results. Only when a schedule appeared to be in jeopardy or the team called for help did they get concerned. The enthusiasm and positive energy that sparks a team having this style of leadership are amazing! The key to these managers' success, in my opinion, was that they empowered design teams while imposing plenty of non-threatening process checks along the way.

The AMERICAN HERITAGE DICTIONARY defines design as "plan[ning] out in a systematic. . . form." I like this definition. It characterizes design as systematic planning. We're still error-prone, even if we are systematic about software design. Is that the fault of the designers, their tools, or the imprecision of inputs to the process? I don't think we should place blame on any of these factors. We software designers are inherently building complex systems. Although some researchers are actively investigating better ways to precisely state requirements while others are working at ways to minimize the transformations we make between software analysis and software design, we designers and implementers still have to deal with unpredictability. Unless we are rebuilding a system for the *n*th time, we will continue to discover additional constraints throughout implementation.

Object technology improves our chances of building well-designed systems. We have conceptual tools that help us decompose the problem. We can find objects in the problem domain that have representations in our executable programs. We can encapsulate functionality and data into objects to build high-level abstractions. Well-designed objects enable us to deal with increasing levels of complexity. Even so, we still haven't changed the bumpy, uneven nature of software development.

INJECTING DESIGN INTO IMPLEMENTATION

While software development isn't a smooth process, we still need a design process. Building systems more predictably demands that we interleave design throughout implementation. We need to consciously expend some fraction of our energy designing and refining our solution. Design needs to naturally occur throughout development. The alternative is to simply fix things so they work, or hack more functionality without considering the impact on future developers or system flexibility.

Incremental design means progressing toward a working solution in a planned fashion. One way to make orderly progress is to decompose design and implementation into a series of many small, inherently more manageable steps. I don't view incremental design as a heavily regulated or tightly monitored activity. I don't want to restrict forward progress or put a crimp on individual creativity. Designing involves an element of understanding how things work now while not accepting the status quo. Responsible designers take a broad perspective. It isn't enough to build the software; you also need to pay attention to the flexibility and elegance of the emerging solution.

Design doesn't come together at the end of a long design cycle and remain sacrosanct throughout implementation. In incremental development, systems aren't designed or integrated according to the Big Bang Theory. There are many small

cycles of discovery, design of a partial solution, analysis of the results, and rebuilding a better solution.

What distinguishes incremental design from rapid prototyping is this analytical step. *Analyze* means to "separate into parts or basic principles so as to determine the nature of the whole, to examine methodically." This is crucial to incremental design. Progress needs to be measured, reflected upon, and reviewed with others periodically. There is an openness on the part of the designer to change and improve.

Another characteristic that distinguishes incremental design from rapid prototyping is the willingness on the part of an incremental designer to throw out a bad design, rethink the problem, and redesign a solution.

The primary goal during rapid prototyping is to simply get it working. Many times an implementer during rapid prototyping knowingly (and quite possibly with some discomfort) builds something that is definitely not cleanly structured. It takes a lot of discipline to stop and clean things up with rapid prototyping.

Incremental designers, on the other hand, take many things into account throughout implementation: How can object interactions be improved? Is there a way to reduce messaging traffic between collaborators? Are interfaces to object services simple enough or powerful enough? Can a higher information bandwidth connection be made between collaborators? Is there a way to reduce the complexity of control logic? Is polymorphism being used to advantage? Is data really being encapsulated correctly? What new classes should be created to reduce existing complexity? How might behaviors be refactored to achieve a better balance and cleaner distribution of responsibilities? Have we formed the right abstractions? What classes should be eliminated? Does the current implementation of an inheritance hierarchy facilitate or unnecessarily constrict the addition of new functionality? Are there existing interactions that could be refactored to encapsulate details or hide objects from one another? How well is the object model holding up? Are there serious flaws that demand major redesign and repair?

Incremental design involves a fundamental shift in goals, values, and process. It requires that we inject incremental design throughout implementation. To do so, we must distinguish between finishing an implementation task and completing a satisfactory design. Working code doesn't automatically signal completion. Getting the design right is a journey. That journey begins as soon as the ink has dried on system requirements. It ends when we declare an end to discovery and invention. There does come a time when we have to stop improving the design and must focus on completing our work. The tricky part is picking the right time to make that dash to the finish line. Stopping design too early means the system "evolves" rather than being "systematically planned and implemented." Stopping design too late can cause problems, too. There is always a tension between getting the design "right" and meeting the schedule. However, embracing incremental design means that change and improvements aren't viewed as threats, instead they are acknowledged and carefully factored into the development process. ■

Rebecca Wirfs-Brock is the Director of Object Technology Services at Digitalk and co-author of DESIGNING OBJECT-ORIENTED SOFTWARE. *She can be reached via email at rebecca@digitalk.com or via US mail at Digitalk, 7585 S.W. Mohawk Drive, Tualatin, OR 97062. Comments, further insights, or wild speculations are welcome.*

GETTING REAL

Juanita Ewing

Don't use Arrays?

This column discusses inappropriate use of arrays and how misuse affects reusability. We will analyze several Smalltalk methods that use arrays and revise them to use classes instead of arrays. We will also show you how to search your image for methods that use arrays.

MOTIVATION

A class in Smalltalk is a specification of behavior and supporting data. Each instance contains a particular set of related data. For example, the data for an instance of Rectangle is two points. The points are related because they are both part of a rectangle: One is the origin point, and the other is the corner point.

In Smalltalk, you can also use a data structure such as an array to represent related data. Instead of the class Rectangle, you could use an array with the first element of the array being the origin point and the second element being the corner point. Which is more reusable?

First, let's examine how clients access data. Clients of the class Rectangle can send the messages origin and corner. Clients of the rectangle-as-array must access the correct element by specifying the index, and the index might not have any correlation to the values stored in the array.

Accessing the data is not the only consideration. Rectangle has specialized behavior, such as height, containsPoint:, intersects:, and expandBy:. The rectangle-as-array has no specialized behavior. For example, each client that needed the height of the rectangle-as-array would have to duplicate the code that subtracted the two y coordinates to obtain the height of the rectangle.

There are three reasons why the class is more reusable than the array:

- **Ease of Use.** Clients of the rectangle-as-array need to know arbitrary indices to obtain the data. Clients of the rectangle-as-class send messages with meaningful names.

- **Encapsulation.** The behavior of rectangle is not encapsulated with the data in the rectangle-as-array. Clients of the rectangle-as-array would need to write much more code than the clients of the rectangle-as-class in order to duplicate the behavior of rectangle. Most clients would write the same code over and over.

- **Information hiding.** The constituent data for the rectangle is accessible to all clients in the rectangle-as-array. Indeed, it must be in order for clients to the duplicate the behavior of Rectangle. But it also means the rectangle-as-array cannot change its representation without affecting all its clients.

INAPPROPRIATE USE I

Standard Smalltalk even provides us with a bad example of array usage (nobody's perfect). On page 109 of SMALLTALK-80: THE LANGUAGE AND ITS IMPLEMENTATION is the specification of a class method for Date:

Date class protocol
general inquiries

dateAndTimeNow	Answer an Array whose first element is the current date (an instance of class Date representing today's date) and whose second element is the current time (an instance of class Time representing the time right now).

Here is one possible implementation of the method:

```
Date class methods
    dateAndTimeNow
        "Answer an Array of two elements. The first element is a
        Date representing the current date and the second element
        is a Time representing the current time."
        ^ (Array new: 2)
        at: 1 put: self today;
        at: 2 put: Time now;
        yourself
```

Clients of this method must keep track of which elements are where in the array. The code to compare two date-and-time arrays looks like this (the variables now and then contain date-and-time arrays):

```
| now then oldest |
then := self oldDateAndTime.
now := Date dateAndTimeNow.
((now at: 1) >= (then at: 1) and: [(now at: 2) > (then at: 2)])
    if True: [oldest := then]
```

This kind of code is not easy to read and is likely to be duplicated in an application that manipulates time stamps.

In the dateAndTimeNow method, the array is merely a shortcut way of implementing a return of two values. The elements in the array have nothing to do with their indices. Clients have to remember which element is which. They also have to remember the algorithm for comparing date/time pairs. This kind of shortcut is not good coding practice because it does not facilitate reuse.

A better solution is to create a new class that represents an associated date and time. We will call this class TimeStamp. It would have messages for accessing its date and time, and for comparing itself with other TimeStamps. Using this new class, the dateAndTimeNow method can be rewritten:

```
Date class methods
    dateAndTimeNow
        "Answer an instance of Time Stamp containing the current
        date and the current time."
        ^TimeStamp date: self today time: Time now
```

Even better would be to eliminate the Date method and create a TimeStamp method that returns the current date and time. A TimeStamp method is better because the instance is created in the class that relates date and time. The Date class is a less desirable location because dates don't have an explicit relationship with time. Time is not referenced in other Date methods.

```
TimeStamp class methods
    now
        "Answer an instance of the receiver containing the current
        date and time."

        | current |
        current := self new.
        current date: Date today.
        current time: Time now.
        ^current
```

The client of this functionality can now write much simpler fragments of code.

```
    | now then oldest |
    then := self oldTimeStamp.
    now := TimeStamp now.
    now > then
        if True: [oldest := then].
```

INAPPROPRIATE USE II

A method from Directory provides us with another inappropriate use of an array. In this method, a collection of arrays provides detailed information about each file in a directory.

```
Directory methods
    formatted
        "Answer a collection of arrays of file information for the re-
        ceiver directory. Each array has four entries: file name, size,
        date/time and attributes."

        | answer file Entries anArray |
        file Entries := self contents.
        answer := Ordered Collection new: file Entries size.
        file Entries do: [ :each |
            anArray := Array new: 5.
            anArray
                at: 1 put: (Directory extract FileName From: each);
                at: 2 put: (Directory extract SizeFrom each);
                at: 3 put: (Directory extract DateTime From: each);
                at: 4 put: (Directory extract ResourceSize From: each);
                at: 5 put: (Directory extract CreatorTypeFrom: each).
            answer add: anArray].
        ^ answer
```

Note that the method comment is wrong. It references an array with four entries, but the code has an array with five entries, indicating that a small change in the implementation has a big impact on clients. Users of this method must know where relevant information is stored in the array. It is impossible to tell from either the comment or the code which array element is new.

In this fragment of code, the client of Directory needs the names of files of zero length. This code must reference elements stored at arbitrary locations, and requires heavy commenting to be maintainable.

```
    | zeros |
    zeros := myDirectory formatted
            select: [:info | (info at: 2) = 0]. "size is stored at 2"
    ^zeros collect: [:info | info at: 1] "name is stored at1"
```

Related data stored in arrays is more appropriate as an instance of a class. In this example, the information stored in an array

represents detailed status information about a file. An alternate solution is to create a class, called FileInformation to store this data. FileInformation has a class method to create new instances, and instance methods to access its components. A partial class specification follows:

file Information methods

fromfileEntry: afileEntry
Create and return an instance of the receiver for a file entry

file Information methods

fileName
Return the name of the file.

size
Return the size of the file, including both the data and resource fork.

timeStamp
Return the date and time when the receiver was last modified.

resourceSize
Return the size of the resource part of the file.

creatorType
Return the code that indicates the application that created the file.

With the FileInformation class, we can eliminate the use of Array and incorporate usage of our new class. The formatted method now looks like:

```
Directory methods
    formatted
        "Answer a collection of file information, one for each entry in
        the receiver."

            | answer file Entries anArray |
            file Entries := self contents.
            answer := Ordered Collection new: file Entries size.
            file Entries do:[ :each |
                answer add: (FileInformation fromFileEntry: each)].
                ^ answer
```

Clients of this method can then use meaningful selectors instead of indexing into an array. This code is more maintainable now and doesn't need any extra commenting.

```
| zeros |
zeros := myDirectory formatted select: [:info | info size = 0].
^zeros collect: [:info | info fileName]
```

There are good examples of Array use in your Smalltalk system. These are uses in which the index is a relevant part of the data structure, such as a numeric id allocated by the operating system. The array contains the relationship between the id and a related Smalltalk object. Literal arrays are convenient for collections of values.

IDENTIFYING INAPPROPRIATE USE

You can look for inappropriate use of Array and other data structures in your image. Use these techniques to find methods that reference Array. You may also want to look for references to other data structures such as OrderedCollection.

- In Team/V: Select Array in the Package Browser. Select the menu item Class/BrowseRefs.
- In Smalltalk/V for OS/2 and Smalltalk/V Windows: Execute Smalltalk senders Of: (Smalltalk associationAt:#Array)
- In Smalltalk/V Mac: Execute Smalltalk referencesTo:#Array.
- In Objectworks\Smalltalk: Select Array in the System Browser. Select the menu item Class Refs from the class pane menu.

> “Don't use arrays as a shortcut to pass around related items. Instead, create a class to represent the abstraction relating the items.”

When examining a method, inappropriate use will have one or more of the following characteristics:

- Indices that are irrelevant to data and functionality.
- Array elements that are related by some abstraction *not* captured by a class.
- Awkward client use due to violation of information hiding and encapsulation.

If you find a method that uses arrays inappropriately, you should improve the quality of your code by:

1. Creating classes to represent related array elements.
2. Rewriting offending methods to reference new classes and to eliminate arrays.

CONCLUSION

Don't use arrays as a shortcut to pass around related items. Instead, create a class to represent the abstraction relating the items. Your code will immediately be more understandable, extensible, maintainable, and reusable. Classes are the basic building blocks of Smalltalk programs. Use them. ■

Juanita Ewing is a senior staff member of Digitalk Professional Services. She has been a project leader for several commercial O-O software projects and is an expert in the design and implementation of O-O applications, frameworks, and systems. In a previous position at Tektronix Inc., she was responsible for the development of class libraries for the first commercial-quality Smalltalk-80 system.

SMALLTALK IDIOMS

Kent Beck

Instance specific behavior:
Digitalk implementation and the deeper meaning of it all

In the last issue, I wrote about what instance-specific behavior is, why you would choose to use it, and how you implement it in Smalltalk-80 . . . er . . . Objectworks\Smalltalk (which way does the slash go, anyhow?) . . . er . . . VisualWorks (is that a capital *W* or not?). This month's column offers the promised Digitalk Smalltalk/V OS/2 2.0 implementation (thanks to Mike Anderson for the behind-the-scenes info) and a brief discussion of what the implementations reveal about the two engineering organizations.

I say "brief discussion" because as I got to digging around I found many columns' worth of material there for the plucking. I'll cover only issues raised by the implementation of classes and method look-up. Future columns will contrast the styles as they apply to operating system access, user interface frameworks, and other topics.

DIGITALK IMPLEMENTATION

Runtime Structures

The Digitalk implementation of method look-up is slightly different from the ParcPlace model. Actually, until Smalltalk/V OS/2 2.0 (hereafter VOS2) the models were quite similar. The Digitalk implementation did not allow you to create Behaviors and instantiate them easily, so the instance specialization implementation presented in the last issue wouldn't work, but the pictures of the objects would have been identical.

The VOS2 model departs from the "classic" by giving each instance a reference, not to its class, but to an Array of MethodDictionaries (see Figure 1). In the normal case, the class constructs this array and all instances share it.

The ParcPlace implementation requires an additional indirection to reach the method dictionary, as the virtual machine has to go from the object to the class, and from the class to the method dictionary. With the VOS2 model, the virtual machine just has to go from the object to the array. Going up the superclass hierarchy is also faster, as the virtual machine can just march along the array rather than trace references from class to superclass.

Performance is not the primary motivation behind this design, however. More important, given the lack of flexibility in the implementation of Behavior and Class, this design makes it possible to specify the behavior of objects in many ways. For example, implementing multiple inheritance (ignoring different instance layouts in different classes) is simple. The class is welcome to create the array of method dictionaries any way it wants.

You may be wondering how the message "class" is implemented given the objects above. Each MethodDictionary has an additional instance variable called class, which is set to the class where it belongs (each class "owns" one and only one dictionary). The primitive for class marches along the array of dictionaries until it finds one whose class instance variable is non-nil, and returns that. That way, you can have dictionaries that don't belong to any class, and the scheme still works.

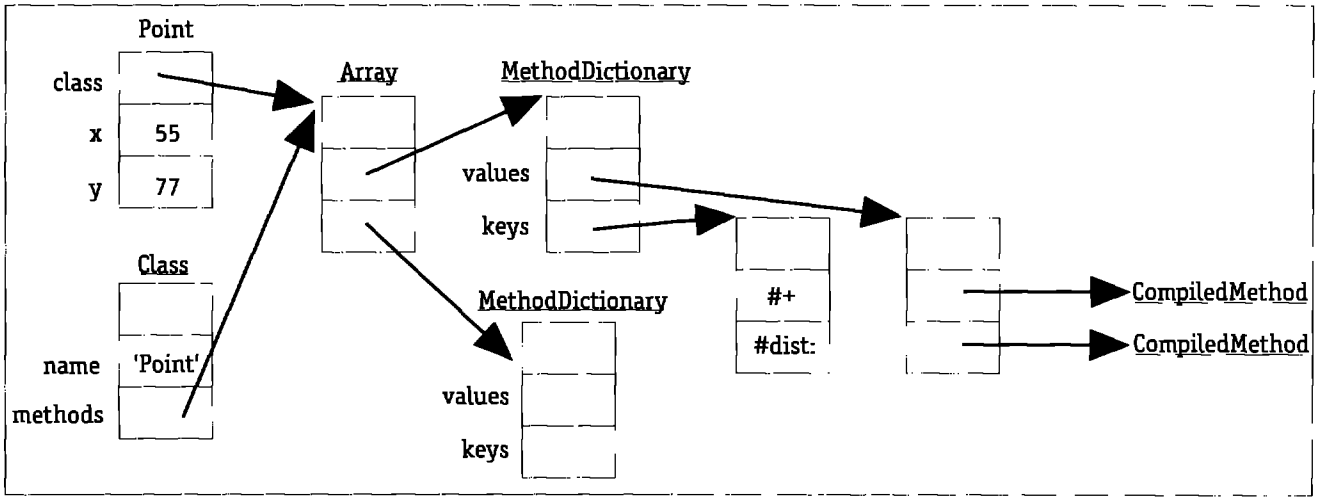


Figure 1. VOS2 objects supporting method lookup.

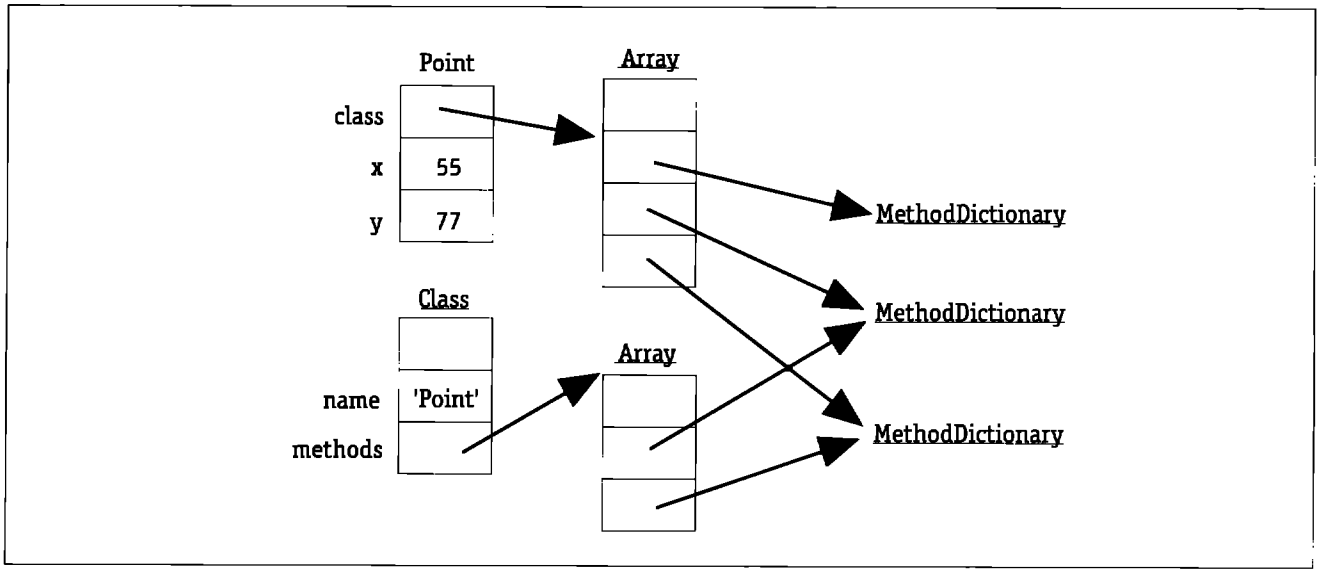


Figure 2. A specialized Point.

Conceptual Model

What's so special about the class constructing the array? It's just an Array whose elements are MethodDictionaries. Any object can build one of those. That's how we'll implement instance specialization. We'll fetch the array that's there and copy it, adding a slot at the beginning containing a fresh MethodDictionary. Then we can make all the changes we want to the private MethodDictionary without affecting any other instances.

Example

Before we can implement the conceptual model we need access to a couple of hidden primitives to get and set the method dictionaries field of the object.

```
Object>>methodDictionaryField
    "Return the Array of MethodDictionaries for the receiver"
    <primitive: 96>
    self primitiveFailed

Object>>methodDictionaryField: anArray
    "Set the Array of MethodDictionaries for the receiver
    to anArray. anArray must contain MethodDictionaries
    or your system will crash!"
    <primitive: 97>
    self primitiveFailed
```

Now we need to get something on the screen to see the effects of our experiments. Fortunately, that's easy in Smalltalk/V.

```
TopPane new open inspect
```

When we execute the above expression we get a window and an inspector on that window. In the inspector we can execute the following to get a fresh MethodDictionary to put our specialized methods in:

```
| old new |
old := self methodDictionaryField.
new := (Array with: (MethodDictionary newSize: 2)) , old
self methodDictionaryField: new
```

Now we can specialize our window by executing the following in the inspector:

```
| association |
association := Compiler
    compile: 'display Transcript show: "Howdy". super display'
    in: self class
self methodDictionaryField first add: association
```

Now if you execute self display you will see that, indeed, the specialized method is being invoked. (You will have to send the window backColor: for the superclass' display method to work).

Methods

I was surprised at how easy it was to implement instance specialization methods that were compatible with the ParcPlace version. I had expected the differences in implementation to leak through into the interface. Hmmm. . . different implementations, same interface—maybe this object stuff works, after all!

The first method I defined last time was one you would duplicate in any class in which you wanted all instances to be specializable. I don't think this is necessary, since the lazy specialization implemented below works fine. For completeness, though, here it is:

```
new
    ^super new specialize
```

The method I defined in the last issue should have been defined this way, rather than duplicating the specialization code in the class and the instance. I think I did it the way I did because that was how I saw it first implemented by Ward Cunningham when he put scripts into HotDraw.

Next is a method to test whether an instance is ready to be specialized. Since all unspecialized instances of a class share the same array of dictionaries, if the receiver has a different array we will assume it has a private array.

```
Object>>isSpecialized
    ^self methodDictionariesField == self class methodDictionaries
```

Next come the methods for actually specializing the receiver. The first sets up an array with a fresh MethodDictionary.

```
Object>>specialize
    | old new |
    self isSpecialized ifTrue: [^self].
    old := self methodDictionariesField.
    new := (Array with: (MethodDictionary newSize: 2)) , old.
    self methodDictionariesField: new
```

The next one takes a string, compiles it, and installs the result in the private dictionary:

```
Object>>specialize: aString
    | association |
    self specialize.
    association := Compiler compile: aString in: self class.
    self methodDictionariesField first add: association
```

CONTRASTS

What do these two implementations of instance specialization say about their respective systems? For one thing, both of them are simple, clean, and easy to understand. The external protocol is exactly the same. There isn't much to choose from between them. From that standpoint, I would have to say that both systems support a fairly esoteric change to the language semantics with a minimum of fuss.

The ParcPlace implementation is conceptually cleaner to me. The user's model that the behavior of an object is always defined by its class is retained. It's just a little easier to create classes than you thought. The Digitalk implementation requires that you understand the particular mechanism they have lying behind that conceptual model so that you can implement the necessary changes.

When I understood the ParcPlace implementation I said, "Ah, that makes sense." When I understood the Digitalk implementation I said, "Cool! That really works?" The ParcPlace model is an extension of the semantics. The Digitalk model is an extension of the implementation.

I am fishing for just the right way to characterize the difference. I don't think I can make it clear yet, but I also don't think it will be the work of a single week, or even a single year, to make it clear. Let's barrel on.

As you get to know both product lines, you will find this same distinction repeated many times. I think that the difference stems from the diverging goals of the technical luminaries at the two companies. The ParcPlace image was driven first by Dan Ingalls and then by Peter Deutsch. Both have strongly developed aesthetic sensibilities to go along with their amazing technical skills. A solution wasn't a solution to them until it was beautiful. Actually, now that both of them have gone on to other things, the ParcPlace models are beginning to show signs of creeping cruft.

Jim Anderson and George Bosworth, on the other hand, are primarily motivated by the belief that software just shouldn't be that hard to write. They produced Smalltalk/V so others could write software more easily. Their success criteria seems to be "if it's better than C, it's good enough." They weren't about to let a little thing like a less-than-perfect conceptual model get in the way of shipping product. Of course, they had a company to run as they were developing their image, unlike ParcPlace in the early (Xerox PARC) years, so they didn't have much choice about the importance of aesthetics.

> "Hmmm... different implementations, same interface—maybe this object stuff works, after all!"

Don't take this to mean that the ParcPlace image is truth and beauty personified and the Digitalk image is a baling-wire-and-chewing-gum collection of dire hacks. There are areas where each beats the other in both conceptual model and implementation. However, I think it is safe to say that the primary motivations behind the two systems are a contrast between aesthetics and pragmatism.

What this means for the workaday programmer isn't entirely clear. Most of the time, the ParcPlace image provides smooth development. Every once in a while, though, you will encounter a good idea that hasn't been taken quite far enough, and you will have to bend yourself into a pretzel or bypass it entirely to get around it. Put another way, if you are going the ParcPlace way you will have lots of support. If, however, you have the misfortune to want to do something a different way than the original implementor imagined, you may be in trouble. In these cases you will often have to browse around and understand lots of mechanism before you can figure out how to wedge your code in.

The Digitalk world is less coercive, but it's also less supportive. For code that relies heavily on their implementations (i.e., not just instantiating collections) I average more lines of code to get the same functionality. I know there have been cases where the Digitalk implementation has been easier. I don't think a Digitalk project has ever been conceptually simpler, though.

In future columns, I will explore more specifics of the contrast between the systems, and try to quantify why one or the other is better for specific tasks. In the meantime, if you run into situations that are surprisingly hard or easy in either sys-

continued on page 23...

THE BEST OF comp.lang.smalltalk

Alan Knight

Breaking out of a loop

This month's discussion started with a question from Deeptendu Majumder (gt0963d@prism.gatech.edu), who writes:

> **I [have] always found a way to avoid this, but I would like to know how to break away from inside a loop and return [to] the immediate-upper-level context.**

Although this question may seem elementary to an experienced Smalltalker, and the straightforward answer is probably the best, I found the wide variety of answers worthwhile and a reminder of how many different ways things can be accomplished in Smalltalk.

Unfortunately, the first answer that comes to mind is to dismiss the question.

FORGET IT

The language doesn't provide it, but it's easy to work around. Anyone who didn't just fall off the cabbage truck knows that. Next message.

This is an effective attitude for getting through news quickly, but it's not very helpful. The least we can do is describe the standard workaround.

HERE'S WHAT YOU DO INSTEAD

The obvious answer is that, although you can't break out of a block prematurely, you can break out of a method. By pushing the loop into a separate method, you can use the normal return mechanism.

For example, suppose we have a method like:

```
SomeClass>>someMethod
  self startUp.
  collection do: [:each |
        each doSomething.
        self specialExitCondition
                ifTrue: ["Break, but still do the finish up code"].
        each doSomeMore].
  self finishUp.
```

We can't break out of the loop and still do the finishUp code. To make it work, we need to break it into two methods.

```
SomeClass>>doSomething
  self startUp.
  self loop.
  self finishUp.

SomeClass>>loop
  collection do: [:each |
        each doSomething.
        self specialExitCondition ifTrue: [^self].
        each doSomeMore].
```

When specialExitCondition is true, we return from the loop method, but still execute the finishUp code. It's a simple transformation on code, and breaking the code into smaller pieces this way often improves it. Who could ask for more?

Well, perhaps it improves the code, but I doubt that it always does. While decomposing code into smaller pieces is usually good, I'd much rather do it along logical lines than along lines imposed by the language.

YOU CAN DO IT IF YOU'RE CLEVER

Saying that Smalltalk can't do something is often a mistake, particularly when you are in a virtual room with a lot of clever programmers.

Jan Steinman (steinman@hasler.ascom.ch), who is well-acquainted with the inner workings of Smalltalk, writes:

> **It is possible, but it is ugly. I had implemented it in Tek Smalltalk for "real" blocks, via a Context stack hack, but I haven't tried to make it work with 4.1 BlockClosures. It would necessarily change the semantics of blocks somewhat—what does the block answer when "broken," for instance?**
>
> **Then there's the case of in-line "pseudo-blocks." My context stack hack never did work with compiled in-line blocks, like #to:do:. This is a real problem, since the system goes out of its way to hide the difference from you!**
>
> **To make it work with pseudo-blocks might actually be easier. It would take a compiler hack that would simply jump out of the loop. But then the semantics would be different than for breaking out of a real block via a stack unwind mechanism. Yuk.**

So it can probably be done if we're sufficiently clever. This is fascinating for dedicated Smalltalk hackers and for language designers, but I don't think it's a good answer for a novice or for somebody who just wants to get things done. It would be easier to just rework the code as in the previous section. Is there a better way?

YOU CAN DO IT WITH EXCEPTIONS

An exception handling mechanism is built to handle just these sort of cases, breaking out of normal processing to handle some special condition. ParcPlace Smalltalk has one integrated with the language, and there are several implementations available for Digitalk versions.

Hubert Baumeister (hubert@mpi-sb.mpg.de) provides a detailed example of how to do this. We can define a signal handler as:

```
LoopBreakSignal := Signal genericSignal
        notifierString: 'Using break without being in a loop';
        nameClass: self message: #loopBreakSginal.
```

repeat a block using:

```
Context>>loop
        "Evaluate the receiver repeatedly, not ending unless 'Object
        loopBreakSignal' is raised or the block forces some stopping
        condition, like method returns, Signals raised but not handled
  etc.."
        Object loopBreakSignal handle: [:exp | ]
                do: [self repeat]
```

and then invoke it with the Object method

```
break
        LoopBreakSignal raise.
```

This is very similar to the use of exceptions for handling assertions, which was discussed in this column in the October 1992 issue. This is nicer, since we don't have to change any system classes, but it still has a couple of disadvantages.

First, it makes the code for looping a bit more complicated, and if we want it to be available everywhere we have to modify system methods like do:. If we want the block to return a value, we have to do even more complicated things. It probably has a fairly substantial overhead. Finally, and most important, it could lead to very confusing results.

Exception handling is a very general facility for handling non-local control transfers. It can be used to implement a facility for breaking out of a loop, but in complicated cases, the programmer needs to have the discipline to ensure that control is being transferred to the intended place.

YOU CAN DO IT WITH BLOCKS

A cleaner solution also uses the method returning mechanism, but to pass a method return as part of another block.

Ralph Johnson (johnson@cs.uiuc.edu) describes this as follows:

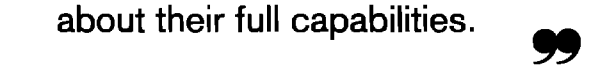
> **There are lots of ways to break out of a loop. The important thing to realize is that the only ways to change control flow in Smalltalk are to send a message and to return from a message, but blocks let you treat code as data and so control where you are going to send a message.**
>
> **The result of the above is that to simulate a go-to, you have to introduce extra blocks. For example, here is a simple way to break out of a loop:**

```
[obj foo]
        whileTrueWithBreak:
        [:exit |
                "loop body is here"
                timeToLeave ifTrue: [exit value].
                "finish up loop"]
```

whileTrueWithBreak: is defined in BlockClosure (in 2.5-4.1, BlockContext in 2.3, and I don't know where in Smalltalk/V) to be:

```
whileTrueWithBreak: aBlock
        ^aBlock value: [^nil]
```

Smalltalk blocks are most often used as simple control structures, and we usually don't have to think about their full capabilities.



Mario Wolczko also advises that the Manchester goodies library has similar code in the BlockWithExit goodie. The library is accessible by ftp@st.cs.uiuc.edu or at mushroom.cs.man.ac.uk.

This kind of code can be very confusing. Smalltalk blocks are most often used as simple control structures, and we usually don't have to think about their full capabilities. In this case, we pass as an argument a block that returns *from the method context in which it was defined.* Although there may be a great deal in the stack below that point, it is immediately discarded, and we resume execution at the next level up from that defining method.

This is quite a neat trick. It breaks out of a loop without using any additional language mechanisms, and it makes the code only a little uglier.

Unfortunately, to handle return values nicely, we have to add a bit more ugliness, adding a parameter to the exit block.

```
whileTrueWithBreakReturningAValue: aBlock
        ^aBlock value: [:returnValue | ^returnValue].
```

Writing a more complicated loop, like injectWithBreak:into: can start to get complicated. For one thing, the block will require three arguments, which is a problem in Digitalk dialects. Also, like exceptions, blocks can provide much more general transfers of control, and the programmer must ensure that the results are correct.

WHAT'S THE BEST WAY?

Considering that you can't return from a block in Smalltalk, there are a lot of different ways of doing it. Unfortunately, they all have their drawbacks. **Ralph Johnson** comments:

continued on page 23...

PRODUCT ANNOUNCEMENTS

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

GRAPHICAL CLASS LIBRARY

ObjectBits 2.0 is a sophisticated class library that permits advanced programmers to create graphical applications effectively in the ObjectWorks\Smalltalk Release 4.1 environment. Programmers can understand it quickly and use it easily because it is implemented using purely Smalltalk technologies and methodologies. ObjectBits is implemented in a modular fashion and features components such as 2-D and 3-D charts, gauges, geometric figures, and bit and image editors. ObjectBits 2.0 is available on the Sun SPARCstation, HP 9000 series 70, IBM RS/6000, and Macintosh platforms.

Fuji Xerox Information Systems, Tokyo, Japan
81.3.3378.8284 (v), 81.3.3378.7259 (f)

GUI BUILDER FOR SMALLTALK APPS

Object Technology International (OTI) and Objectshare Systems have announced a new version of WindowBuilder that is integrated with ENVY/Developer. The two companies will also cooperate to ensure that future releases of their respective products are compatible.

The new version of WindowBuilder will be available in the format of an ENVY/Developer library. Previous versions of the two products required an integration effort by the customer before they could coexist in the same Smalltalk image. Customers will now be able to load and unload WindowBuilder into their ENVY/Developer environment with no additional effort.

ENVY/Developer is an integrated multiuser environment for large-scale Smalltalk development. It provides a highly productive team programming environment that supports the prototyping, development, release, and deployment of Smalltalk applications. The product's features include configuration management, version control, support for multiplatform development, performance profiling tools, a high-speed object storage and retrieval utility, and packaging tools for producing standalone executables.

WindowBuilder is the leading Smalltalk product for building graphical user interfaces. Developers can quickly construct sophisticated user interfaces for their end-user applications. The result is less manual programming and tedious layout when developing applications with windowing front-ends. WindowBuilder is available for Digitalk's Smalltalk/V for Windows and Smalltalk/V for OS/2.

Objectshare Systems, San Jose, CA
408.727.3742 (v), 408.727.6324 (f)

BUSINESS RE-ENGINEERING METHODOLOGY

CONSTRUCT is a leading-edge business re-engineering methodology that integrates all three facets of a business—strategy, operations, and information systems, to help companies manage change. CONSTRUCT is the first methodology to enable companies to define their fundamental purpose and ensure that all work performed in the organization has a demonstrable link to that purpose. In addition, CONSTRUCT is the only methodology that incorporates Business Works, an object-oriented software tousled developed by ParcPlace Systems that enables companies to refine strategy and rapidly translate it to every element of the business.

BusinessWorks is based on ParcPlace's VisualWorks, an ADE for creating graphical, client/server applications that are completely portable across PC, Macintosh, and UNIX systems. VisualWorks' database access capabilities allow developers to combine the power of hierarchical, relational, and object-oriented database systems with object-oriented programming technology for client/server applications. VisualWorks is based on ObjectWorks\Smalltalk.

Gemini Consulting, Morristown, NJ 07960
201.285.9000 (v), 201.285.9586 (f)

AUTOMATIC DOCUMENTATION TOOL

Synopsis for Smalltalk/V provides an automatic class documentation tool for development teams using Digitalk Smalltalk/V. The automatic documentation of Smalltalk classes allows development teams to eliminate the lag between the production of code and the availability of documentation. Using information already present in the Smalltalk/V environment, Synopsis automatically generates class documentation for any class in the system. Class documentation takes the form of a summary, made up of class comments, comments about variables, and documentation strings from class and instance methods. These summaries are similar to what you find in the Encyclopedia of Classes section of any Digitalk's Smalltalk/V manual.

With Synopsis, any effort by developers to improve class or method comments in the code is immediately reflected in the net class summary generated. Therefore, documentation lag time is minimized. In addition, documentation time is reduced because a large part of the work is done once during coding.

Synopsis Software, Raleigh, NC
919.847.2221 (v), 919.847.0650 (f)

PRODUCT ANNOUNCEMENTS

OBJECT THINK

Peter Coad and Jill Nicola have just completed a new book titled Object-Oriented Programming. The book teaches "object think," the thinking stategies necessary for effective use of object technology. It also teaches how to program effectively using the two leading object-oriented programming languages: C++ and Smalltalk. The OOP book consists of four large examples: a counter, a vending machine, a sales transaction system, and a traffic flow management system. It introduces strategies and language details just at the moment each can be applied with success. According to Fotios Skouzos, IS Director at Falcon, "The OOP book has quickly become the most consulted desk reference within my development group."

The book is available from Prentice Hall technical bookstores or directly from the authors at Object International.

Object International, Austin, TX
800.662.2557 or 512.795.0202 (v), 512.795.0332 (f)

OOP WORKBENCH FOR MACS

SmalltalkAgents for Macintosh is an object-oriented software development workbench and application delivery tool with advanced computing capability.

Based on a superset of the Smalltalk language, SmalltalkAgents has extensions patterned after C and LISP, and fully supports the Macintosh toolbox including traps and callbacks. It provides a powerful new set of tools which greatly increases a programmer's productivity. SmalltalkAgents possesses advanced computing capabilities such as dynamic linking, true preemptive interrupt-driven threads and events, transparent memory management, a 24-bit international character set supporting Unicode and Worldscript, and a rich class library. SmalltalkAgents requires a Macintosh with at least a 68020 CPU, 5 MB of RAM, and a hard disk. All features are fully functional with System 7 and 7.1, with limited support for System 6.0.7.

Quasar Knowledge Systems, Bethesda, MD
301.530.4858 (v), 301.530.5712 (f)

Highlights

Excerpts from industry publications

SPECIFICALLY SMALLTALK

One of the more significant happenings this year has been the emergence of Smalltalk as an application development environment for commercial application developers. American Airlines, for example, has deployed a commercial system to manage the resources required for all flights worldwide. This high-reliability, high availability distributed system was programmed in Smalltalk and is considered a major success.

1992 was also the year that Smalltalk companies got professional management.... The other challenge facing new professional managers of Smalltalk companies is that MIS directors can be very demanding to do business with. They demand services and insist upon delivering new products on or about the published schedules. As they evaluate Smalltalk, they see a lot missing. The challenge for the next couple of years will be to rapidly add capability without losing focus. Development environment companies should build strong development environments and kernel classes for their language....

Just as Smalltalk has begun to creep into mainstream businesses, the harsh, cruel realities of using C++ as an application development language have been felt in company after company. while C++ can be used as an object-oriented language, it typically is not. Rather, it is used as a more complex C with esoteric new features that someday must be understood....

A rather startling change has been in the paychecks of highly competent O-O designers and developers. Some have doubled; a few have tripled in the last year. Companies are beginning to recognize that someone who really knows existing object-oriented libraries and tools can be worth more than five greenhorns. For this time-to-market advantage, they are willing to pay handsomely. I have seen individual Smalltalk programmers working for $2,000 per day on long-term contracts and Objective-C programmers making a salary of $200,000 per year, and this trend will accelerate."

MIS radar detects objects for the first time, Tom Love, HOTLINE ON OBJECT-ORIENTED TECHNOLOGY, February 1993

Current technologies for packaging class libraries have several problems; the most important is that they are highly lan-

■ HIGHLIGHTS (CONT'D)

guage biased. Class libraries developed in one language cannot be used with other languages. For example, a class library developed in C++ cannot be used by a Smalltalk programmer, and a Smalltalk library is of no use to a COBOL programmer. The System Object Model (SOM) is a new packaging technology designed to address this and other packaging issues. . . .

In the current version of SOM as released on OS/2 2.0, we provide full tool support for only C language bindings. . . . We also have experimental C++ bindings, designs for Smalltalk bindings, and binding to an experimental object-oriented version of REXX.

Developing with IBM's System Object Model (SOM), Roger Sessions, First Class, OMG NEWSLETTER, Feb/Mar 1993

[Mel Beckman, Duke Communications Int'l.]: One brass-tack thing you can do to improve your professional perspective is to buy Smalltalk/V for the Mac or PC and go through the tutorial. In about a week of evenings, you will pick up more insight into object-oriented programming and where new design programming is headed than you will in two or three seminars. . .

[Nick Knowles, Steam Intellect, Ltd.]: We may be hearing about C++ from IBM Toronto, but we are also hearing about Smalltalk from Rochester. Smalltalk is probably a better fit for high-level business problems. C++ may give better performance for low-level tools. . .

[Paul Conte, Picante Software]: . . . What's important is to pick a language that lets you go through the exercise of building something with objet-oriented techniques. Then you'll see that while object-oriented languages may help solve some syntactic-level problems and code-organization problems, these languages lead to another generation of problems—the creation and management of class libraries. . .

Roundtable 1992: Change and challenge, Dale Agger, NEWS 3X/400, 12/92

■ SMALLTALK IDIOMS

...continued from page 17

tem, please pass them along. You'll find my address at the end of the article.

CONCLUSION

Instance specialization has a place in the toolbox of every experienced Smalltalker. You won't use it every day—maybe not even every year—but when you want it, nothing else will do. The implementations for VisualWorks and Smalltalk/V OS/2 2.0 are quite different, but they present the same external interface to the programmer.

The contrasts between the implementations hint at fundamental differences in approach between Digitalk engineering and ParcPlace engineering. I will explore the practical consequences of this difference in future columns. ■

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPar Computer. He is also the founder of First Class Software, which develops and distributes reengineering products for Smalltalk. He can be reached at First Class Software, P.O. Box 226, Boulder Creek, CA 95006-0226, or at 408.338.4649 (phone), 408.338.3666 (fax), 70761,1216 (Compuserve).

■ THE BEST OF COMP.LANG.SMALLTALK

...continued from page 19

In general, however, I think this technique is inferior to simply restructuring your code to have an inner method that can perform the loop and that can return from the loop when needed.

In the end, I have to agree that restructuring the code is usually the best solution. The number of different possibilities available does serve, however, as a reminder of the powerful facilities available in Smalltalk. ■

ERRATA

Jon Hylands, an alert colleague who obviously reads my columns very carefully, has pointed out an error in a recent column on copying (February 1993). I had said that adding named instance variables to indexed collections in ParcPlace Smalltalk required overriding the grow method to copy these variables. In fact, the method that should be overridden is copyEmpty, which will be called by grow.

Alan Knight works for The Object People, 509-885 Meadowlands Dr., Ottawa, Ontario, K2C 3N2. He can be reached at 613.225.8812, or at knight@mrco.carleton.ca.

THE TOP NAME IN TRAINING IS ON THE BOTTOM OF THE BOX.

Where can you find the best in object-oriented training?

The same place you found the best in object-oriented products. At Digitalk, the creator of Smalltalk/V.

Whether you're launching a pilot project, modernizing legacy code, or developing a large scale application, nobody else can contribute such inside expertise. Training, design, consulting, prototyping, mentoring, custom engineering, and project planning. For Windows, OS/2 or Macintosh. Digitalk does it all.

ONE-STOP SHOPPING.

Only Digitalk offers you a complete solution. Including award-winning products, proven training and our arsenal of consulting services.

Which you can benefit from on-site, or at our training facilities in Oregon. Either way, you'll learn from a

staff that literally wrote the book on object-oriented design (the internationally respected "Designing Object Oriented Software").

We know objects and Smalltalk/V inside out, because we've been developing real-world applications for years.

The result? You'll absorb the tips, techniques and strategies that immediately boost your productivity. You'll reduce your learning curve, and you'll meet or exceed your project expectations. All in a time frame you may now think impossible.

IMMEDIATE RESULTS.

Digitalk's training gives you practical information and techniques you can put to work immediately on your project. Just ask our clients like IBM, Bank of America, Progressive Insurance, Puget Power & Light, U.S. Sprint, plus many others. And Digitalk is one of only eight companies in IBM's International Alliance for AD/Cycle—IBM's software development strategy for the 1990's. For a full description and schedule of classes, call (800) 888-6892 x411.

Let the people who put the power in Smalltalk/V, help you get the most power out of it.

100% PURE OBJECT TRAINING.

DIGITALK