The Smalltalk Report

The International Newsletter for Smalltalk Programmers

June 1993 **Volume 2 Number 8**

SMALLTALK BENCHMARKING REVISITED

By Bruce Samuelson

When Smalltalk emerged from the Xerox PARC labs in the early 1980s, performance was a major issue. CPU speeds and memory densities were both nearly two orders of magnitude lower than in today's machines. The 1983 "green book," SMALLTALK-80, BITS OF HISTORY, WORDS OF ADVICE, included many articles with detailed performance analysis.[1] One chapter even studied the feasibility of implementing Smalltalk in hardware, namely in the Intel 432 chip. The Xerox Dorado workstation was the fastest Smalltalk machine, and implementations on chips such as the DEC VAX and Motorola 68000 did well to run at a small fraction of a Dorado.

As the decade progressed, hardware got faster at a factor of nearly 10 every five years. Efficient techniques were employed for method look-up caches and for generation-scavenging garbage collectors. By mid 1992, a midrange machine running ParcPlace Smalltalk performed several times faster than a Dorado, and a fast machine a dozen times faster. One could buy a cheap PC running either ParcPlace or Digitalk Smalltalk faster than a Dorado.

These developments raise the question of whether Smalltalk is now fast enough. Shouldn't vendors concentrate on features rather than performance? Won't hardware advances take care of any lingering problems with speed? This was, in fact, the position taken by a senior representative of one of the major Smalltalk vendors in a conversation with me last year. If Smalltalk were the only language, and if there were only one vendor, the answer might be yes. But Smalltalk implementations are not only vying with one another for prominence, they are also competing with other languages.

PERFORMANCE OPTIMIZATION IN OTHER LANGUAGES

One reason C++ has become so popular is that it adds object extensions to C without sacrificing much of C's efficiency. This is a frequent theme in USENET news groups such as comp.lang.c++ and is commonly cited as a reason for using C++ instead of Smalltalk. Smalltalk users cite Smalltalk's consistent use of the object paradigm, productive development environment, rich class library, flexibility, and portability (for ParcPlace's products) as reasons to choose it over C++. A language with Smalltalk's features that approaches C++'s speed would attract a larger community of users. It this possible or only a dream?

Perhaps the researchers who are most aggressively trying to demonstrate its possibility is the Self group at Stanford University. Like Smalltalk, Self is a fully dynamically typed language. It uses prototypes and delegation in place of classes and inheritance. Whereas other researchers have tried to achieve performance gains (and perhaps other benefits) by adding strong typing to Smalltalk, the Self group is seeing how far they can push the performance envelope by using various compiler optimization techniques without sacrificing type flexibility.

They have pushed the envelope quite far. An example of their results is described in an article by Craig Chambers and David Ungar in the OOPSLA 91 con-

continued on page 16...

Contents:

John Pugh

Paul White

EDITORS' CORNER

Over the past 24 months, we have often discussed Smalltalk's move into the business world. Both Digitalk and ParcPlace have spent a significant effort to not simply improve their existing products, but instead to change their products to position Smalltalk as the best development tool for large organizations across all industries. To this end, both PARTS and VisualWorks represent the next generation of products for their respective vendors, which attempt to make Smalltalk more accessible to the mass development market—and newer Smalltalk vendors are sure to arrive. Easel's Enfin product is already having an impact on the object-oriented market that is likely to grow as time goes on.

Recently, we have noted that Smalltalk is being talked about in arenas that would not have been dreamed of before. One such place was a recent column in the April 19th issue of **Business Week** in which Smalltalk is described as being an extremely successful development tool for many corporations including American Airlines, JP Morgan, and Citicorp, and the list of these companies keeps growing. Reports such as these can be used as fodder for those of you who are still fighting to justify Smalltalk to your management.

In our feature article this month, Bruce Samuelson offers some benchmarks he has performed for the various dialects of Smalltalk. This is a new arena for THE SMALLTALK REPORT, and we believe efficiency is an issue that many of you face "in the trenches." Bruce has been very active in recent months on Internet discussing this topic, and has invested a great deal of time in preparing this study. More important than the raw numbers he presents, he has many insightful comments concerning the implementation strategies of both ParcPlace and Digitalk. While not endorsing the numbers presented by Bruce, we strongly believe these types of studies are crucial to the further mainstreaming of Smalltalk.

The debate over whether to use accessor methods has raged in the Smalltalk community since "the beginning of time." As Kent Beck points out in his column this month, this one question has probably been debated more vehemently in Smalltalk labs than any other style issue. In our own shop, the question of the appropriate use of accessors has been argued so much that it is now considered a taboo subject. We believe Kent has put this debate in the right context, especially the comment that programmers will "do anything, given enough stress," and suggest anyone responsible for the integrity of their corporate libraries give these arguments attention.

This month we have two columns that address the issue of GUI development using Smalltalk. First, Greg Hendley and Eric Smith return this month with their GUI column, getting you started with integrating VisualWorks with Microsoft's Help facility. Second, Eric Clayberg and S. Sridhar take a first look at WindowBuilder Pro, the next generation of the well-known WindowBuilder product originally released by Cooper and Peters.

Alan Knight's look at comp.lang.smalltalk takes him into a review of the implementation of sets and dictionaries. In doing so, he studies how well (or not well) abstracted the implementation of these reusable data types is and some suggestions for improving them. Also this month, Wayne Beaton describes an implementation of a mechanism for storing and managing DLLs for Smalltalk/V for windows.

Enjoy the issue!

John Pugh Paul White

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

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

Using Windows Resource DLLs from Smalltalk/V

Wayne Beaton

Microsoft provides a handy mechanism for Windows-compliant applications to store resources in Dynamic Link Libraries (DLL). While an extensive tool set exists to access resources stored in DLLs, seasoned Smalltalk programmers are a little spoiled and generally hope to avoid contact with operating system details. I have implemented a Windows resource DLL manager in Smalltalk to protect hardworking problem solvers from the semantics of dealing with Windows directly. The resources of primary interest are bitmaps, icons, and cursors; I have left room, however, for expansion to include resources such as string tables and perhaps programmer-defined resources.

A resource dynamic link library can be constructed reasonably easily—provided you have a resource compiler and a lot of time to figure out how to use it. Fortunately, Digitalk provides a resource DLL for free: the file vwsignon.dll contains the dialog that Smalltalk displays as it loads itself during runtime. A copy of this file, placed in the working directory of the image, can be easily modified by a resource editor.

All the Windows functions required to access DLLs, which are detailed in The Microsoft Windows Software Development Kit (SDK) manuals, have hooks in the base Smalltalk/V image. Also in the base image is the class DynamicLinkLibrary, which provides an abstract representation of a DLL. Equipped with this class and the battery of existing methods, all that is really required is management of the resources.

The class WindowsResourceManager has been developed to manage resource DLLs. As an instance is created, it is provided with the name of the DLL file whose resources it represents. The instance will automatically open the DLL when required and will automatically close it when the image is either saved or exited. The programmer need only ask the instance for a particular resource by type and name. The methods bitmapAt:ifAbsent:, cursorAt:ifAbsent: and iconAt:ifAbsent answer the named bitmap, cursor, or icon, respectively. The first parameter is a case-independent string containing the name of the resource; the second is a block to evaluate if the resource cannot be successfully accessed.

As each resource is loaded, it is cached to prevent the same resource from monopolizing system resources. The Windows Graphics Device Interface (GDI), for example, allocates a special handle for bitmaps. As only a relatively small number of these handles are available, frugal use will allow many bitmaps to be used frequently. Caching also relieves the programmer of the responsibility of releasing the DLL resources; all cached resources are released when a WindowsResourceManager closes itself.

The provided example methods show how an instance of WindowsResourceManager might be used. In Listing 1, an instance is created using the message

```
WindowsResourceManager class>>onDLLNamed:
```

and stored in a global variable. The instance is then asked for a bitmap with the message

```
WindowsResourceManager>>bitmapAt:ifAbsent:
```

Inspection of the method

```
WindowsResourceManager>>bitmapAt:ifAbsent:
```

reveals that the receiver is first opened. Then the cache is inspected to see if a bitmap already exists with the provided name. That failing, Windows is asked to find the bitmap. If no bitmap exists, the ifAbsent block is evaluated.

When an instance of WindowsResourceManager is asked to open, it first checks to see if it is already open. If it is not, it attempts to open the DLL it is to access and remembers it. After the DLL has opened, it tells Smalltalk to notify it on exit. The method SystemDictionary>>notifyAtExit: ensures that the instance will be notified with the message WindowsResourceManager>>exit when Smalltalk attempts to exit gracefully.

The method WindowsResourceManager>>exit simply closes the instance, releasing the resources which have been loaded, closing the DLL and removing itself from notification with the method SystemDictionary>>removeExitObject:.

When the image is saved, all classes are sent the message aboutToSaveImage. The class WindowsResourceManager reroutes this message to all of its instances. Each instance directs itself to close when the image is about to be saved. Long-term references to resources should be avoided: Accessing resources exclusively through the WindowsResourceManager will avoid embarassment when they are automatically released as the image is saved.

The code that I have included provides all the necessary equipment to effortlessly access bitmaps, cursors and icons from a DLL. As always, I am open to any suggestions as to how this may be extended, or modified for efficiency. ■

Wayne Beaton is a senior member of the Technical Staff at the Object People. He likes to think of objects as having personality as well as behavior. He can be contacted at the Object People at 613.225.8812 (v) or 613.225.5943 (f).

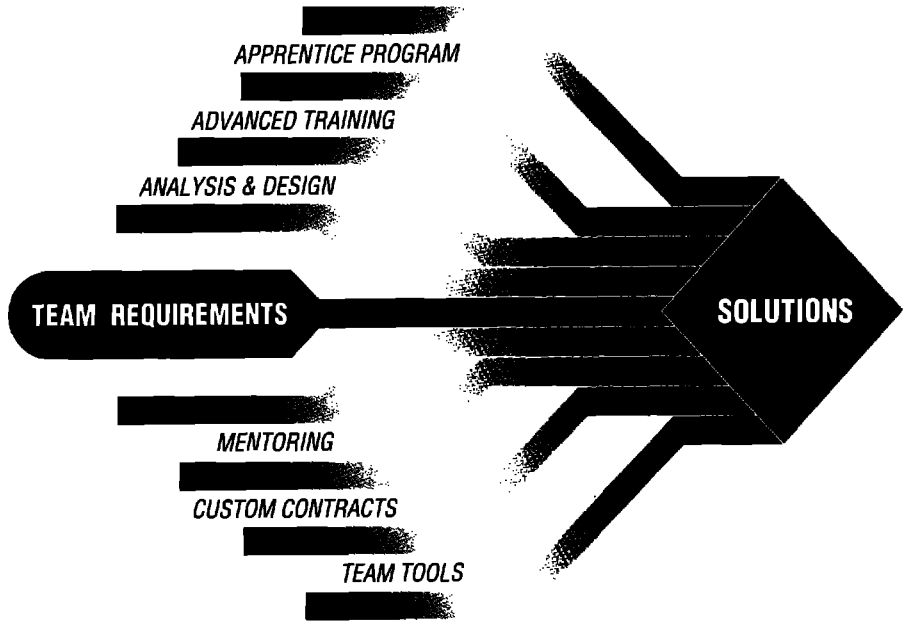
Listing 1.

```
Object subclass: #WindowsResourceManager
    instanceVariableNames:
        'fileName dll cachedResources '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'DLL'!

!WindowsResourceManager class methods

aboutToSaveImage
    "When the image is about to be saved,
    inform any of my instances."
    "(WindowsResourceManager aboutToSaveImage)"
    self allInstancesPrim do: [:each | each aboutToSaveImage]! !

!WindowsResourceManager class methods

example1
    "Answer the icon named 'Balloon' in the dll named
    'vwsignon.dll'."
    "(WindowsResourceManager example1)" | resources |
    resources := WindowsResourceManager on: 'vwsignon.dll'.
    ^resources iconAt: 'Balloon'! !

!WindowsResourceManager class methods

onDLLNamed: aString
    "Answer an instance of myself for use
    with the DLL named aString."
    ^self new fileName: aString! !

!WindowsResourceManager methods

aboutToSaveImage
    "When the image is about to be saved, close myself so that next
    time the image is opened, I open in a clean state."
    self close! !

!WindowsResourceManager methods

bitmapAt: aString
    "Answer the bitmap named aString."
    ^self bitmapAt: aString ifAbsent: [self error: 'No such bitmap.']!

bitmapAt: aString ifAbsent: block
    "Answer the bitmap named aString. If no such bitmap exists,
    then evaluate block (with no parameters)."
    | key |
    self open.
    key := Array with: Bitmap with: aString asUpperCase.
    ^self cachedResources
        at: key
        ifAbsent: [
            self cachedResources
                at: key
                put: (self buildBitmapNamed:
                    aString ifAbsent: [^block value])]!

buildBitmapNamed: aString ifAbsent: block
    "Private - Answer the bitmap named aString."
    | handle |
    handle := UserLibrary
        loadBitmap: self dll asParameter
        name: aString asParameter.

    handle = 0 ifTrue: [^block value].

    ^Bitmap fromHandle: (WinHandle fromInteger: handle)! !

!WindowsResourceManager methods

buildCursorNamed: aString ifAbsent: block
    "Private - Answer the cursor named aString."
    | handle |
    handle := UserLibrary
        loadCursor: self dll asParameter
        name: aString asParameter.

    handle = 0 ifTrue: [^block value].

    ^CursorManager fromHandle: (WinHandle fromInteger: handle)!

cursorAt: aString
    "Answer the cursor named aString."
    ^self cursorAt: aString ifAbsent: [self error: 'No such cursor.']!

cursorAt: aString ifAbsent: block
    "Answer the bitmap named aString. If no such bitmap exists,
    then evaluate block (with no parameters)."
    | key |
    self open.
    key := Array with: CursorManager with: aString asUpperCase.
    ^self cachedResources
        at: key
        ifAbsent: [
            self cachedResources
                at: key
                put: (self buildCursorNamed:
                    aString ifAbsent: [^block value])]! !

!WindowsResourceManager methods

buildIconNamed: aString ifAbsent: block
    "Private - Answer the icon named aString."
    | handle |
    handle := UserLibrary
        loadIcon: self dll asParameter
        name: aString asParameter.

    handle = 0 ifTrue: [^block value].

    ^Icon fromHandle: (WinHandle fromInteger: handle)!

iconAt: aString
    "Answer the icon named aString."
    ^self iconAt: aString ifAbsent: [self error: 'No such icon.']!
```

continued on page 6

```
iconAt: aString ifAbsent: block
    "Answer the bitmap named aString.
    If no such bitmap exists,
    then evaluate block (with no parameters)."
    | key |
    self open.
    key := Array with: Icon with: aString asUpperCase.
    ^self cachedResources
        at: key
        ifAbsent: [
            self cachedResources
                at: key
                put: (self buildIconNamed: aString ifAbsent:
                    [^block value])]! !

!WindowsResourceManager methods

cachedResources
    "Private - Answer my collection of cached resources."
    ^cachedResources!

cachedResources: aDictionary
    "Private - Set my collection of cached resources."
    cachedResources := aDictionary!

initializeCachedResources
    "Private - Initialize my resources cache."
    self cachedResources: Dictionary new!

releaseCachedResources
    "Private - Explicity release the cached
    resources to free up system resources."
    self cachedResources do: [:each |
        each release]! !

!WindowsResourceManager methods

close
    "Close myself. If I am not open then do nothing.
    Otherwise, release my cache and free my DLL.
    Set my DLL to nil so that I know I'm closed.
    Remove myself from notification at exit."
    self isOpen
        ifTrue: [
            self
                releaseCachedResources;
                initializeCachedResources.
            self dll free.
            self dll: nil.
            Smalltalk removeExitObject: self]!

exit
    "Force myself to close before exiting
    if I have not already done so."
    self close!

fileName
    "Answer my file name."
    ^fileName!

fileName: aString
    "Set my file name."
    fileName := aString!

isOpen
    "Answer whether or not I am open."
    ^self dll notNil!

open
    "Open myself with the resources in my file.
    Tell smalltalk to notify me before
    exiting (or saving the image), so that I
    can clean up.  If I am already open, then
    do nothing."
    self isOpen ifFalse: [
            self
                initializeCachedResources;
                dll: self openDLL.
                Smalltalk notifyAtExit: self]! !

!WindowsResourceManager methods

dll
    "Private - Answer the DLL which actually contains my resources."
    ^dll!

dll: aDynamicLinkLibrary
    "Private - Set the DLL which
    actually contains my resources."
    dll := aDynamicLinkLibrary!

openDLL
    "Private - Answer an instance of DynamicLinkLibrary
    opened on my file name."
    ^DynamicLinkLibrary open: self fileName! !
WindowsResourceManager comment: ''!
```

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

Kent Beck

To accessor or not to accessor?

A debate has been raging on both CompuServe and the Internet lately about the use and abuse of accessing methods for getting and setting the values of instance variables. Since this is the closest thing I've seen to a religious war in a while, I thought I'd weigh in, not with the definitive answer, but with at least a summary of the issues and arguments on both sides. As with most, uh, *discussions* generating lots of heat, the position anyone takes has more to do with attitude and experience than with objective truth.

First, a little background. The classic accessor method comes in two flavors, one for getting the value of an instance variable:

```
Point>>x
    ^x
```

and one for setting an instance variable:

```
Point>>x: aNumber
    x := aNumber
```

Accessing methods are also used to do lazy initialization, or as caches for frequently computed values:

```
View>>controller
    ^controller ifNil: [controller := self getController]
```

ACCESSORS

When I was at Tektronix, Allen Wirfs-Brock (now a Digitalk dude) wrote (or at least discussed writing—it was a while ago) a think piece called "Instance variables considered harmful." His position was that direct reference to instance variables limits inheritance by fixing storage decisions in the superclass that can't be changed in a subclass. His solution was to force all accesses to instance variables to go through a method. If you did an "inst var refs" on a variable of such a class, you'd find two users, one to return the value of the variable and one to set the value.

Points make a good example of why inheritance demands consistent use of accessing methods. Suppose you want to make a subclass of Point that obeyed the same protocols, but stored its location in polar coordinates, as r and theta. You can make such a subclass, but you will swiftly discover that you have to override most of the messages in the superclass because they make direct use of the variables x and y. This defeats the purpose of inheritance. In addition, you would have to be prepared to either declare new variables, r and theta, and waste the space for x and y in your subclass, or store r in x and theta in y and keep track of which is which. Neither is an attractive prospect.

If Point had been written with accessing methods, at least the problem with inheritance would not arise. In your subclass, you could override the messages accessing and setting x and y, replacing them with computations converting polar to Cartesian coordinates and vice versa. At the cost of four methods you would have a fully functioning PolarPoint. A more fully factored solution, one that solves the problem of wasted or misnamed storage, would be to have an abstract Point class with no variables, and subclasses CartesianPoint and PolarPoint.

ACCESSORS—*NOT!*

Many in the Smalltalk community were compelled by this argument (or arrived at the same conclusion independently). Vocal and influential organizations such as Knowledge Systems Corporation made consistent use of accessors a fundamental part of their Smalltalk teaching. Why are there still heathens who refuse to bow to this superior wisdom?

Most easily dismissed is the issue of productivity. All those accessors take too long to write. Most extended Smalltalk environments include support for automatically generating accessing and setting methods. Some are activated when the class is recompiled, asking whether you want accessors for the new methods, others appear when a "message not understood" error occurs, by noticing that the receiver has an instance variable of the same name as the offending message. In any case, writing accessors need not be time consuming.

A slightly more serious argument is performance. All those accessors take time to execute. While it is true that accessing a variable directly is faster than sending a message, the difference is not as great as you might think. Digitalk and ParcPlace are careful to make sure that looking up a method is fast, particularly in common cases like sending a message to the same class or receiver as you did the last time you were in this method. In addition, the CompiledMethod representing the accessor has special flags set to allow it to be executed quickly, without even the overhead of pushing a frame on the stack. In tight loops where the performance of accessors might still be a problem, you can probably cache the value in a temporary variable, anyway.

The crux of the objection is that accessors violate encapsulation. Accessors make details of your storage strategy visible to

continued on page 22...

Greg Hendley & Eric Smith

Using MS Help from within VisualWorks

The host "looks" that can be achieved with ParcPlace's VisualWorks can be impressive. However, once impressed, a client may ask for even more host–user interface integration. These requests can extend past the look that ParcPlace provides. The client may ask for the feel of the host system. In the case of the Microsoft Windows platforms this may include the ability to run any application without a mouse. This is an anathema for most Smalltalk programmers. Another request under windows might be integration with the help system.

Think about it. While it is not a "widget," the help system is very much part of the user interface. It is the users' way of obtaining more information on how to use an application. The rest of this column will show you how to get started integrating VisualWorks with the help system under Microsoft Windows.

Accessing Microsoft Help from VisualWorks requires knowledge of ParcPlace's Objectkit\Smalltalk C Programming (otherwise known as C Programming Object Kit or CPOK) and Microsoft Help (MS Help), each of which deserves its own column (at least). In this column, we'll explain only enough of each to get you going. The goal is for you to be able to activate MS Help from within VisualWorks and have the help document open on the topic you specify.

MS HELP

The MS Help application (MSHELP.EXE) lets you read hypertext-like help files. Help files may contain multiple topics. A *topic* is the unit of information that may be presented at one time by the MS Help application. In your application a topic may provide information on a visual part, a menu, or a window. The Microsoft Help Compiler generates help files from word processing documents saved in Rich Text Format (RTF). Refer to the Microsoft Windows Software Development Kit for more information on generating help files and defining topics.

In your Smalltalk application, you will invoke the MS Help application. Your application can simply activate MS Help, or it can specify the help file and topic that MS Help should open on. MS Help is invoked through the MS Windows API (Application Programming Interface) WinHelp().

CPOK

ParcPlace's Objectkit\Smalltalk C Programming lets Smalltalk access programs written in C. This includes the Microsoft Windows API functions. We will use it to invoke WinHelp(). CPOK is a definite improvement over writing your own primitives.

Access to C API functions is through subclasses of ExternalInterface. In general you will create a class for each API and a method for each function. The subclass creation method for ExternalInterface is different from that of most classes.

```
subclass: t
includeFiles: if
includeDirectories: id
libraryFiles: lf
libraryDirectories: ld
generateMethods: gm
beVirtual: bv
instanceVariableNames: f
classVariableNames: d
poolDictionaries: pd
category: cat
```

This method, in addition to creating a subclass, parses header files and creates methods corresponding to the functions defined in the header file. The method also creates methods corresponding to other externals of the header file.

Once ExternalInterface creates the subclass and methods, all you have to do is use them.

USING CPOK TO ACCESS MS HELP

First, we will define the class. Then we will go over how to use it.

Class definition

Create the class WindowsLibraryInterface as a subclass of ExternalLibrarySupport. If all your files are on your C: drive and your directory structure is similar to ours, your class definition will look something like this:

```
subclass: #WindowsLibraryInterface
includeFiles: '\windows.h'
includeDirectories: 'c:\windev\include'
libraryFiles: 'gdi.exe krn/386.exe user.exe'
libraryDirectories: 'c:\windev\debug'
generateMethods: '*'
beVirtual: false
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'ExternalLibrarySupport'
```

An explanation of each of the parameters can be found in the Objectkit\Smalltalk C Programming User's Guide. One parameter is worth explaining here, though. The argument gm ('*' in the above code) indicates that methods should be generated for all externals, functions, and otherwise. You could instead list just

continued on page 15...

THE BEST OF comp.lang.smalltalk

Alan Knight

Sets and dictionaries

Sets and dictionaries are widely used classes implementing well-known data types. In many ways they are exemplary, as the basic public interface is simple to use, efficient, and corresponds well to the standard abstract data types of the same name. Unfortunately, both classes can present a number of subtle difficulties. Many of these difficulties relate to the fact that both are implemented by hash tables, and that this implementation shows through more than it should.

A good abstract data type is specified without reference to its implementation, and ideally should have several possible implementations, differing only in performance characteristics. The specification should not be written to favour or depend on a particular implementation.

These goals are not always easy to live up to, and Sets and dictionaries fall short in a number of areas.

HASHING

The hashing mechanism provides an efficient search mechanism with little space overhead. It does, however, require the user to provide certain operations. These discussions refer to both dictionary keys and set elements. To save repetition, I'll refer to both as *keys*, and to both sets and dictionaries as *hash tables.*

Any hash table key must provide two methods: = and hash. A simple description of the hashing process follows. For a particular key, the hash method is used to compute an offset into the table. If that slot at that offset contains nil, the key is not present. If the slot is occupied, we test for equality with the key. If the two are equal the search has succeeded. If the two are not equal, the offset will be repeatedly incremented until an object equal to the key or a nil slot is found.

This implementation has a few implications. First, since nil is used to mark empty slots, it cannot be used as a dictionary key or inserted into a set.

Second, objects must provide = and hash methods. More importantly, they must provide these methods such that equal objects have the same hash value. Note that the converse need not hold: Objects with the same hash value do not have to be equal.

The default implementation of = is the object identity test ==, and the default hash method is compatible with this. A common mistake for Smalltalk novices is to define a different equality relation without defining a corresponding hash method. Although this is a well-known mistake, there are similar, more subtle problems.

CHANGING HASH VALUES

A hash function that is not based on object identity will probably be based on instance variables of the object. A common strategy is to add or XOR together the hash values of the significant instance variables, possibly with some additional scrambling. For example, in V/Windows:

```
Point hash
    ^x hash + y hash.
```

ParcPlace Smalltalk has

```
Point hash
    ^(x hash bitShift: 2) bitXor: y hash
```

The problem arises if any of those instance variables are changed. The hash value is then changed, and the object will hash to a different place in a set or dictionary. Any hash tables with that object as a key need to be rehashed, and there is no standard way of finding which tables those are.

This can be a very serious problem and difficult to track down. In practice, however, it doesn't seem to arise all that often. I suspect the explanation lies in the normal usage patterns. The most common dictionary keys are strings and symbols, which are not normally modified. Sets often use a greater variety of objects, but mostly use the default identity-based hash function.

IDENTITY HASHING WITH become:

Even identity-based hashes aren't completely safe, since the become: operation can change them. I've encountered an example of this with a simple version control system in V/Windows. In order to keep track of which added classes belonged to an application, the system maintained a set of classes. Classes do not override = or hash, so they inherit the identity-based version.

In Smalltalk/V Windows, there is a special class DeletedClass. When a class is deleted, the last thing the system does is:

```
classToBeDeleted become: DeletedClass.
```

This achieves two goals. It ensures that classToBeDeleted can be garbage collected, since any references to it have been re-

moved. It also ensures that any code which referenced classToBeDeleted will report an error when executed.

Unfortunately, if a class is removed outside the framework of this version control system, any applications that contained it now contain references to DeletedClass. Further, those references are stored according to the hash value of classToBeDeleted, so they can't be removed using the public set interface.

In order to remove DeletedClass, the set must be rehashed.

As a final difficulty, V/Windows does not provide a rehash operation. Fortunately, for this application, the slow-and-dirty implementation

```
aSet become: aSet copy
```

is sufficient.

HASHING PERFORMANCE

Even if your hash function doesn't play tricks on you, defining one with a good distribution can be difficult. **Jeff McAffer** (jeff@is.s.u-tokyo.jp) writes:

> **I was recently looking at a system that made extensive use of sets. . . . One of the benches was putting a whole bunch of two-element arrays into the sets. It turns out that 60% of the processing time was in the set hashing. The cause? In V/Win (likely all Vs) the hash function for arrays returns the receiver's size. I changed the hash function and doubled the speed of the benchmark.**

The identity-based hash function usually has a good distribution, but has a relatively small number of significant bits.

> "Performance will suffer greatly any time a hash table contains more elements than the hash function can handle well."

Bruce Samuelson (bruce@ling.uta.edu) writes:

> **I think the IdentityDictionary hash function runs out of steam at about 14 bits (16K objects).**

Performance will suffer greatly any time a hash table contains more elements than the hash function can handle well. To help determine if this is the case, Bruce Samuelson has also written a method to measure dictionary hash performance. It is written for ParcPlace Smalltalk, but should be easily adaptable to Digitalk dialects, and is available from either the Manchester or Illinois Smalltalk archives, under the title *dictionary-performance:*

```
!Dictionary methodsFor: 'statistics'!
hashStatistics
"This method tests how well the receiver is hashed.
It is adapted from
<Dictionary findKeyOrNil:>."
"Smalltalk hashStatistics"
    "Return an array:
    at: 1    basicSize of dictionary
    at: 2    size of dictionary, i.e., number of elements (associations)
    at: 3    average miss of hash function
        0 means hash is ideal
        N means avg element is placed N steps beyond its hash value
        large number means hash is bad
    at: 4    histogram (using a sorted collection) of misses"

    | basicSize size total histogram |
    basicSize := self basicSize.
    size := self size.
    total := 0.
    histogram := Bag new.
    self keysDo: [:key |
        | miss location probe |
        miss := 0.
        location := key hash \\ basicSize + 1.
        [(probe := self basicAt: location) isNil or: [probe key ~= key]]
            whileTrue: [
```

```
            miss := miss + 1.
            (location := location + 1) > basicSize
                ifTrue: [location := 1]].
            histogram add: miss.
            total := total + miss].
    ^Array
        with: basicSize
        with: size
        with: (total / (size max: 1)) asFloat
        with: histogram sortedElements! !
```

LARGE INSTANCES

There are other factors that might affect the performance of hash tables. For example, very large arrays of pointers (most collections, but not ByteArrays or WordArrays) can cause problems for the garbage collector. Earlier versions of ParcPlace Smalltalk included an arbitrary limit of 100,000 on the size of such collections. They've removed the limit, but the problem remains. The source of the problem is the copying garbage collectors used in Smalltalk, which can be forced to spend a lot of time copying these large objects back and forth.

> "For very small dictionaries, it may not be necessary to use a dictionary at all."

Is poor performance on very large hash tables a problem? It's certainly not the common case. **Rik Fisher Smoody** (riks@ogicse.cse.ogi.edu) writes:

> **Consider Dictionaries. The overhead of creating a small one is small. This is good. I checked one handy image: there were 540 instances of dictionary or subclasses with a total of 4,137 elements...an average of less than 10 objects/dictionary.**
>
> **But occasionally a giant arises.... What if there were a class called BigDictionary that obeys all of the external protocol of Dictionary, but is tuned for performance when it is large? Perhaps when a small (ordinary) dictionary grows, it could automatically turn into a BigDict.**

Very large hash table performance is one of those things you don't usually worry about, but when you do need it, it's very important. A BigDict would be a very handy thing to have, and I'm sure there's already more than one implementation out there. **Jan Steinman** (steinman@ascom.hasler.ch) writes:

> **To get a start on this, look at the Symbol class variable USTable, which is sort of an ordered BigSet, although it isn't implemented as a class. The general strategy is divide and conquer, as in KSAM.**

USTable (I looked at ParcPlace R4.1) seems to be a bucketed hash table with some code for choosing good dictionary and bucket sizes. The buckets are weak arrays, which stops USTable from holding on to otherwise unreferenced symbols. It may also improve speed, since weak arrays have some additional searching primitives.

Divide and conquer normally means splitting a problem up into sub-problems, each of which can be solved more easily than the whole and reassembled to form a solution to the complete problem. For a large set, the obvious decomposition is into smaller sets. By converting USTable's buckets into sets (or IdentitySets), it would be easy to convert this into a divide and conquer solution that would help avoid the performance problems of very large hash tables.

SPACE OVERHEAD

Most dictionaries are small, so the performance problems of large hash tables don't affect them. Applications that use many small dictionaries can, however, suffer from serious space problems. In particular, regular dictionaries are implemented using associations, which requires another object with two instance variables for each element in the dictionary.

IdentityDictionaries are implemented without associations in both ParcPlace and Digitalk versions. ParcPlace uses two parallel arrays of keys and values. Digitalk uses one array, storing keys at odd indices, values in even indices. Both are much more space efficient than normal dictionaries, but make operations that access associations (e.g., associationsDo:) much slower.

I'm not sure why this particular choice was made. It's nice to have more space-efficient dictionaries, but I don't see why that should be coupled to the use of identity versus equality.

For very small dictionaries, it may not be necessary to use a dictionary at all. If the number of keys is a small constant, a class using linear search may be just as efficient in time, and save even more space (this would have much less impact than regular vs. identity dictionaries). Lazy initialization can help enormously if not all objects have properties.

CONCLUSION

I've shown a few examples of problems that can arise using the hash table classes in Smalltalk. There are other tricks, such as assuming the identity of associations in a dictionary remains constant and retaining or modifying them. I think this is a bad thing, but the base Smalltalk system does it, so it's not likely to disappear soon. A broader issue is that some people believe the association-based nature of dictionaries is too public and that this imposes excessive costs on other implementations (such as IdentityDictionaries). A future column may explore these and other issues. ■

Alan Knight works for The Object People. He can be reached at 613.225.8812, or by e-mail as knight@mrco.carleton.ca.

SNEAK PREVIEW

Eric Clayberg & S. Sridhar

WindowBuilder Pro: new horizons

GUI builders have become *de riguer* in the PC desktop computing marketplace. For the past few years, WindowBuilder from Cooper and Peters has been the primary tool for building Smalltalk/V-based GUI applications. At the beginning of 1993, C&P decided to get out of the Smalltalk market. A new company, Objectshare Systems Inc. (OSI), took over the responsibility of marketing C&P's WindowBuilder line of products. WindowBuilder is the premier tool for Smalltalk/V GUI development. WindowBuilder is designed to coexist with the standard Smalltalk/V environment and, as such, generates human-readable class definitions and message interfaces. To meet the ever-increasing demands of sophisticated GUI applications, OSI is evolving the WindowBuilder product line into a professional version of the GUI builder called the WindowBuilder Pro.

As early beta testers, we'll report in this article on a number of the new features and enhancements that are an integral part of WindowBuilder Pro. Because WindowBuilder was reviewed in one of the very first issues of THE SMALLTALK REPORT, we'll skip over all its basic features.

NEW LOOK AND FEEL

WindowBuilder Pro has a nicer look and feel than WindowBuilder. Colorful toolbars abound. Across the top of the screen are buttons for creating new windows; theses include Cut, Copy, Paste, Alignment, Distribution, and Z-Order Control among others. A duplicate command that works like the corresponding command in MacDraw is a new feature. Selecting a widget or collection of widgets and hitting Duplicate creates a copy offset from the original. Moving the copy relative to the original and hitting Duplicate again results in more copies at the new offset.

WindowBuilder Pro provides increased access to the Font, Color, Framing, Menu, and other commands. Although the commands work the same way the did before, they are now accessible through a toolbar and via pop-up menus. The toolbar is right below the main editing area, and you can access the pop-up menu by clicking the right mouse button over any widget. The Framing editor has been slightly enhanced to allow users to lock objects to the horizontal and vertical centerlines of a window (as opposed to just the right, left, top, or bottom sides).

Next to the attribute toolbar are two new items that VisualBasic fans will appreciate: size and position indicators. As you move or resize widgets, these indicators constantly update to reflect the new information. This feature is very useful for precise work. A new status line at the bottom of the screen gives context-sensitive help. As you drag through menu commands or over toolbar choices, the status line describes each option. As you drag through the widget tool palettes, it describes each widget type. This is especially helpful for those whom the "intuitive" meaning of the many icons is not so intuitive. Also, as you click on any object in the editing window, the status line identifies its name and type.

In addition to the new look, WindowBuilder Pro has several nice ergonomic enhancements. You can now leave autosizing on all the time. StaticText, Buttons, CheckBoxes, and RadioButtons will automatically autosize as you type in labels. StaticText autosizes in the proper direction depending on its style. (The right-justified labels now autosize to the left! This should eliminate many of those type-autosize-move sequences). Autosizing now also conforms to the grid, rectifying an annoying oversight in the original WindowBuilder.

All widgets now include an attribute editor, and all widgets draw correctly in the edit pane (no more generic rectangles). ListBoxes and ComboBoxes feature a list editor that allows users to enter an initial list of items. Although this is not useful in cases where dynamic list data is needed, it is handy during rapid prototyping or when the items are static. DrawnButtons and StaticGraphics can now display a bitmap in the editing window. In the field for entering text for a widget, you enter the name of the bitmap file (.BMP) that you would like to use. If WindowBuilder Pro finds the file, it will display it for you. Your other option is to double-click to bring up a file dialog from which to select a bitmap. The application window's attribute editor also has been enhanced to allow the addition of minimization icons to window definitions.

RAPID PROTOTYPING

WindowBuilder Pro has four new components to facilitate rapid prototyping. These are the LinkButton, ActionButton, LinkMenu, and ActionMenu. These components provide easy ways to link windows together and perform simple actions without writing any code. LinkButtons provide a way to hook windows together without writing any code. Place a LinkButton on the screen and double-click on it to see a list of all of your ViewManager subclasses. Pick the subclass you want, then select the type of link you want. There are three types of links, *independent, child,* or *sibling.* Independent links have no logical dependency on the window that created them. Child links create windows that float

on top of their parents (great for floating toolbars), minimize with them, and close when their parents close (very much like MDI without the clipping). Sibling links create a child window of the current window's parent (e.g., your desktop window).

ActionButtons allow you to attach predefined code snippets to a button. Some of these, like "Cancel" come standard with the product. ("Cancel" performs "window close" on any window it sits on). The ActionButton attribute editor lets you select these predefined actions or create your own in standard Smalltalk. Almost any action that is not window specific could be coded once and then reused. WindowBuilder Pro needs to provide a rich variety of these predefined code snippets. The user can modify them appropriately, thus adding to the catalog of these reusable code snippets.

The LinkMenus and ActionMenus function the same way as the LinkButtons and ActionButtons. Any menu option defined with the menu editor may have a link or action associated with it. For example, you can assign the action "Cancel" to the "Exit" menu item.

WIDGET MORPHING

This is a nifty feature that will alleviate the frustration of many a WindowBuilder user. What is widget morphing? It is a feature that allows you to transform a widget from one type into any other while mapping over any common attributes.

To demonstrate how useful this is, suppose you create a ListBox, give it a name, attach a list, set its color and fonts, and give it a few event handlers. Later, let's suppose you discover that your window doesn't have room for a ListBox and you opt to use a ComboBox instead. Before the advent of this feature, you would have had to add a new control and copy all of the original control's attributes to the new control by hand (or you could change the WindowBuilder generated code by hand, which is *verboten*). Now, you can accomplish the same thing by clicking on the widget with the right mouse button and selecting the "Morph" option. This presents a cascaded list of all "similar" widget types (e.g., ComboBoxes, ListPanes and MultiSelectListBoxes in the case of ListBoxes) as well as an "Other. . ." choice (for those rare occasions when you want to transform a ListBox into a totally different widget, such as a button). Choose the one you want and your widget transforms instantly. Only events that both the old and new widget understand will be mapped over; the new widget will acquire as many of the original widget's attributes as it understands and default the rest. Be careful when morphing widgets, because while all widgets respond to #getContents, they expect very different things. It would be nice if WindowBuilder Pro added a warning message when potentially troublesome morphing is attempted.

SCRAPBOOK

One of our favorite new features is the Scrapbook. Anyone who has used the Macintosh will appreciate this one right away. (Actually anyone who has ever had to reuse visual components will appreciate this right away). The Scrapbook provides a place to store fully defined widgets or sets of widgets. It allows you to organize your creations in multiple chapters containing multiple pages. Each page contains a user-defined object.

Start by creating and defining a group of widgets. Select them all and select the Store option from the Scrapbook menu. Name your creation and select one or more chapters in which to place it. You can organize your objects under as many categories as you like. New chapters can be created with the touch of a button. There is a single special chapter entitled "Quick Reference." Anything added here is automatically appended to the "Scrapbook. . . Quick Reference" cascading menu for instant access.

In order to retrieve something from the Scrapbook, select "Retrieve." You are then presented with a listing of all of your chapters and pages. Clicking on any page will display its contents in a graphic view to the right. This allows you to preview any object before placing it on the screen. Selecting a page and hitting OK loads the cursor with the selected object which you can then drop anywhere you like.

You can easily save Scrapbooks to disk and retrieve them. Each developer can have a Scrapbook, and these can then be merged together to provide a common set of components across a development team.

CompositePanes

While the Scrapbook provides a repository for storing reusable visual components, WindowBuilder Pro's new CompositePane technology provides the mechanisms to actually create these reusable visual components. In Smalltalk, we routinely build complex classes by synthesizing structure and behavior from simpler classes. In a like manner, CompositePanes allow you to create compound or composite widgets out of other atomic widgets. WindowBuilder Pro includes an example of this in a sample CompositePane subclass called SexPane. A SexPane is composed of three widgets: two RadioButtons (Male and Female) and a GroupBox (labeled Sex). WindowBuilder Pro treats it like any other standalone widget. If you resize it, its components resize relative to itself. It even has its own instance variables and events. For example, in response to a #sexChanged event (issued whenever the user clicks one of the RadioButtons), you could bring up a MessageBox announcing the new state. Setting its contents is as simple as sending the message: aSexPane contents: #male.

OSI has seamlessly integrated this functionality with the rest of the product.

To create a CompositePane, select the appropriate option from the File menu or select several existing widgets that exist in your editing window and select the Create Composite command. This opens a new copy of WindowBuilder Pro with the selected components in it. (Here the WindowBuilder Pro itself acts as an attribute editor for the CompositePanes. Neat!). Give them names or further define them anyway you like. When you save them you are prompted for a class name and a superclass (generally CompositePane). WindowBuilder Pro creates the class and then enquires whether you would like to replace the original widgets with the new composite. Once you have a CompositePane subclass defined you may add code to it exactly the same way you would add code to a WindowBuilder

generated ViewManager subclass. You can add your own events and include them in the list of supported events.

CompositePanes can be nested within one another to any level. If you define tabbing order within your CompositePane, this nests properly as well. However, you must be careful to avoid potentially recursive definitions. WindowBuilder Pro was only able to detect single level recursion (e.g., you can't place a copy of a CompositePane within itself) but it cannot check for later recursion. If you defined A to contain B and vice versa you would be in big trouble. CompositePanes may have one of three styles: default, borders, and scroll bars. The last style is the most interesting. Placing scroll bars on a CompositePane allows you to place widgets within scrolling panes for the first time. While we wouldn't necessarily recommend doing this from a GUI point of view, its nice to know that we *can* do it.

WindowBuilder Pro provides several additional features that simplify working with CompositePanes. If you double-click on a CompositePane, it will open another copy of WindowBuilder Pro on the CompositePane definition itself. If you change the definition, it will change the CompositePane everywhere you have used it. If you decide that you don't want the CompositePane and would rather use its components directly, use the Ungroup command to split them apart losing any "composite" behavior.

OPEN ARCHITECTURE

In addition to adding lots of features for the end-user developer, OSI has also opened the WindowBuilder architecture to make it easier for third parties to build tools that integrate with the product. A new Add-in Manager allows other products to bind themselves to WindowBuilder Pro and add their own functionality and menus. Adding new widgets to the tool palettes is also easy. You must still define support for your widget the same way you would under WindowBuilder. Once you've done that, you create a tool palette bitmap for it and a simple add-in that adds your widget to the Add menu.

PLATFORMS

OSI plans to include a number of follow-on products that integrate with WindowBuilder Pro. They have already announced ENVY/Developer and TEAM/V versions of the product and they plan on having a Macintosh version that is compatible with the current Windows and OS/2 versions.

CONCLUSION

WindowBuilder has been the tool of choice for many Smalltalk/V developers for years. WindowBuilder Pro represents a logical and necessary evolution of the product that should serve the Smalltalk community well into the future. It provides significant new capabilities with its CompositePane technology and adds novel GUI building features such as the Scrapbook and Morphing utilities that should make for a pleasant GUI development environment. ■

Eric Clayberg is Director of the Computer-Human Interaction Lab at American Management Systems. He is an expert in applying O-O and Smalltalk technology to the design and construction of advanced graphical user interfaces. He can be reached on Compuserve at 72254,2515. S. Sridhar is an independent Smalltalk developer whose interests include building professional quality class libraries. He is affiliated with classAct Technology in Cary, NC. He can be reached on Compuserve at 71031,3240.

■ **GUIs** *...continued from page 9*

those externals essential for bringing up help. Unfortunately, there are dependencies in the externals defined in the windows header file. For a first pass it is easier to use '*' and create all possible methods. Warning: this may take 15 to 20 minutes.

Initiate help on a specific topic

MS Help may be opened on a help file in a number of different ways. To get you started, we will show you how to open on a particular topic. In a workspace, do:

```
WindowsInterface new
WinHelp: self GetActiveWindow
with: 'c:\my-help.hlp'
with: 20
```

where my-help.hlp is your help file and 20 is the context number for a topic in your help file. MS Help will then open on your help file and show the information for topic 20. Keeping track of which topic is which is an interesting issue that you will have to work out for yourself.

The above test code uses the method GetActiveWindow. This method answers the handle of the active window. The method was generated when you created your subclass of ExternalInterface. This is one of the side benefits of using '*' and having all methods created instead of specifying only those that look like they are necessary for help.

CLOSING

We have shown you the essential low-level Smalltalk necessary to get MS Help working with VisualWorks applications. Now you are ready to tackle the higher-level tasks of associating help topics with your windows, menus, and other visual components.

Acknowledgments

We would like to thank our coworkers Kyle Brown, who made using Objectkit\Smalltalk C Programming much easier, and John Cribbs, who applied it to accessing MS Help from Smalltalk. ■

Greg Hendley is a member of the technical staff at Knowledge Systems Corporation. His OOP experience is in various dialects of Smalltalk. Other experience includes flight simulator out-the-window visual systems. Eric Smith is a member of the technical staff at Knowledge Systems Corporation. His specialty is custom graphical user interfaces using Smalltalk (various dialects) and C. They can be contacted at Knowledge Systems Corporation, 114 MacKenan Drive, Cary, North Carolina 27511, or by phone at 919.481.4000.

...continued from page 1

Table 1. Language execution speed as percentage of optimized C.

Language	Stanford Suite	Puzzle	Richards
ST80 2.4	10%	4.4%	9.4%
Self 91	57%	27%	35%

ference proceedings.[2] They compared the execution speeds on a Sun SPARC workstation of C, Smalltalk-80 2.4, and Self 91 (and some other languages) on the Stanford Suite of integer benchmarks, the Puzzle benchmark, and the Richards operating system simulation benchmark.

The results, given in Table 1, are impressive, especially since Self is at least as hard to optimize as Smalltalk, and the same techniques used to tune it can be applied to Smalltalk. Self actually did better than the numbers indicate. Relative to Smalltalk-80, its optimization doesn't freeze the definition of low-level looping constructs. And it supports several features not found in optimized C: "generic arithmetic, robust error-checking primitives, and support for source-level debugging." In demonstrating an object-oriented environment that is both efficient and full-featured, the authors claim that "programmers no longer need to choose between semantics and performance."

Smalltalk did better on the Richards benchmark in an independent test posted to comp.lang.smalltalk in mid-1991. While the Self group measured Smalltalk-80 2.4 to run at 9.4% of optimized C++ on a Sun 4/260 running UNIX, the poster measured Smalltalk-80 4.0 to run at 27% relative to Borland C++ 2.0 on a 386SX running DOS/Windows. He suggested that the difference could be due to using ST80 version 4.0 rather than 2.4.

Just how practical would it be for the commercial Smalltalk vendors to get its speed up to that of Self? Self's optimizations exact several penalties. First, whereas Smalltalk compiles individual methods incrementally at an almost instantaneous speed, the optimizations performed by Self's compiler slow it down to a compilation speed comparable to C. Second, compilation of "uncommon cases" is deferred, but when it does happen during runtime, it may be somewhat intrusive. Third, Self's code takes between one-third to four times more space than the C code generated for the benchmarks. I haven't seen data on Smalltalk code density, but I would expect it to take less space than C. I'm referring to incremental code density, not to the initial size of the class library. In Self's favor, it might look better when compared to heavily object-oriented programs written in C++ than to procedural programs written in C because of the space C++ uses for virtual function dispatch tables. Fourth, the Self environment requires more space than Smalltalk. The people I've talked with at ParcPlace have the impression that on a 32MB machine, Self pages unacceptably, and that you need at least 64MB to run it comfortably. When I raised this point with a Self researcher, he made two rebuttals. First, Self has not been optimized for space. He cited examples of major savings that could be made. Second, he said that Self runs fine on a 32MB machine and does not require 64MB. He did concede that it still takes more space than Smalltalk, but this is at least partly because it is a research language and the focus of the research has not been to minimize memory requirements.

I have talked with several representatives of both ParcPlace and the Self team in the last couple of years about these issues. My impression is that they're not communicating enough. I think that some of ParcPlace's misgivings about Self could be confirmed or denied by talking more with the Self people. I don't know to what extent Digitalk is in touch with Self. If the commercial vendors decide to focus on performance tuning as hard as the researchers, the communication channels will no doubt open wider!

We'll return later to the question of whether Self's optimizations can be applied to Smalltalk.

Table 2. Looping benchmark results.

Tester	Vendor	Version	Word length, bits*	while loop, milliseconds (avg result) original	modified	fibonacci, milliseconds (avg result)
Nouwen	Digitalk	ST/V-Win 2.0	16	3,570	n/a	32,960
Feldtmann	Digitalk	ST/V-PM 1.4	16	2,530	n/a	9,503
Nouwen	Digitalk	ST/V-PM 1.4	16	2,530	n/a	9,470
Feldtmann	Digitalk	ST/V-PM 2.0	32	125	n/a	4,673
Nouwen	Digitalk	ST/V-PM 2.0	32	120	n/a	4,650
Samuelson	ParcPlace	VW-Win 1.0†	32	353	92‡	6909

* *These are my assumptions about the underlying word length in the virtual machines. Although VW-Win runs on a 16-bit operating system (Win 3.1), it is a 32-bit implementation because it is compiled with a 32-bit DOS extender.*

† *This is ParcPlace's new VisualWorks product, which is ST80 with an interface builder and other extras bundled.*

‡ *Notice the dramatic speed-up for ParcPlace when the declaration of the temporary index variable is moved inside the outer block. ParcPlace distinguishes between clean blocks, copying blocks, and full blocks. These vary, respectively, from fastest to slowest and from least context overhead to most overhead. In moving the variable declaration, the outer block goes from a full block to a clean block and the loop runs four times faster. This confirms ParcPlace's admonition to use clean blocks whenever possible. I don't think Digitalk makes these distinctions, at least for its DOS version.*

HOW IMPORTANT WOULD A FAST SMALLTALK BE TO USERS?

The jury is out on whether Self's optimizations will find their way into Smalltalk and other commercial languages. In the meantime, it would help your vendor to know how much priority you give to performance. I took a survey of ParcPlace customers in the comp.lang.smalltalk newsgroup in October 1991 and asked them to rank the importance of 19 features. Faster execution speed came in third place, with first place going to maintaining cross-plat-

form portability and second to supporting true native look and feel. Here are the comments I received on performance:

- It would certainly be nice if it ran faster, but I think resources might be better devoted elsewhere. [Speed] might help attract potential new customers, though.
- Speed is very important (that simple).
- Speed is the standard problem with Smalltalk.
- My first major program in Smalltalk (a simulation) still doesn't run fast enough to be useful. Definitely give me more speed.
- Faster execution speed will have a large effect on the use of Smalltalk in industry. Although Smalltalk would be fast enough for their applications, C is often used instead "just in case."
- Execution speed will always be important and [it will] never be [fast] enough, so it needs constant attention.
- The biggest negative perception Smalltalk has from the general computing community is that it is too slow. Unless this perception is corrected, Smalltalk will remain a "cult language." My particular project is a large-scale Smalltalk effort, and I am anticipating execution speed to be a major problem.
- We do some heavy computation using it.

I heard a contrary opinion recently. At the February meeting of the North Texas Society for Object Technology, a speaker from Texas Instruments described a chip fabrication software system they developed using ParcPlace Smalltalk, Gemstone, The Analyst, Envy, and other third-party products. This is a big system with over 3,000 classes. The speaker said that in no case did they encounter a performance bottleneck that was Smalltalk's fault. The problems they did have were due to misapplying the technology. So there are some major users who do not consider performance to be a problem.

COMPARING PARCPLACE SMALLTALK TO DIGITALK SMALLTALK: FIRST TRY

There is considerable data in the literature measuring Smalltalk-80's performance. The green book mentioned previously covers early, experimental implementations. ParcPlace's newsletter publishes Dorado benchmarks for current commercial versions. And the Self group has compared ST80 2.4 to Self 91 and to C.

I haven't seen literature comparing the performance of ParcPlace's Smalltalk-80 with Digitalk's Smalltalk/V. The current article is a modest, if flawed, step in this direction.

People often claim that ST80 is faster than ST/V. Is this true? Recent articles in comp.lang.smalltalk bring this into question. Someone published two very simple benchmarks for the OS/2 versions of ST/V, and others published results for ST/V-Windows and ST80-Windows. The results were surprising, because the 32-bit version of ST/V for OS/2 was, at first glance, between 1.5 and 3 times faster than the 32-bit version of ST80 for Windows. The 16-bit versions of ST/V fared much worse, probably because both benchmarks generated numbers that would be LargePositiveIntegers for 16-bit Smalltalk. Later, I discovered that by slightly modifying one of the benchmarks, ParcPlace moves from being 3 times slower to one third faster than the fastest Digitalk version. It remains 1.5 times slower in the other benchmark. The code as posted follows, and the results are given in Table 2:

1. while loop (original posting)

```
| anIndex |
Time millisecondsToRun: [
    anIndex := 100000.
        [anIndex 0] whileTrue:[ anIndex := anIndex - 1]]
```

2. while loop (modified for ST80 by declaring anIndex as a temporary block variable)

```
Time millisecondsToRun: [
    | anIndex |
    anIndex := 100000.
        [anIndex 0] whileTrue:[ anIndex := anIndex - 1]]
```

3. Fibonacci number generator (tested with "30 fib")

```
fib (in class integer)

    self 1
        ifTrue:[ ^((self - 1) fib + (self - 2) fib ) ]
        ifFalse:[ ^1]
```

Hardware 486/33 (Feldtmann, Samuelson 16MB; Nouwen 8MB).

COMPARING PARCPLACE SMALLTALK TO DIGITALK SMALLTALK: SECOND TRY

As a ParcPlace customer, I was intrigued and startled enough by these results that I decided to measure how the Digitalk and ParcPlace products perform on a wider range of tests. The goals of the benchmarks I developed are:

- Portability between versions of ST80 and ST/V, including ST/V-DOS.
- Writing in as idiomatic a style as portability would allow.
- Being able to compile and run within ST/V-DOS's 640K limit.
- Keeping integers small enough to not skew the results against 16-bit versions.
- Running for a long enough time to get fairly accurate results.
- Being cpu intensive while avoiding accesses to disk or video subsystems.
- Avoiding disk paging.
- Measuring both low-level and medium-level operations.

These goals were to some extent mutually exclusive. For example, it is hard to keep integers and loop counts within the bounds of 16-bit integers while still consuming measurable amounts of time. And it is hard to consume enough time without exceeding runtime resource limits of ST/V-DOS. It took experimentation and dozens of reboots of my machine during ST/V-DOS runs before I arrived at something that met all the goals. The resulting benchmarks are called *slopstones* (Smalltalk Low-level OPeration

Table 3. Slopstone and smopstone results.

Vendor	Version	GUI	OS	Brand	CPU	flPt	MHz	Extrn cache, KB	RAM MB	Slopstone (low),*	Smopstone (med)*
PPS	VW 1.0	OpW3.0	SunOS 4.1.3	Sun SS/10-30	SPARC	int	36	0+	32	.905	1.932+
PPS	VW 1.0	—	HP/UX 2.7	HP 720	PA	intrn	50?	—	32	1.498	1.673
PPS	VW 1.0	Win3.1	DOS 5.0	Amax none	486DX	intrn	33	256	16	1.0	1.0
PPS	80 4.0—	—	Sun	SS/2	SPARC	—	40	64	64	1.137	0.995
Dig	V 2.0	PM	OS/2 2.01b	clone	486DX	intrn	33	256	16	0.411	0.982‡‡
PPS	80 4.0	Win3.1	DOS 5.0	Amax none	486DX	intrn	33	256	16	0.995	0.973
Dig	V 2.0	PM	OS/22 2.01b	clone	486DX	intrn	33	256	16	0.411	0.71
PPS	VW 1.0	Mac	MacOS 7.01	MacQuadra700	68040	intrn	25	0?	20	0.525	0.572
Dig	V 1.4	PM	OS/2 1.4	clone	486DX	intrn	33	256	16	0.236	0.470
Dig	V 1.2	Mac	MacOS	Mac accel***	68040	intrn	25	—	—	0.137	0.344‡‡
Dig	V 2.0c	—	DOS 5.0	Amax none	486DX	intrn	33	256	16	0.070	0.261
Dig	V 2.0	WinOS2	OS/2 2.01b	clone	486DX	intrn	33	256	16 0	167	0.25
Dig	V 1.2	Mac	MacOS 7.0.1	Mac II ci	68030	68882	25	32?	16	0.078	0.191‡‡
Dig	V	Mac	MacOS 7.01	Mac II ci	68030	68882	33?	0?	5/6	0.072	0.184‡‡
PPS	VW 1.0	Mac	MacOS 7.01	Mac IIci	68030	68882	25	0?	16	0.174	0.180
Dig	V 1.2	Mac	MacOS	Mac accel***	68040	intrn	25	—	—	0.137	0.131
PPS	80 4.0	OpW2.0	SunOS 4.1	Sun 3/50	68020	68881	16	0	12	0.114	0.107
PPS	80 2.5	SunVw	SunOS 4.1	Sun 3/50	68020	68881	16	0	12	0.067	0.102**
Dig	V286 1.2	none	DOS 5.0	OPTI	386DX	none	25	0	4	none	0.096
Dig	V 1.2	Mac	MacOS 7.0.1	Mac IIci	68030	68882	25	32?	16	0.078	0.072
Dig	V	Mac	MacOS 7.01	Mac IIci	68030	68882	33?	0?	5/6††	0.072	0.069
Dig	V	Mac	MacOS 7.01	Mac PB100	68000	none	16	0	3/4††	0.020	0.051‡‡
Dig	V	Mac	MacOS 7.01	Mac PB100	68000	none	16	0	3/4††	0.020	0.019
Dig	V 2.0	none	DOS 3.3	clone XT	8088	none‡	5	0	640K	0.002‡	0.008‡
Dig	V	Mac	MacOS 7.01	Mac IIsi	68030	none	25	0?	3/4††	0.044	none

* *Results are normalized to one for VisualWorks 1.0 on my 486/33.*

† *SS/10-30 has 36K internal cache. 80486 and 68040 (and I think SS/2) have 8K.*

‡ *Floating point performance was extrapolated assuming an 8087.*

** *Smopstones didn't include set formation benchmark—string hash inadequate.*

†† *x/y means Mac allocated x MB to Smalltalk out of y MB total.*

‡‡ *Smopstones excluding the two worst cases (stream, set) and the best case (sorting). The stream and set results were bad for ST/V Intel and atrocious for ST/V Mac, probably because of weak implementations of mixed integer and float arithmetic (used in streams) and string hash (used in forming sets).*

*** *This machine was a Mac IIci with a 25 MHz 68040 Radius Rocket accelerator.*

Note: The entries are sorted by Smopstones (last column). Higher numbers in the last two columns mean greater speed. Eight people contributed these results. For the 486, I did the PPS runs and ST/V-DOS run. Marten Feldtmann did the remaining ST/V 486 runs.

Stones) and *smopstones* (Smalltalk Medium-level OPeration Stones), and the results are summarized in Table 3.

I wanted to avoid any tests that stressed the disk or video systems. Although these are important in real applications, modern caching disk controllers and video coprocessors make it hard to make objective cross-platform comparisons. Also, portability is difficult to achieve between ST80 and ST/V in video tests.

SLOPSTONES

The seven low-level tests are:

- Adding integers
- Adding floats
- Accessing a character in a string
- Creating an object
- Copying an object
- Performing a unary selector
- Evaluating a block without arguments

Each test is repeated many times inside a block. For example, integer addition looks like [1+1+1+1... many times]. The block, in turn, is evaluated many times.

Table 4. Benchmark results of 32-bit implementation.

slopstones (low-level)	benchmark (med level)	smopstones	benchmark
1.09	add integers	1.46	generate fractonaccis
0.53	add floats	1.09	generate primes
0.56	access strings	0.14	generate and parse streams
0.62	create objects	0.68	generate strings
0.45	copy objects	0.30	form sets
0.11	perform selectors	2.19	sort strings
0.12	evaluate blocks	0.86	intersect rectangles
0.41	geometric mean	0.71	geometric mean

SMOPSTONES

The seven medium-level tests are:

- Generating fractonaccis (like fibonacci, but using fractions)
- Generating prime numbers
- Generating and parsing streams
- Generating and manipulating strings
- Forming a set of strings
- Sorting this set
- Recursively creating sets of overlapping rectangles

Each test is repeated once using fixed values for its parameters. It can be repeated more times if necessary for fast machines. I used fractonacci rather than fibonacci because fibonacci runs were either too fast or generated 32-bit integers. Fractonacci fit within the constraints imposed by my goals.

ANALYSIS OF THE RESULTS

Digitalk didn't beat ParcPlace after all, at least in these benchmarks. The fastest version of ST/V for Intel machines ran at 41% of ST80 for the low-level tests and 71% for the medium-level tests. However, the numbers in the chart are the geometric mean of seven individual tests (`x1*x2*...*x7)**(1/7`). Digitalk did beat ParcPlace on some of the tests.

The results of comparing ST/V-OS/2 (32-bit) relative to ST80-Windows (32-bit) are presented in Table 4. Numbers greater than one mean ST/V is faster. Marten Feldtmann did these ST/V runs and I did the ST80 runs.

Table 4 suggests the two vendors have optimized different parts of their systems. For example, on the low-level tests, the two versions add integers at about the same speed, but Digitalk is quite inefficient at performing selectors and evaluating blocks without arguments. ParcPlace is consistently better on the remaining tests by a factor of two.

For the medium-level tests, Digitalk whips ParcPlace on sorting. Perhaps this is because Digitalk's string compare is better or perhaps they are using a better sorting algorithm. I haven't checked. Digitalk also beats ParcPlace on fractonaccis with the same margin they won on Marten Feldtmann's fibonacci test. I think this is because Digitalk is faster on tight, recursive block or method calls and—or—because of the performance penalties ParcPlace pays for full blocks and copying blocks versus clean blocks, a distinction I doubt Digitalk makes. However, Digitalk got clobbered on the stream tests, possibly because it is slow at mixed mode arithmetic between integers and floats. And it fared badly on set formation, probably because its hashing algorithm for strings is less effective than the sophisticated one used by ParcPlace, especially for ST/V Mac 1.2.

If we omit these stream, set, and sort tests from Smopstones, the 32-bit version of ST/V for OS/2 comes in at 98% of the geometric mean of ParcPlace's 32-bit version for Windows—a dead heat.

There is a wide divergence between the low- and medium-level results. ST/V-OS/2 rose from 0.41 on Slopstones to 0.71 on Smopstones. The most dramatic rise was for ST/V-DOS. It rose from 0.070 to 0.261 on a 486/33 and from 0.002 to 0.008 on an 8088/4.77. Perhaps ST/V-DOS bogs down more on Slopstone garbage collection than on Smopstones. This is pure speculation. The general advantage that ST80 has over ST/V in low-level tests relative to medium-level ones may be caused by the performance penalties that ST80 pays for distinguishing between clean, copying, and full blocks. I may have written the medium-level tests to be more susceptible to this distinction. More tests would be needed to verify this hypothesis. Recall above how I sped up the while loop for ST80 in Marten Feldtmann's test by converting a full block to a clean one.

ParcPlace would have come off slightly better if the tests were not constrained by portability. Some of the code could have been shortened by using ParcPlace's larger class library. More significantly, some of the variables that are declared as method temporaries could have been declared as block temporaries, thus converting some dirty blocks to clean ones. This would have left Slopstones unaffected, but would have improved ParcPlace's relative Smopstone performance by 2.5% on average, with the biggest gain coming in the intersecting rectangles (19%).

ParcPlace's syntax for declaring block temporary variables is shown below. ST/V-DOS does not support it. I don't know whether newer Digitalk versions do.

```
[:arg1 :arg2 |
| temp1 temp2 temp3 |
statements]
```

ST/V-Windows comes off rather poorly. I don't know whether this is because Digitalk hasn't optimized it as much as their OS/2 versions or because the only test ran it under Windows which itself ran under OS/2. Perhaps it would run faster under native Windows. A recent COMPUTERWORLD article says that Digitalk is beta testing a new Windows version based on The Win32s 32-bit interface. It yields "a big performance boost" and is expected to be ready in July.

You definitely want to run ST/V-DOS under native DOS rather than in a DOS shell under Windows. In the latter it runs at only 62% of native capacity.

Although ST/V-Windows beat ST/V-DOS by 0.167 to 0.070 on Slopstones for a 486/33, they came in nearly tied on Smopstones. Moreover, the individual Smopstone tests, which are not given in this article, were quite close for the two versions. Since the DOS version I tested was 1987 vintage or earlier (its file dates were 1987), this suggests that Digitalk's Windows version is in need of a performance tune-up. I don't understand the divergence between low- and medium-level results.

The Macintosh results for ST/V aren't too bad if you omit streams and sets from Smopstones. I haven't included the individual runs, and I don't have a Mac version. I think the same theory as outlined earlier applies; namely, ST/V-Mac must be really bad on mixed mode integer-float arithmetic, which streams use, and absolutely terrible on string hashing, which set formation uses. I've heard that a new Mac version may be shipping by the time this article is published.

Ideally, the tests comparing ParcPlace to Digitalk should be made on the same machine. I am assuming that the 486/33 on which I tested ParcPlace is about equal to the 486/33 on which Marten Feldtmann tested Digitalk. Similar comments can be made about the Macs. Although my machine benchmarks faster than Marten's on ParcPlace's Dorado benchmarks, I think this is due to differences in our video cards.

There are several means one could report. Three popular ones are:

- Arithmetic mean = (x1+x2...+xn)/n
- Harmonic mean = n/((1/x1)+(1/x2)...+(1/xn))
- Geometric mean = (x1*x2...*xn)**(1/n)

I chose geometric mean because it has the best scaling properties and it is the least sensitive to one number being particularly low or high. The ParcPlace Dorado benchmarks use harmonic mean. When I first posted the benchmarks to comp.lang.smalltalk, I used geometric mean, but erroneously called it *harmonic*. Urs Hölzle corrected me.

The benchmarks have several shortcomings, some of which were pointed out by people posting to comp.lang.smalltalk. There should be a lot more than seven tests in each suite. They concentrate on too few areas of Smalltalk and omit many of the diverse capabilities of its class library. With so few tests, they could be sensitive to one particularly weak link in an implementation. Examples of such links we probably encountered were a bad string hash function for ST/V (especially for Mac), weak floating point performance for ST/V, poor string compare or sort algorithm for ST80, and possibly poor recursion or poor performance of nonclean blocks in ST80 (e.g., fibonacci and fractonacci, and especially Marten Feldtmann's while loop.).

Two low-level tests I wish I had included in Slopstones would have been to test direct method dispatch efficiency with

```
Object new yourself; yourself; yourself...
```

and then to test inherited dispatch with something like

```
Dictionary new yourself; yourself; yourself...
```

This would have determined the absolute maximum number of method dispatches that can be performed per second. In an informal test, direct dispatch with ST80 ran at the same speed as integer addition. On a 486/33, this means you get a maximum of 6.6 million dispatches per second. Most machine language instructions probably run in one clock cycle, yielding 33 million machine language operations per second (MIPS). The actual mips rating of a 486/33 is perhaps half or two third this, but is still much higher than its MDPS (million dispatches per second) rating.

Other additions to Slopstones could be using arguments when evaluating blocks and performing selectors. One could imagine many more, too.

I received some suggested additions to Smopstones by email after I had already frozen them. I also should have included some of the benchmarks already developed by the Self group. Richards was donated to them by Peter Deutsch, formerly of Parc Place.

The benchmarks are not at a level high enough to test actual applications. Slopstones, in particular, is hardly a predictor of real-life performance. However, by comparing ST80 and ST/V on low-level and medium-level operations, the style of benchmark we have written does shine the spotlight on operations that need to be optimized. If Slopstones and Smopstones were made more comprehensive, this could help the vendors find areas in which their performance is not competitive.

The benchmarks do not test the speed of user interactions such as opening windows or scrolling lists. These consume a lot of a user's time in practice. Nor do they test how quickly Smalltalk accesses disk files, which can be important in some applications. For example, I can read an ASCII file on my 486/33 machine running ParcPlace at only 40K bytes per second when doing high-level access with contentsOfEntireFile. Although much higher rates are achievable with IOAccessors (in the 1MB range), this is low-level and inconvenient.

Although the benchmarks are portable between ST80 and ST/V, they are less portable to other languages. Urs Hölzle ported Slopstones to Self easily enough, but couldn't easily port Smopstones because Self lacks streams and fractions. A user of GNU Smalltalk couldn't port Smopstones because GNU lacks rectangles and fractions. Although I hadn't anticipated the benchmarks being run for any languages except ST80

and ST/V, if I had used more plain vanilla classes, they could have had a wider reach.

Slopstones is so low-level that many of its individual tests may get completely optimized away by the compiler. I knew this wouldn't happen with current Smalltalk compilers, but it did happen when Urs Hölzle compiled it under Self.

The Smopstone test for sorting a set of strings was subtly flawed. The raw material for the sort was different for ST80 and ST/V because the ST80 Set enumerates an instance from low index to high, while ST/V-DOS enumerates from high to low. Moreover, the sets being sorted are hashed differently resulting in a different ordering of their elements. If I had sorted the original array of strings rather than the derived set of strings, this flaw would be removed. The result of doing this is to slow down the ST80 sort speed by 5–10% while leaving the ST/V sort speed virtually unchanged. In other words, ST/V wins the sort test by 5–10% more than in the Smopstone chart.

CAN SMALLTALK PERFORMANCE BE FURTHER OPTIMIZED?

The benchmarks in this article show that there are areas in which ST80 excels over ST/V and others in which ST/V excels. This suggests that both ParcPlace and Digitalk could wring out better performance by conventional means. As for more exotic optimizations, the Self researchers claim the answer to the question is most definitely *yes*, both in their publications and in private conversations. Vendor representatives are less convinced. I have only talked with ParcPlace, but my impression is that they either feel it is not technically feasible to achieve Self performance in a commercially viable way (e.g., without requiring 64MB machines), or it would be too expensive for them to do it, or their customers do not regard it as a priority.

Last May there was a flurry of discussion in comp.lang.smalltalk on Smalltalk efficiency. One thread focused on the ParcPlace virtual machine, and in particular, on whether using register windows in native machine code on Sun Sparc platforms would speed it up much. A second thread focused on whether Self optimization techniques could be applied profitably to Smalltalk. After considerable discussion, Peter Deutsch made a summary statement for both threads. He used to be with ParcPlace and has considerable experience in implementing and optimizing Smalltalk. Regarding the second thread, he expressed the following private opinion (his views do not necessarily reflect those of his employer):

> As for the comparison [of Smalltalk] against Self, the Self authors acknowledge that the factor of 5 [improvement of Self over Smalltalk] is only achievable under some circumstances. I do think it would be exciting to apply the Self compilation ideas to Smalltalk, and doing this could well produce dramatic performance improvements (on all platforms), but this would require wholesale redesign of most of the platform-independent code (other than the memory manager) in the [ST80] runtime support system. The optimizing compilation experiments I did at ParcPlace were based on an alternative approach that would not have required such substantial changes to the [ST80] virtual machine, but might have required type declarations (or at least type hints) provided by the user (or a type inference system).

I don't know whether ParcPlace has continued their experiments or whether Digitalk has any active projects to push toward Self's performance. It is interesting that two of the prime movers, Peter Deutsch (Smalltalk) and David Ungar (Self) have moved respectively from ParcPlace Systems and Stanford University to Sun Microsystems. I wonder what Sun has up its sleeve?

In conclusion I would urge you to let your vendor know if performance optimization is important to you. Report serious bottlenecks to them. I have found ParcPlace to be quite responsive in correcting them.

COMPILING AND RUNNING THE BENCHMARKS

The benchmarks require floating point hardware or emulation software. They compile and run without difficulty on all the versions of ST80 and ST/V for which they have been tested. At least two Smopstone benchmarks, fractonacci and rectangle intersection, won't run under GNU Smalltalk because it lacks Fraction and Rectangle classes.

It is a good idea to file the code into a clean image and do a garbage collect before running it if possible. The individual times will fluctuate somewhat, but the geometric mean is pretty stable. You can reduce fluctuations by running more iterations (the n variable in the execute method). Doing so for ST/V-DOS may crash it though.

Be sure to run ST/V-DOS benchmarks under native DOS. For example, Smopstones in a full screen DOS shell under Windows only runs at 62% of its speed under native DOS.

Mail the results to me or post them to comp.lang.smalltalk. If you want to try the Self performance suites, contact self-request@self.stanford.edu. Or ftp from the directory benchmarks/st80-2.4.

SOURCE CODE

You may ftp the source code from the public domain Smalltalk archives at the University of Illinois (st.cs.uiuc.edu 128.174.241.10) or University of Manchester (mushroom.cs.man.ac.uk 130.88.13.70). ■

REFERENCES

1. G. Krasner. SMALLTALK-80, BITS OF HISTORY, WORDS OF ADVICE, Addison-Wesley, Reading, MA, 1983.
2. Chambers, C., and D. Ungar Making pure object-oriented languages practical, OOPSLA 91 CONFERENCE PROCEEDINGS; also published as SIGPLAN Notices 26.11, November 1991.

Bruce Samuelson uses ParcPlace Smalltalk for linguistic applications at the University of Texas at Arlington and with the Summer Institute of Linguistics. Bruce can be reached via internet at bruce@utafll.uta.edu (uta-eff ell ell).

continued from page 8

the outside world. Messages should present the services an object is willing to provide. Using them to give an abstract view of storage turns those implementation decisions into yet more services. Revealing implementation is exactly what encapsulation is supposed to avoid.

"Just make the accessors private." That's the common solution, but there are two reasons why this isn't a sufficient solution. First, anyone can invoke any method (and will, given enough stress). There is currently no way to make truly private methods that cannot be used outside the class. Digitalk and ParcPlace are both working on this problem. More seriously, programmers are notoriously bad at deciding what should be private and what should be public. How many times have you found "just the right method," only to find it marked private? If you use it, you are faced with the possibility that it may go away in the next release. If you don't, you have to violate the encapsulation of the object to do the computation yourself, and you have to be prepared for that computation to break in the next release.

The argument against automatically using accessors rests on the assumption that inheritance is less important than encapsulation. Rick DeNatale of IBM argues that inheritance should be kept "in the family." Anytime you inherit from a class you don't own, your code is subject to unanticipated breakage much more than if you merely refer to an object. If you want to use inheritance, do it only between classes whose change you control. While this may not result in the most elegant solution, it will save you headaches in the long run.

Using this model, you can access variables directly. If you want to make a subclass that needs to access a variable through a message, you use the programming environment to quickly change "x := ..." into "self x: ..." and "x ..." into "self x ...". Encapsulation is retained, and the cost of changing your decision is minimal. If you don't own the superclass or the subclass, you can't do this, as it would involve making changes in code you can't control.

CONCLUSION

Aesthetics does not provide a compelling argument one way or the other. There's a giddy feeling when you make a subclass the original programmer never anticipated, but only need to make a few changes to make it work. On the other hand, there is satisfaction in thinking you finally have to reveal a variable, only to discover that by recasting the problem you can improve both sender and receiver.

Regardless of how you choose to program, you are faced with the hard choice of deciding which variables should be reflected as messages. Pushing behavior out into objects rather than just getting information from them and making decisions yourself is one of the most difficult, but most rewarding, jobs when programming objects. Making an accessing method public should be done only when you can prove to yourself that there is no way for the object to do the job itself. Making a setting method public requires even more soul-searching, since it gives up even more of an object's sovereignty.

Either way, you accept a discipline not supported by the language. If you choose to use accessors, you and everyone who uses your code must swear an oath never to send messages that invoke methods marked private in the receiver. You also must be wary of using the accessor from outside the object when you really need to add more services to the receiver. If you do not use accessors, you accept the burden of refactoring classes, either making an abstract class or at least adding accessors, should a later inheritance decision make it necessary.

> “Programmers are notoriously bad at deciding what should be private and what should be public.”

Whichever style you choose, make sure it pervades your team's development. Einstein is reputed to have said, "You can be consistent or inconsistent, but don't be both." The same simplifying assumptions should hold throughout all of your code.

If you use accessors, make them all private at first. Only make them public if you must, and struggle to discover a less centralized solution first. Don't assume that because you access variables through messages you have made all of the abstraction decisions you'll have to make. Using an accessor, internally or externally, should alert you that there may be missing behavior.

If you use variables directly, be prepared to recant your decision when the time comes. If what you thought was state is really a service, make the change everywhere. Don't have external users getting a variable's value through a method and internal users accessing it directly.

So, what's The Answer? In my own code, I change state into service (define an accessing or setting method) only when I am convinced it is necessary. Otherwise, my classes access their variables directly. I think inheritance is overrated. Providing the right set of services has more bearing on the success of a design. There are plenty of successful, experienced folks who would call me a reactionary hick for this (and worse things, for other reasons). Try some code each way and decide for yourself which style you find more comfortable. That's the only right answer. ■

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPar Computer. His is founder of First Class Software, which develops and distributes reengineering products for Smalltalk. He can be reached at First Class Software, P.O. Box 226, Boulder Creek, CA 95006, by phone at 408.338.3666, or on CompuServe at 70761,1216.

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.

ICONIX Software Engineering's **ObjectModeler** now supports Smalltalk. ObjectModeler is an OOA/OOD/OOP module. This recent addition was made in response to the developing trend in the object-oriented market that more and more COBOL and IS shops are moving into Smalltalk while technical shops continue to move into C++.

ICONIX ObjectModeler already supports C++ and SQL development and the company believes that the addition of Smalltalk will be of particular interest within the IS market. ObjectModeler users already have the ability to attach text files to any symbol on a Rumbaugh, Coad/Yourdon, or Booch diagram within ObjectModeler. In the same way that C++ and SQL templates are used to link source code to diagrams, they can now pick from 9 menus containing over 270 Smalltalk language constructs.

ICONIX Software Engineering, 2800 28th St., Suite 320, Santa Monica, CA, 310.458.0092 (v), 310.396.3454

WindowBooster is a simple and powerful utility that optimizes the opening of windows and dialog boxes programmed using Digitalk's Smalltalk/V. WindowBooster significantly improves the overall speed of any application. The product is easy to install, transparent to the user, and compatible with products such as WindowBuilder. The product is available for Windows and OS/2 and includes complete source code.

Tau Ceti, 1801 Avenue of the Stars, Suite 404, Los Angeles, CA 90067-5906, 310.556.9723 (v), 310.556.9725

Tensegrity is an object-oriented database system for Smalltalk. Using Tensegrity, Smalltalk developers can create single-user or multi-user network applications without changing code. The product provides transparent object persistence, advanced transactional capabilities, two-phase commit, distributed garbage collection, and exceptional speed. Because the product is network-independent and requires no dedicated database server, the company anticipates that it will have great appeal to developers of workgroup applications.

Polymorphic Software, 1091 Industrial Rd., Suite 220, San Carlos, CA 94070, 415.592.6301 (v), 415.592.6302 (f)

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