THE Smalltalk REPORT

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

SIGS Publications Advisory Board
Tom Atwood, Object Design
François Bancilhon, O_2 Technologies
Grady Booch, Rational
George Bosworth, Digitalk
Adele Goldberg, ParcPlace Systems
Tom Love, IBM
Bertrand Meyer, ISE
Meilir Page-Jones, Wayland Systems
Cliff Reeves, IBM
Bjarne Stroustrup, AT&T Bell Labs
Dave Thomas, Object Technology International

THE SMALLTALK REPORT Editorial Board
Jim Anderson, Digitalk
Adele Goldberg, ParcPlace Systems
Reed Phillips, Knowledge Systems Corp.
Mike Taylor, Digitalk
Dave Thomas, Object Technology International

Columnists
Kent Beck, First Class Software
Juanita Ewing, Digitalk
Greg Hendley, Knowledge Systems Corp.
Tim Howard, RothWell International
Ed Klimas, Linea Engineering Inc.
Alan Knight, The Object People
William Kohl, RothWell International
Mark Lorenz, Hatteras Software, Inc.
Eric Smith, Knowledge Systems Corp.
Rebecca Wirfs-Brock, Digitalk

SIGS PUBLICATIONS GROUP, INC.
Richard P. Friedman, Founder & Group Publisher

Editorial/Production
Kristina Joukhadar, Managing Editor
Susan Culligan, Pilgrim Road, Ltd., Design
Seth J. Bookey, Production Editor
Margaret Conti, Advertising Production Coordinator
Tanya Trowell, Editorial Assistant
Brian Sieber, cover illustration

Circulation
Bruce Shriver, Jr., Circulation Director
John R. Wengler, Circulation Manager

Advertising/Marketing
Shirley Sax, Director of Sales
Gary Portie, Advertising Manager, East Coast/Canada/Europe
Michael W. Peck, Advertising Sales Assistant
Sales Representative: Diane Fuller & Associates, West Coast
408.255.2991 (v), 408.255.2992 (f)
Sarah Hamilton, Director of Promotions and Research
Caren Polner, Promotions Graphic Designer

Administration
Margherite R. Monck, General Manager
David Chatterpaul, Accounting Manager
James Amenuvor, Bookkeeper
Michele Watkins, Special Assistant to the Publisher
Joanna Lowenstein, Administrative Assistant

SIGS PUBLICATIONS

Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, SMALLTALK REPORT, THE X JOURNAL, REPORT ON OBJECT ANALYSIS & DESIGN, OBJECTS IN EUROPE, DIRECTORY OF OBJECT TECHNOLOGY, and OBJEKT SPEKTRUM (Germany)

Table of Contents

September 1994 **Vol 4 No1**

Features

Columns

Departments

The Smalltalk Report (ISSN# 1058-7976) is published 9 times a year, monthly except in Mar–Apr, July–Aug, and Nov–Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. Mailed First Class. Canada Post International Publications Mail Product Sales Agreement No. 290386.
Subscription rates 1 year (9 issues): domestic, $79; Foreign and Canada, $114. To submit articles, please send electronic files on disk to the Editors at 885 Meadowlands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).
POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, P.O. Box 2027, Langhorne, PA 19047. For service on current subscriptions call 215.785.5998, 215.785.6073 (fax), P00976@psilink.com (email). PRINTED IN THE UNITED STATES.

Editors' Corner

We have just returned from attending the ObjectWorld conference in San Francisco and the first ParcPlace International Users Conference in Santa Clara. The attendance at the latter was a big surprise, not only to us but also to the organizers as 500 Smalltalk users gathered together in what must have been the largest gathering of Smalltalkers ever in one place. Here's a quick review of both conferences, from a Smalltalk perspective.

ObjectWorld is a large conference and exposition with more than 90 exhibitors, which, unlike OOPSLA, has no booth size or height restrictions. The theme of the conference was "Get Real," and this was reflected in the large number of presentations by early adopters of object technology who now have real experiences to pass on to others. The most popular buzzwords remain "distributed," "client-server," "interoperability," and "object-oriented."

Smalltalk was much in evidence, although two of the major vendors, Digitalk and ParcPlace, took the approach of being present in the booths of their partners rather than having booths of their own. IBM unveiled their new IBM Smalltalk product alongside their VisualAge application development environment.

The most striking Smalltalk presence however was found in the booths of the database vendors. In the past only a few OODBMS-supported interfaces to Smalltalk. In recent months this situation has changed dramatically. At ObjectWorld, Objectivity introduced Objectivity/DB for Smalltalk (their interface to VisualWorks/Smalltalk) and Object Design was showing ObjectStore for Smalltalk (the result of their partnership with ParcPlace Systems). Servio introduced their 4.0 release of GemStone. Their booth featured an innovative presentation on the impedance mismatch between objects and relational databases complete with "Relational Wall" that the poor hapless relational database used to bump into. On the nonobject database front the major activity has been on bridging the gap between Smalltalk and relational databases; permitting Smalltalk programmers to manipulate relational data as objects. Both the ObjectLens feature found in the new Database Application Creator component of VisualWorks 2.0 and the UniSQL Smalltalk interface provide developers with the ability to create classes from relational tables or create tables from Smalltalk class definitions.

For a number of years we have advocated that the best approach to CASE for Smalltalk developers would be the integration of tools into the Smalltalk environment that would permit developers to move bidirectionally and seamlessly between analysis, design and implementation. A major objective of such systems is to keep information from each phase consistent in the presence of changes. This is a major task and developments are still in their infancy, but the first fruits and potential of this approach can be seen in the initial release of the Synchronicity product for Enfin Smalltalk, where changes made to the business model are reflected in the Smalltalk implementation and vice versa.

JOHN PUGH

PAUL WHITE

Finalists for the ComputerWorld Object Applications Awards included a number of projects utilizing Smalltalk technology; Caterpillar's system for forging steel plating and problem resolution and two production client-server systems built with Enfin Smalltalk at Sprint and Canadian Tire.

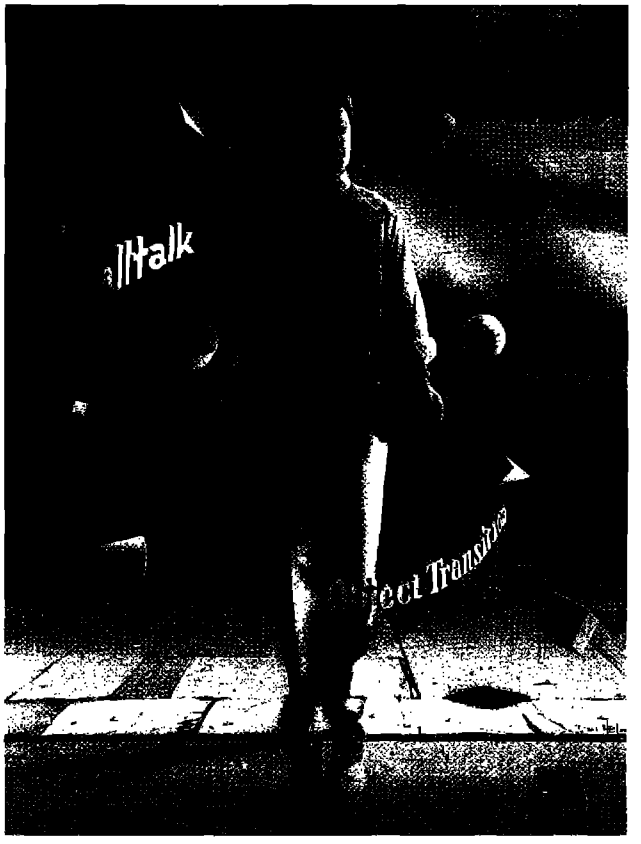
The ParcPlace Users Conference opened with two half-day tutorials; the first on Object Behavior and Design (OBA/D) ParcPlace's object-oriented analysis and design methodology; and the second aimed at developers moving their applications from VisualWorks 1.0 to 2.0. Many of the technical sessions were devoted to the new capabilities added in the new release of VisualWorks 2.0 such as interacting to relational database report writing and business graphics and connecting to C and DLL's. One of the most significant recent developments in the Smalltalk world has been its adoption by major systems integrators such as Anderson Consulting, EDS, and American Management Systems. Conference presentations featured discussion of the AMS ObjectCore framework, which extends the VisualWork's environment in areas such as data validation security and internationalization. The conference also included an exhibition by approximately 20 vendors of Smalltalk-related products and services and a reception held at the Tech Museum of Innovation in San Jose, CA. The highlights of the conference, however, were provided by the inspiring and thought-provoking keynote speeches of Adele Goldberg and Alan Kay; two of the original Smalltalk pioneers.

Finally, thank you for all your positive comments regarding the new look of THE SMALLTALK REPORT. We have passed on your praise to those responsible.

—The Editors

STEP INTO THE FUTURE WITH THE COMPANY THAT DEFINED OBJECT TECHNOLOGY SERVICES

When object oriented programming was in its infancy, Knowledge Systems Corporation was already putting it to work in companies like yours. Today, we're positioned to take you into the future of object technology in ways that no other company can. With the most complete range of services in the industry, KSC can assure your successful object transition every step of the way. Classroom instruction, project-focused apprenticeships, and consulting are all part of our exclusive commitment to object technology services.



Once you've made the decision to move to object technology, you want to get the benefits as quickly as possible. KSC offers a complete curriculum of classroom education, at your site or in our corporate training facility. These courses help you establish a firm foundation in object technology concepts and Smalltalk programming.

To cut months off your transition time, we've developed an exclusive Smalltalk Apprentice Program (STAP). Already proven in companies such as American Management Systems, GE Capital Corporation, IBM, Northern Telecom, The Prudential, Southern California Edison and Sprint, the STAP is a total immersion, project-focused program that compresses six to ten months of learning experience into four to six weeks.

KSC can also tackle your object technology projects head-on with the most experienced analysts, designers and programmers in the business. You can outsource the entire job, or use our consultants to lend expertise to your own development group.

In addition to our service offerings, KSC is a distributor of third party tools such as ENVY®/Developer, the premier Smalltalk team development environment.

If you're ready to step into the future of object technology, call the one company that will lead you there—Knowledge Systems Corporation, 919-481-4000. Or email: salesinfo@ksccary.com. 4001 Weston Parkway, Cary, North Carolina 27513.

KNOWLEDGE SYSTEMS CORPORATION

919-481-4000

ENVY is a registered trademark of Object Technology International Inc.

Name spaces in Smalltalk/V for Win32

Wayne Beaton

All in Smalltalk is not perfect. However, Smalltalk is better than anything else that's available. One of the major problems with Smalltalk is its annoying tendency to dump all classes into a single global name space. This means that if I have a class named "Employee," then nobody else had better have one. There are those (including myself) who would say that if you have a conflict between classes named "Employee," then chances are you've implemented something incorrectly. But never mind that.

For years, programmers have been exchanging useful Smalltalk code quite freely; today, there are a number of vendors who are selling Smalltalk classes that provide many wonderfully useful facilities. With more and more classes being included in images, programmers are starting to prefix their classes to ensure that their names are unique. For example, if I suddenly were to become insane, I might prefix every class I make with **WTB** (Wayne Thomas Beaton). My image then could be polluted with classes named **WTBSortedDictionary**, **WTBEmployee**, etc. Compound this with the annoying abbreviations that some people choose to employ, and you can have an entire image filled with classes named **WTBSrtdDict**, **WTBGblEmployee**, and **WTBLclEmployee**. What do these classes do? Whatever.

When I get a bunch of code from somebody else, I generally don't want to know how it works. That it works is enough (I imagine that this is the philosophy behind DLLs). All that I really want to know is the public interface (thoroughly documented, of course). The bottom line is, I don't want to have to look at stuff that I don't need to look at. If the code requires some 50 classes, then great — just don't make me look at them.

```
Object
    subclass: #Junk
    instanceVariables: ''
    classVariables: ''
    poolDictionaries: '' !

Smalltalk at: #JunkClasses put: Dictionary new !

JunkClasses at: 'Junk' put: Junk !

Smalltalk removeKey: #Junk ifAbsent: [] !
```

Figure 1. File-in code to define a class and move it into a pool dictionary.

```
Object
    subclass: #UsesJunk
    instanceVariables: ''
    classVariables: ''
    poolDictionaries: 'JunkClasses' !

!UsesJunk methods !
answerANewJunk
    ^Junk new ! !
```

Figure 2. Using a class stored in a pool dictionary.

What is needed to bring Smalltalk to the next level, is some way of permitting programmers to use the names that make sense for their classes without worrying about conflicting with other names, and the ability to hide away classes that nobody else particularly cares about.

In the February 1993 issue of THE SMALLTALK REPORT, Nik Boyd discussed modules for Smalltalk. Modules provide a way of hiding "private" classes. This provides some of the desired behavior, but more is needed.

HAVE I GOT A DEAL FOR YOU...

While this is not particularly clever, radical, or new, it's something that should have been done years ago. In fact, I marvel that this hasn't shown up yet; there are many very smart people in the Smalltalk community. In all fairness to Digitalk, their code seems to indicate that they are leaning in this direction (and in fact, Smalltalk Agents has had this from the beginning).

Smalltalk needs name spaces. The global variable Smalltalk is a name space; this is where all the global names are kept. We should be able to have more than just one name space and use it just like we use the global name space. In fact, we can do this already.

Pool dictionaries are moderately useful things that can be used to hold information that is to be shared between classes. In our code, pool variables are accessed just like global variables. Pool dictionaries can hold many wonderful things. In fact, they can hold any object — numbers, strings, collections, classes, etc.

When a class is defined, it is created automatically as a global variable. After a class has been created, we easily can include it in a pool dictionary and then remove the global reference.

Global variables are all kept in the global variable called Smalltalk. Smalltalk is an instance of the class **SystemDictionary**. When you attempt to remove a reference to a class from Smalltalk using the method **removeKey:**, the class itself is removed from the system. However, the method **removeKey:ifAbsent:** does remove the key from Smalltalk, but it has no effect on the class.

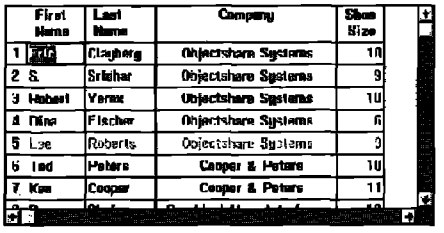
Figure 1 shows some code that can be filed-in to create a class and then moved into a pool dictionary. Figure 2 demonstrates how this class can be used by another. While the magnitude of the usefulness of the demonstration method may not be appreciated, it is obvious that from a use standpoint, very little is different. Only a mention of the pool dictionary is required in the class definition to use the class-in-a-pool-dictionary.

We can now freely define another class called Junk, and either keep it as a global variable, or move it to another pool dictionary.

When the compiler encounters a variable that begins with an up-

Get Powerful New Controls for Smalltalk/V®

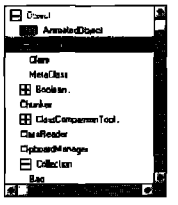
Subpanes™/V is a library of unique controls for Smalltalk/V. Place and edit them interactively with WindowBuilder™ Pro/V. When you use the right controls, your applications will be easier to use. And you'll save time because you won't need to fight controls that aren't right for the job.



A Table of Editable Cells

TablePane provides a scrollable grid of editable cells. In addition to handling a matrix of strings, it can manage a collection of objects. Users edit cells in-line by selecting them with the mouse or keyboard.

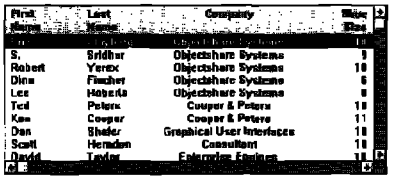
Hierarchical List Box



HierarchicalListBox extends a normal listbox to view a hierarchical group of objects. Collapse or expand the hierarchy, use icons, use indentation to show the relationships. Display any objects that have hierarchical relationships.

A List Box with Columns

ColumnarListBox displays multiple pieces of information about each object of a collection. You control headers, justification, color*, multiple select* and more.



Bitmap Panes, 3D Frames, & More

Subpanes/V also includes BitmapPane, 3D frames, ValueSet, Gauges, date, number, and time editors, BitmapButton, and more.

No Runtime Fees

No runtime fees for applications developed with Subpanes/V. It includes complete documentation, full source, free support to registered users for the first 90 days, and a 30-day money-back guarantee.

SUBPANES/V

NEW! **For OS/2 ..... $235** *(v2.0)*
For Win ..... $129 *(v1.0)*
For Win32 ..... $195 *(v1.0)*

*These features in version 2.0 only. Version 2.0 for Win and Win32 will ship in 3Q94.

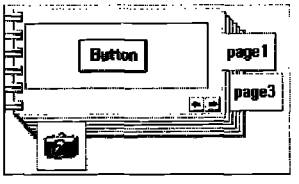
Subpanes/V requires WindowBuilder Pro/V. Subpanes /V is compatible with Team/V and ENVY/Developer. Subpanes is implemented in Smalltalk, as subclasses in Digitalk's Subpane hierarchy. Support subscription available.

...And CUA'91 Controls Are Easy Too!

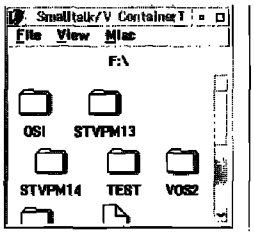
WidgetKit™/CUA'91 is a library of CUA'91 controls for Smalltalk/V. CUA'91 controls provide a distinctive and powerful user interface. WidgetKit/CUA'91 makes them easy to use and portable. Place and edit the controls interactively with WindowBuilder™ Pro/V. WidgetKit/CUA'91's specialized editors give you easy access to all of the control's attributes.

Notebooks, Cached for Performance

CachedNotebooks provide the CUA'91 notebook control. Performance is dramatically improved by dynamic page loading. You get complete control of orientation, tabs, alignment, color, binding, and caching.



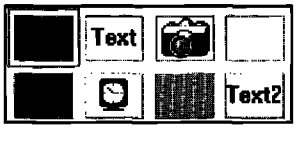
Containers



CuaContainers provide text or icon representations of items they contain. Items can be dragged and dropped between containers. Supports icon, name, text, tree, and detail views. CuaContainers can hold objects of any type.

Value Set and More

CuaValueSet provides a way for users to select from icon and text choices with a mouse click. WidgetKit/CUA'91 also provides full support for the rest of the CUA'91 controls, including slider and spin button.



For WindowBuilder Pro/V

WindowBuilder Pro/V lets you build Smalltalk/V user interfaces fast. Place the controls and edit them interactively. Increase consistency, ease maintenance. Call for a free brochure.

No Runtime Fees

No runtime fees for applications developed with WidgetKit/CUA'91. It includes complete documentation, full source, free support to registered users for the first 90 days, and a 30-day money-back guarantee.

WIDGETKIT/CUA'91

NEW! **For OS/2 ..... $295**
For Win ..... $295 *(3Q94)*
For Win32 ..... $295 *(3Q94)*

WidgetKit/CUA'91 requires WindowBuilder Pro/V. WidgetKit/CUA'91 is compatible with Team/V and ENVY/Developer. Includes DLLs. User interfaces built using WidgetKit/CUA'91 are portable to supported platforms. Support subscription available.

SHARE OBJECT SYSTEMS INC.

Objectshare Systems, Inc.
5 Town & Country Village, Suite 735
San Jose, CA 95128-2026
Fax 408-970-7282
CompuServe 76436,1063

Call to order today (408) 970-7280

9 AM to 5 PM PST, Monday through Friday
30 day money-back guarantee

Name spaces in Smalltalk/V

per case character, it first checks the pool dictionaries of the class and then Smalltalk (I imagine that class variables are also consulted at some point). We can count on our method using the Junk class defined in the pool dictionary before it attempts to use any other Junk class that we may create. There is some potential for trouble if a class uses more than one pool dictionary — the order in which multiple pool dictionaries are searched is unspecified (in the process of defining a class, a collection of pool dictionary names is created as a set and then turned into an array).

THAT'S GREAT, BUT...

This pool dictionary idea works well, with a couple of exceptions.

The browsing tools just aren't up to snuff. When the Class Hierarchy Browser populates its list of classes, it asks each class for its subclasses. The list of classes is actually a list of the names of classes. When the user actually clicks on a class name, the string is used to ask Smalltalk for the actual class (those readers that have been paying attention will notice the obvious problem with this).

It appears as though Digitalk has been anticipating the inclusion of name spaces in some future version of their Smalltalk products. In the process of creating a class, an instance of ClassInstaller is created; this ClassInstaller is provided with the "environment" in which to install the class. The "environment" is expected to be something with dictionary-like behavior; by default this always ends up being Smalltalk (an instance of SystemDictionary). Any other dictionary can be substituted (by changing the appropriate methods) — except that the key that is placed into the dictionary always will be a Symbol. The SystemDictionary called Smalltalk uses Symbols for its keys. Pool dictionaries use Strings (again, an attentive reader might notice that we have a bit of an inconsistency here...)

NAME SPACES

Pool dictionaries can be used to provide name spaces. However, it would be nice to have an easy-to-use facility to use name spaces seamlessly. There is perhaps more behavior that name spaces could have that could justify the creation of a new class. The class NameSpace provides the behavior required for name spaces in Smalltalk. This class, combined with a helper class and some small system changes, provides easy-to-use protocols for using name spaces.

Creating a new name space is a simple matter of passing the NameSpace class the message new and storing the result in a global variable. For convenience, name spaces know their name; this makes things a little easier later, when we need to know their name. Figure 3 shows how a name space can be created and stored.

NameSpaces have been provided with all the behavior required for them to perform as a SystemDictionary and as a pool dictionary. The implementation is simple: the class has two variables, name and dictionary. Methods have been provided to access the name space, just as a SystemDictionary or a pool dictionary is accessed. The methods at:, at:ifAbsent:, at:put:, includes:, and includesKey: work just as expected. The method add: is needed for when a new class is created; after the class is created, an association containing it is added to the name

space (this is part of the inner workings of Smalltalk — it really seems odd to be able to add an association to a dictionary...). The method associationAt:ifAbsent: also is required as part of the pool dictionary behavior; when a pool variable is accessed, the association containing the value is borrowed from the dictionary. I have added another method, @ (at-sign), which provides some "syntactic sugar" for the at: message.

All of the methods that involve a key convert the key into a symbol. This way it is possible to access a name space by either using a string or a symbol as the key.

Adding a class to a name space is another simple matter. When the class is created, the name space in which the class should reside is specified. As well, the names of all the name spaces that the class has access to are specified. Figure 4 shows how a class can be created in the name space called TestNameSpace with access to the name spaces called NameSpace1 and NameSpace2.

The name spaces are listed by their name, just like pool dictionaries. This is a consequence of how they are being stored and used. Currently, name spaces are associated with a class the same way a pool dictionary is — by their names. When the compiler needs to access a variable from a pool dictionary, it goes through its collection of pool names and asks Smalltalk for each one. This forces on us a restriction that name spaces must all be globally known. A small change to the method poolVariableScopeFor:, in class CompilerInterface, fixes this restriction to permit name spaces and pool dictionaries to reside within name spaces.

The method subclass:instanceVariableNames: classVariableNames:poolDictionaries: nameSpacesAccessed:nameSpace: has been added to the class Behavior; it adds a small delta to the original method subclass:instanceVariableNames::poolDictionaries:. The original method creates an instance of class ClassInstaller, loads it up with information for the new class and then sends the message install. The new method creates an instance of class NameSpaceClassInstaller that takes name spaces into consideration.

One of the instance variables of ClassInstaller is "environment"; ClassInstaller only looks at its environment for global information. By default, this variable holds Smalltalk. When an instance of NameSpaceClassInstaller is created, the environment variable is set to the name space specified in the formal parameter.

When the NameSpaceClassInstaller creates a new class, it first checks to see if the parameters it has been provided with are valid (the super class does most of the work). The name spaces specified are checked to ensure that they are in fact name spaces. Once everything

```
TestNameSpace :=
    NameSpace new name: #TestNameSpace
```

Figure 3. Creating a name space.

```
Object
    subclass: #Test
    instanceVariables: ''
    classVariables: ''
    poolDictionaries: ''
    nameSpacesAccessed:
        'NameSpace1 NameSpace2'
    nameSpace: #TestNameSpace
```

Figure 4. Defining a class in a particular name space that accesses classes in other name spaces.

has been validated, the name spaces are included with the pool dictionaries and the installation continues as normal.

The CompilerInterface method poolVariableScopeFor: was changed to look into the name space for pool dictionaries rather than directly into Smalltalk. Name spaces have further been extended so that the methods at:, at:ifAbsent:, includes: and includesKey: all look into themselves for the specified key first, and if the key is not part of the name space, the buck is passed to the outer name space (Smalltalk by default). The current implementation only provides for one level of name spaces (i.e., no name spaces within name spaces), but the potential for nested name spaces has been worked into the code.

Name spaces are stored in a class with pool dictionaries. For most purposes, this is acceptable — however, when a class prints itself, it would be better to not have the name spaces appear as pool dictionaries. The method fileOutOn: in class Class has been modified to accommodate name spaces. Methods have been added to extract name space information from a class; the methods nameOfNameSpace, namesOfNameSpaces, and sharedVariableString answer the name of the name space the class belongs to, the names of all the name spaces the class has access to, and a string containing the names of all the pool dictionaries that the class accesses, respectively.

WITH A LITTLE BIT MORE EFFORT...

This article discusses names spaces exclusively. It does not consider the modifications that have to be made to the browsing tools in order to make any real use of name spaces. As well, the code presented here could be changed easily to permit name spaces within name spaces. An even easier extension would permit variable and variable byte subclasses to exist in a name space.

CONCLUSION

This article has been written primarily as food for thought. The code provided here is in no way a complete solution.

It really seems odd that Smalltalk has lived as long as it has without some of the fundamental problems having been addressed. It seems that everybody that uses Smalltalk can tell you 40 things that are wrong with it. Fortunately, those same people can tell you four billion things wrong with the alternatives.

As more and more industrial strength applications are written in Smalltalk, and more classes are becoming commercially available, naming conflicts are only going to get worse.

Soon, Smalltalkers will be able to tell you 50 things that are wrong with Smalltalk, and only three billion things wrong with the alternatives. The gap is closing and Smalltalk has to evolve to keep ahead.

OH YEAH...

Alan Knight pointed out to me that I misused a term in a previous submission. I had made reference to functional programming when I had meant procedural programming. Perhaps with age I will gain a better appreciation for terminology... ♀

Wayne Beaton is a senior member of the technical staff at The Object People. His interests include user interfaces and neural networks. He can be reached at The Object People in Ottawa, ON, Canada, at 613.225.8812, or by email at wayne@ObjectPeople.on.ca.

Listing 1. The NameSpace class.

```
NameSpace methods for accessing
name
    ^name

name: aSymbol
    name := aSymbol

NameSpace methods for initializing

initialize
    self dictionary: self defaultDictionary

NameSpace methods for accessing with associations

add: anAssociation
    "When the compiler adds a class, it adds it as
    an association. Why? Nobody really knows
    for sure..."
    self
        at: (self convertKey: anAssociation key)
        put: anAssociation value

associationAt: aStringOrSymbol ifAbsent: absentBlock
    "When the compiler attempts to access a global, it wants the
      association. Checks the receiver for the name first, and then
      the outer scope."
    ^self dictionary
        associationAt: (self convertKey: aStringOrSymbol)
        ifAbsent: [
            self outsideScope
                associationAt: aStringOrSymbol asSymbol
                ifAbsent: absentBlock]

NameSpace methods for accessing by key

at: aStringOrSymbol ifAbsent: absentBlock
    "Answers the object found at the key named aStringOrSymbol. The
      receiver is checked first, then the outer scope. If the key is not
      found, then the evaluation of absentBlock is answered."
    ^self dictionary
        at: (self convertKey: aStringOrSymbol)
        ifAbsent: [
            self outsideScope
                at: aStringOrSymbol asSymbol
                ifAbsent: absentBlock]

@ aStringOrSymbol
    "Answers the object found at the key named aStringOrSymbol."
    ^self at: aStringOrSymbol

at: aStringOrSymbol
    "Answers the object found at the key named aStringOrSymbol."
    ^self
        at: aStringOrSymbol
        ifAbsent: [^self error: 'Key is Missing']

at: aStringOrSymbol put: anObject
    "Puts anObject at key position aStringOrSymbol."
    ^self dictionary
        at: (self convertKey: aStringOrSymbol)
        put: anObject

NameSpace methods for iterating

do: block
    "Evaluates block for each object in the receiver."
    self dictionary do: block

NameSpace methods for removing keys

removeKey: aSymbolOrString ifAbsent: absentBlock
    "Removes the object with key aSymbolOrString from the receiver."
    self dictionary
        removeKey: (self convertKey: aSymbolOrString)
        ifAbsent: absentBlock
```

NameSpace methods for testing

```
includes: anObject
    "Answers whether or not an object is included in the receiver or the
      outside scope."
    ^(self dictionary includes: anObject) or:
        [self outsideScope includes: anObject]

includesKey: aStringOrSymbol
    "Answers whether or not an object with key aStringOrSymbol is
      included in the receiver or the outside scope."
    self at: aStringOrSymbol ifAbsent: [^false].
    ^true

isNameSpace
    "Answers whether or not the receiver is a NameSpace."
    ^true
```

NameSpace methods for scoping

```
outsideScope
    "Answers the name of the receiver's outside scope."
    ^Smalltalk
```

NameSpace private methods

```
defaultDictionary
    "Private - Answer the dictionary to use by default."
    ^Dictionary new

convertKey: aStringOrSymbol
      "Private - Converts aStringOrSymbol into the appropriate object to be
      used as a key in the receiver."
    ^aStringOrSymbol asSymbol


dictionary: aDictionary
    "Private"
    dictionary := aDictionary

dictionary
    "Private"
    ^dictionary
```

Listing 2. Additions to Behavior class.

Behavior methods for name spaces

```
subclass: className
    instanceVariableNames: instanceVariables
    classVariableNames: classVariables
    poolDictionaries: poolDictionaries
    nameSpacesAccessed: namesOfNameSpaces
    nameSpace: nameOfNameSpace

    "Create or modify the class named <className> to be a subclass of the
      receiver with the specified instance variables, class variables, pool
      dictionaries, and class instance variables."
    | installer |
    installer := NameSpaceClassInstaller
        name: className
        environment: (Smalltalk at: nameOfNameSpace)
        subclassOf: self
        instanceVariableNames: instanceVariables
        variable: self isVariable
        pointers: true
        classVariableNames: classVariables
        poolDictionaries: poolDictionaries
        nameSpaces: namesOfNameSpaces
        nameSpace: nameOfNameSpace.
    ^installer install
```

Listing 3. The ClassInstaller class.

```
ClassInstaller subclass: #NameSpaceClassInstaller instanceVariableNames:
    ' namesOfNameSpaces nameOfNameSpace ' classVariableNames:
        "poolDictionaries: '"nameSpaces:" nameSpace: #Smalltalk category:
            'Name Spaces'
```

NameSpaceClassInstaller class methods for instance creation

```
name: className
    environment: globalDictionary
    subclassOf: superclassObject
    instanceVariableNames: instanceVariableString
    variable: variableBoolean
    pointers: pointerBoolean
    classVariableNames: classVariableString
    poolDictionaries: poolDictionaryString
    nameSpaces: namesOfNameSpaces
    nameSpace: nameOfNameSpace

    | installer |
    installer := self
        name: className
        environment: globalDictionary
        subclassOf: superclassObject
        instanceVariableNames: instanceVariableString
        variable: variableBoolean
        pointers: pointerBoolean
        classVariableNames: classVariableString
        poolDictionaries: poolDictionaryString.

    installer
        namesOfNameSpaces: namesOfNameSpaces asArrayOfSubstrings;
        nameOfNameSpace: nameOfNameSpace.

    ^installer
```

NameSpaceClassInstaller methods for accessing

```
nameOfNameSpace: aSymbol
    nameOfNameSpace := aSymbol

namesOfNameSpaces: aCollection
    namesOfNameSpaces := aCollection

nameOfNameSpace
    ^nameOfNameSpace

namesOfNameSpaces
    ^namesOfNameSpaces
```

NameSpaceClassInstaller methods for installing

```
namesOfPoolsAndNameSpaces
    "Private - Answers an array containing the names of the pool dictio-
      naries and the names of all name spaces."
    ^self poolNames,
        self namesOfNameSpaces,
        (Array with: self nameOfNameSpace)

editSubclass
    "Create or change the subclass the receiver should install. Overrides
      the superclass implementation to include name spaces with the pool
      dictionaries."
    ^self metaclass
        name: self className
        environment: self environment
        subclassOf: self superclass
        instanceVariableNames:
            self instanceVariableNames
        variable: self isVariable
        words: true
        pointers: self isPointers
        classVariableNames:
            self classVariableNames
        poolDictionaries:
            self namesOfPoolsAndNameSpaces
        comment: String new
        changed: nil
```

NameSpaceClassInstaller methods for validating

```
validate
    "Answer true if the receiver contains a legal class definition, false
      otherwise. Overrides the super class implementation to validate
      name spaces."
    ^super validate and: [self validateNameSpaces]
```

```
validateNameSpaces
    "Private - Ensures that the name spaces specified are all valid name
     spaces."
    (NameSpace isNameOfNameSpace: self nameOfNameSpace)
        ifFalse: [
            ^self invalidBecause: 'Name space is invalid'].
    self namesOfNameSpaces do: [:each |
        (NameSpace isNameOfNameSpace: each)
            ifFalse: [
                ^self invalidBecause: 'Name space is invalid']].
    ^true
```

Listing 4. Changes to CompilerInterface class.

CompilerInterface methods for name spaces

```
poolVariableScopeFor: aClass
    "Return a scope containing all of the pools for the argument class."
    | aPool poolVariableScope |
    poolVariableScope := MultiplePoolScope new.
    aClass sharedPools do: [:pn|
    "Look at the class name space. It will automatically look at the
     outer scope."
        aPool := "was —> Smalltalk" aClass nameSpace
            at: pn ifAbsent: [nil].
        aPool == nil ifFalse: [
            poolVariableScope add:
                (self scopeForPool: aPool named: pn)]].
    ^poolVariableScope
```

Listing 5. Changes/additions to Class class.

Class methods for name spaces

```
fileOutOn: aStream
    "Append the class definition message for the receiver to aStream.
     Extended to file out name space information correctly."
    | aString |
    aStream cr;
        nextPutAll: self superclass printString; space;
        nextPutAll: self kindOfSubclass; space;
        nextPutAll: name storeString; cr; space; space.
    self isBits
        ifFalse: [
            aStream nextPutAll: 'instanceVariableNames: '.
            (aString := self instanceVariableString) isEmpty
                ifFalse:[aStream cr; nextPutAll: ' ''].
            aStream
                nextPutAll: aString storeString;
                cr; space; space].
    aStream
        nextPutAll: 'classVariableNames: '.
    (aString := self classVariableString) isEmpty
        ifFalse:[aStream cr; nextPutAll: ' ''].
    aStream
        nextPutAll: aString storeString; cr; space; space;
        nextPutAll: 'poolDictionaries: '.
    (aString := self sharedVariableString) isEmpty
        ifFalse:[aStream cr; nextPutAll: ' ''].
    aStream
        nextPutAll: aString storeString.

    aStream
        cr; space; space;
        nextPutAll: 'nameSpaces: '''.
    self namesOfNameSpaces do: [:each |
        aStream nextPutAll: each; space].
    aStream
        nextPut: $'; cr; space; space;
        nextPutAll: 'nameSpace: #';
        nextPutAll: self nameOfNameSpace

removeFromSystem: checkForInstances
    "Private - Remove the receiver from Smalltalk. Report an error if
     there are any subclasses or instances of the receiver. If
     checkForInstances is true then we check if there are any instances of
     the receiver. Remove the receiver from its name space, not
     necessarily Smalltalk."
    | index index2 |
    ((OrderedCollection new
        add: UndefinedObject;
        add: Class;
        add: True;
        add: False;
        add: DeletedClass;
        add: EmptySlot;
        add: SmallInteger;
        yourself)
            includes: self)
                ifTrue: [
                    ^self error: 'Class cannot be removed.'].
    checkForInstances ifTrue: [
        self allInstances notEmpty ifTrue:
        [^self error: 'Has instances'] ].
    self allSubclasses notEmpty ifTrue:
        [^self error: 'Has subclasses'].
    self nameSpace "was —> Smalltalk"
        removeKey: self symbol ifAbsent: [].
    self class superclass == nil ifFalse:
        [self class superclass
            removeSubclass: self class].
    self class become: DeletedClass class.
    self superclass == nil ifFalse:
        [self superclass removeSubclass: self].
    self become: DeletedClass

nameSpace
    "Answers the name space the receiver is a part of."
    ^NameSpace named: self nameOfNameSpace

nameOfNameSpace
    "Answers the name of the name space the receiver is a part of."
    ^self sharedPools
        detect: [:each |
            (NameSpace isNameOfNameSpace: each)
            and: [(NameSpace named: each) includes: self]]
        ifNone: [#Smalltalk]

sharedVariableString
    "Private - Answer a String containing all of the pool dictionary names
     referred to by the receiver. The names are separated with blanks.
     Modified to exclude name spaces."
    | aStream pools |
    aStream := WriteStream on: (String new: 16).
    pools := self sharedPools reject: [:each |
        (NameSpace isNameOfNameSpace: each)].

    pools asSortedCollection do: [ :each |
        aStream
            space;
            nextPutAll: each ].
    aStream position = 0 ifFalse: [ aStream space ].
    ^aStream contents

namesOfNameSpaces
    "Answers a collection containing all the name spaces the receiver has
     access to."
    ^self sharedPools select: [:each |
        (NameSpace isNameOfNameSpace: each) and:
            [((NameSpace named: each) includes: self) not]]
```

Listing 6. Additions to MetaClass class.

MetaClass methods for name spaces

```
nameSpace
    "Answers the name space the receiver is a part of."
    ^self instanceClass nameSpace
```

Managing system changes with carriers

Panu Viljamaa

Good object-oriented classes adapt to the changing needs of their environment. Sometimes it is also important to adapt the environment to its objects. "System-changes" appear to be a necessity if rules of good design are obeyed as discussed in this article. With the reflective capabilities of Smalltalk we build classes that carry system-changes with them.

Much of the power of object-oriented development comes from the reuse of existing class libraries, the "environment" or the "system" on top of which applications are built. The impact of class libraries and tools is well appreciated by the vendors. What is lacking is an emphasis on techniques and approaches for assuring the incremental development of the environment.

In this article we argue for the importance of system-changes. We utilize reflective features of Smalltalk to build classes that carry source-code for the system changes they need with them.

THE IMPORTANCE OF SYSTEM CHANGES

A typical use of system changes could be to modify the system's menus to add your own tools to them. Another typical reason to change the system is porting. Instead of adapting an application to a new environment it is often economical to adapt the environment to the application.

Reflective capabilities of environments like Smalltalk introduce exotic possibilities for system changes. You can modify the compiler to allow for syntactic differences of different Smalltalks. Or modify it to accept a totally different language.

One important rationale for system changes is subtle: good style. Good object-oriented style often means dividing the responsibilities among classes so that robustness and reuse is maximized. The creators of the environment can't anticipate all future uses of their classes. When unanticipated usage occurs, the optimal division of work has to be reconsidered.

In an example of good and bad design, Beck[1] describes a way to avoid testing for an object's class. Testing for the class is bad because a decision is based on the class of the object, as opposed to its true capabilities. A bad method looks like:

```
someElement: aCollection
(aCollection isKindOf: IndexedCollection)
    ifTrue: [^aCollection first]
    ifFalse:[^aCollection asArray first]
```

To make the code better you can define the method #first for Collection (the root of all collections) to return ^self asArrayfirst. Then all collections understand #first and the complicated code-example can be replaced by:

```
someElement: aCollection
    ^aCollection first
```

To write in good style, we have introduced a system change. A system change appears to be the solution for good style in many other situations also.

The Law of Demeter[2] aims at reduced coupling between classes. It advocates moving functionality from the users of a class to that class itself. This allows that functionality to be reused by several clients. The law forbids chained messages. According to Demeter, the following is bad style because we are not only requiring the argument to understand #name but also that the result of #name understands #asUppercase:

```
badMethod: anArg
^anArg name asUppercase
```

We can rewrite the example in good style like this:

```
goodMethod: anArg
^anArg nameAsUppercase
```

Functionality is moved from the client into the server's method #nameAsUppercase. But when we create clients of system classes and we want to move functionality into them, we want system changes. The changes could be gathered in a subclass but it is a good idea to place functionality in the highest possible places because then they can be reused by subclasses. With system changes we make argument classes more similar, instead of making the recipient handle many dissimilar arguments.

The #someElement: example exhibits more reusability if #first is defined also in Object to return the object itself. After that #someElement: will work with any kind of argument. In general, we achieve the ability to treat collections and non-collections similarly and (re)use the same client-code with both kinds of arguments.

THE NEED FOR TOOLS TO MANAGE SYSTEM CHANGES

To reap the benefits that system changes can provide, they should be easy to use and reuse. We now look at how the conventional tools are lacking, and how they could be better.

ChangeSets is a traditional tool in Smalltalk-80. In ChangeListView individual changes can be browsed, removed, and added. The set of changes can be stored in a file and multiple files kept around. This is not particularly object-oriented however. The basic shortcoming is that files are not objects and therefore:

- The link between applications and their needed changes is weak. Files cannot be arranged in dependency relations. No one warns you if you delete a needed change-file. It is hard to know which applications need a given file.

- Files cannot be "subclassed" to redefine a given change but inherit others. Files may be treated as file-objects but they are *not* change-objects.

Instead of file-based change-management, we want change-objects that:

- Know the changes they contain.
- Carry the source code for these with them.
- Extract that source-code automatically from the current image.
- Are persistent and can be imported to standard environments.
- Can be worked on with standard tools like the browser.

The requirement of persistence with portability appears to be a hard one when there is no vendor-independent external representation for objects. Yet there is a common solution: the file-out-format of Smalltalk is a standard way for exporting and importing classes as portable, persistent objects.

THE CARRIERS

We now introduce software that allows you to treat changes as persistent objects—as classes. The source-code for such a class, called Carrier940701 is given in Listing 1. By subclassing it, you create your own change-modules, called "carriers." The code has been tested on Smalltalk/V for Macintosh 2.0 and Windows 1.1. It should be easy to port to other platforms also.

The basic design is to create methods automatically in the carrier that return the source-code for a given external method. You modify and test the system methods to be taken along, then ask the carrier to generate in itself the methods that return their sources. The method that returns the source-code for #method of AClass is named #AClassXmethod.

The class Carrier940701 carries one system-change itself, the method #systemChange of Object. The purpose of #systemChange is just to make possible the marking of places where the system has been changed. By looking at its senders you find those places. When Carrier940701 is #INSTALLED, all Objects gain the following method:

```
systemChange
"Called from places where system has been modified. Does nothing. By
 looking at my senders you find those places."
true ifFalse: [self Carrier940701]
    "Reference to the carrier of this method."
```

#systemChange calls "self Carrier940701" to create an explicit link back to the class that produced its source-code. The method named after the carrier (#Carrier940701) will be autocompiled into each target-class modified. Looking at methods named like carriers, you quickly see which ones have contributed to a given system-class. The created method in Object is:

```
Carrier940701
"Autocompiled by Carrier940701 (3/22/94 10:27). By looking at my
 senders you find the places it has modified."
true ifFalse: [self Carrier940701].
```

The changes carried are described by the result of a carrier's #systemChanges. Below is that method for Carrier940701. As shown, it specifies a single system-change, the #systemChange of Object, plus an example of specifying a class method, commented out:

```
systemChanges
"Return a list of pairs #(class selector) that tell the methods whose
 source-code I carry."
TEST: Carrier940701 FREEZE
"
| aSet |
    aSet := Set new.
        aSet add: (Array with: Object with: #systemChange) "—
            ; add: (Array with: Object class with: #COMMENT) —"
        . ^aSet
```

To make the changes of a carrier have effect, you send it #INSTALL. Based on the result of #systemChanges , it decides the names for the source-code-returning X-methods (in #carrySelectorFor:of:), calls them and compiles their results.

The code-returning methods are automatically generated from the current sources in the system. This happens when #FREEZE is called. Now it is possible to modify the sources as usual with the browser, instead of having to deal with the more complicated file-out format of source-code inside files.

#FREEZE calls #codeFor:of: to get the current source for a method, then #myCodeReturning:named: to generate the carrier's own method that will return that source.

APPLYING CARRIERS

To apply system changes effectively, we must detect and resolve conflicts between change-sets. All dependent changes must be carried along together. This introduces the problems of multiple replicants of the same carrier in separate modules, possibly out of sync if they are versioned.

If two carriers define the same method differently, there will be a conflict. But because the changes are explicitly de-

> *"Good object-oriented style often means dividing the responsibilities among classes so that robustness and reuse is maximized."*

clared, it is easy to build conflict-checkers, to be used *before* the changes are compiled. The versioning problem is solved by suffixing the creation date to the names of carrier classes and methods they *add* to the system. The time-stamp in the names makes the changes into objects with identity. If others modify your code, they probably use a new date-stamp also. The end-result looks technical, which is just fitting in component-based software industry.

The changes you make are either redefinitions of existing methods or new methods for existing classes. If possible, use the latter. There can be many different additions simultaneously, but only one current version of a (changed) system method.

Because a carrier's changes are compiled with an explicit message, we can first file-in them all, then use arbitrary programs to activate the relevant ones, in the needed order. A simple procedure would be: before #INSTALLing a carrier, first install its superclass. This allows a subclass to modify the changes of its superclass.

To collect groups of related changes together, we misuse subclassing. A change-set consisting of a class and its subclasses can then be easily filed-out and transferred to other images. This suggests we start collecting changes in a hierarchy like:

```
Carrier940701
    StandardChanges940701
        ObjectChanges940701
        CollectionChanges940701
    ST80Changes940701
        ST80GUIChanges940701
            ST80ViewChanges940701
            ST80ControllerChanges940701
        ST80CompilerChanges940701
    STVmacChanges940701
```

References

1. Beck, K. It's not just the case, THE SMALLTALK REPORT, 3(3), 1993.
2. Lieberherr, K. J., and I. M Holland, Assuring good style for object-oriented programs, IEEE SOFTWARE, 6(5), 1988.

Panu Viljamaa may be reached via internet at panu@ajk.tele.fi.

Listing 1.

Source-code for the class-methods of "Carrier940701," in the "browserformat."

Class definition:

```
Object
    subclass: #Carrier940701
    instanceVariableNames:"
    classVariableNames: '
    poolDictionaries:"
```

Class-methods of Carrier940701:

INSTALL

```
"Compile the source code returned by my methods that carry source-code of
external classes. The methods to call depend on my #systemChanges.
TEST:
    Carrier940701 INSTALL.
    "| changedClass chSelector mySelector sourceToCompile |
    self systemChanges
    do: [:c |
            changedClass  := (c at: 1).
            chSelector := (c at: 2).
            mySelector := self carrySelectorFor: chSelector
                        of: changedClass.
            sourceToCompile  := self perform: mySelector.
            (ReadStream on    :sourceToCompile) fileIn.
            self compileReferenceIn    : changedClass
    ]
```

systemChanges

```
"Return a list of pairs (class selector) that tell the methods whose source-
 code I carry. Note the way a class-method is specified.
TEST:
    Carrier940701 FREEZE"
| aSet |
    aSet := Set new.
        aSet add: (Array with: Object with: #systemChange) "—
            ; add: (Array with: Object class with: #COMMENT)
            ; add: (Array with: AClass with: #aSelector)  —"
        . ^aSet
```

carrySelectorFor: aSelector of: aClass

```
"Return my method-selector that will store the source-code for the method
 aSelector of aClass. The result will be <aClass name>X<selector> but in
 the names of meta classes the spaces are removed and "class" is replaced
 by "Class." "
aClass isMetaClass
 ifTrue: [^(aClass instanceClass name,
            #ClassX , aSelector) asSymbol]
 ifFalse: [^(aClass name ,
            #X , aSelector) asSymbol ]
```

ObjectXsystemChange

```
"Autocompiled by Carrier940701 (4/5/94 10:36 PM). The string below is in
Smalltalk file-out -format."
^'
! Object methods !
systemChange
"Called from places that have something to do with modifying the system to
mark the place as such."
true ifFalse: [ self Carrier940701 ]
"— Reference to the carrier of this source-code.—"
 ! !
'
```


continued on page 16

Introducing VisualAge

Mark Lorenz

Recently, IBM released VisualAge, its own Smalltalk environment and GUI builder. For Smalltalk, this is an event worth noting. So, let's take a look at VisualAge and relate it to the other major Smalltalk vendors' products: ParcPlace's VisualWorks, Digitalk's Smalltalk/V and PARTS, and Easel's Enfin.

WHAT IS VISUALAGE?

VisualAge is a complete Smalltalk environment, with all the corresponding tools including a debugger and browser, plus an instance-based GUI builder with logic connection capabilities. The Smalltalk environment is similar to Smalltalk/V, VisualWorks, and Enfin. The GUI-building is different than any of the others, but closest to the way PARTS works. VisualAge comes in a standalone and team version. The team version has ENVY capabilities built in.

VisualAge lets you develop and utilize visual, nonvisual, and IBM Smalltalk reusable components to create new applications.

The *visual* components include most of the widgets on the Composition Editor palette, such as push buttons and listboxes. It also includes the windows and dialogs that you create.

The *nonvisual* components are the model objects from your business domain that you create. A few nonvisual components are supplied by IBM: the Models folder has OrderedCollection, ObjectFactory (to create instances), and Variable (to hold instances); the Database Functions folder has DatabaseQuery and StoredProcedure.

The IBM Smalltalk components are the same kind of class-based objects that you find in Smalltalk/V, ObjectWorks/Smalltalk and Enfin. You can work at the Smalltalk level and later include IBM Smalltalk components through the Options/Add part... menu item on the Composition Editor.

To visually develop applications, you primarily work with three editors:the Composition Editor, the Script Editor, and the Public Interface Editor.

THE COMPOSITION EDITOR

The Composition Editor is where you compose your visual and nonvisual parts into collaborating pieces of the application puzzle. Figure 1 shows an example application I built following the User's Guide that comes with the product. You see a Road Race window visual component and two nonvisual components: a Runner Factory and a Runner Collection. These were connected interactively to compose a simple application to record the results of a race. This application is composed in the containing visual component RoadRaceView.

Putting the pieces together involves creating instances of the component classes and making service request connections between them. This can be done programmatically or visually. Changes to the class affects all the instances in the application. This is similar to the way PARTS works, but PARTS also allows local instance scripts to be written for that instance only. VisualWorks and WindowBuilder Pro used in conjunction with Smalltalk/V facilitate programmatic connections but don't have visual connection support.

It is possible, certainly, to use multiple visual pieces to create an application. See the section "So how do I put an application together?" for a discussion of how this is done.

THE SCRIPT EDITOR

The Script Editor is where you write IBM Smalltalk code that

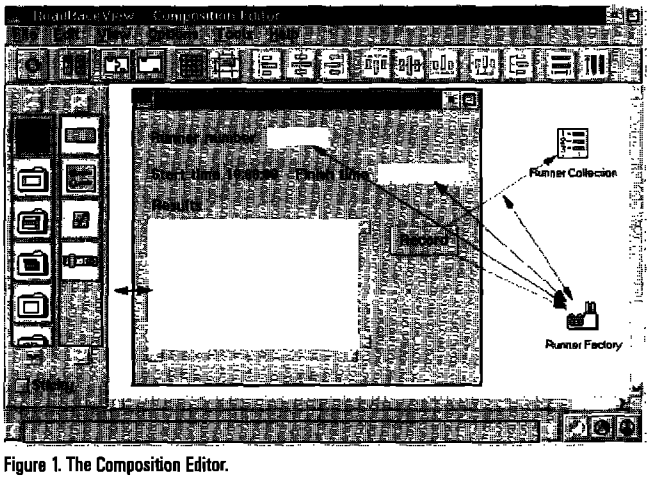


Figure 1. The Composition Editor.

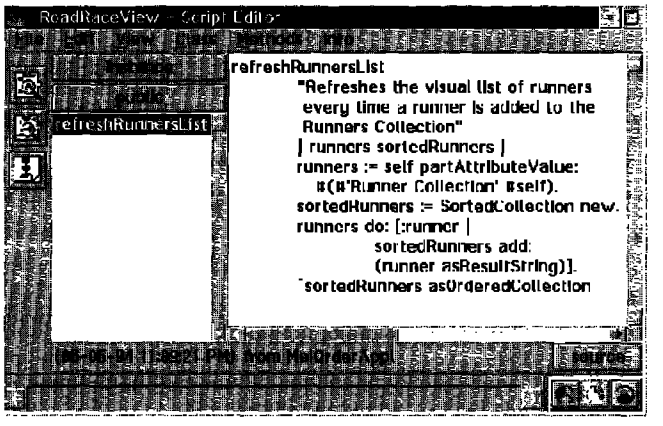


Figure 2. Script Editor

will work with the visual editors. Figure 2 shows an example method from the runner application.

Note the code that reads self partAttributeValue: #(# 'Runner Collection' #self). This gives you a peek into the underlying techniques of working with VisualAge's visual programming tools. Interacting with VisualAge-enabled components involves these indirect references through Dictionary lookups. The editor has capabilities to help you write this VisualAge-enabled logic: Attributes and Actions. These buttons bring up windows that list the components that are a part of the RoadRaceView (see Fig. 3).

Upon selection, the attributes or actions of the selected component are listed. The VisualAge code to get or set the selected attribute or access the selected action is then inserted at the cursor in the method under construction.

So, what's going on here? VisualAge uses dictionaries to look up attributes and actions. In the above example, the RoadRaceView object's components instance variable is a Dictionary with one of its keys equal to Runner Collection (see Fig. 4).

THE PUBLIC INTERFACE EDITOR

The Public Interface Editor defines the attributes, events, and actions that are visible to clients of the component. Figure 5 shows that I have made the refreshRunnersList action a part of the available public services of the RoadRaceView.

Similarly, I can make attributes available to clients and define event notifications as attributes are changed in my component. Each of these appear in the component's connect menu on the Composition Editor.

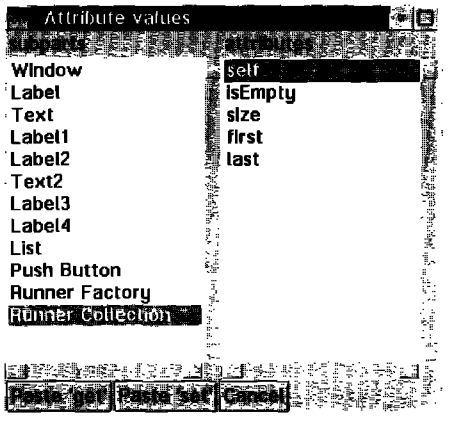


Figure 3. Attribute code lookup.

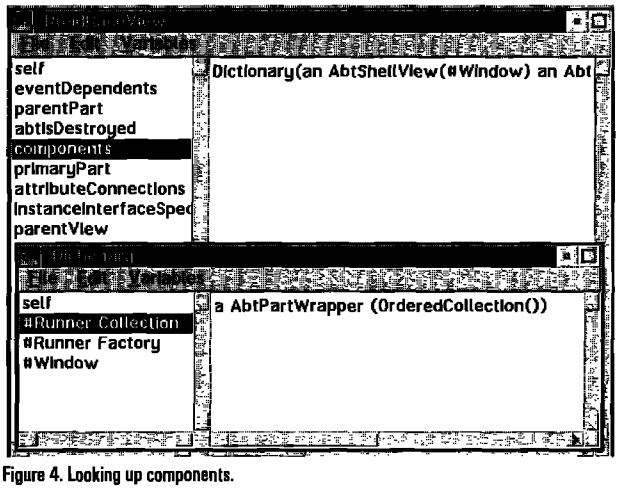


Figure 4. Looking up components.

SO HOW DO I PUT AN APPLICATION TOGETHER?

We've seen the editors available to define and connect visual and nonvisual objects together. One application view object, such as the RoadRaceView, will normally have one window and possibly one or more popup dialogs. An application will normally have multiple windows though. To make these windows work together, we need to make sure that we work with the same instance objects across our view components. We can do this two ways:

- put the instance in a variable from the palette in one view and make that part of the public interface for the view object (from the second view)
- start up the second view with the component as a parameter.

The following code will accomplish this feat:

```
SecondViewClass newPart
    valueOfAttributeNamed: #variableName ifAbsent: [ ^nil ]
    put: ( self partAttributeValue: #( #partName #self ) );
    openWidget.
```

> *VisualAge lets you develop and utilize visual, nonvisual, and IBM Smalltalk reusable components to create new applications.*

CONCLUSION

At this point, the verdict is out on how well VisualAge will do compared to the other Smalltalk environments in the race to meet customers' needs. There are certainly challenges ahead for IBM, which they are working on for the next release. Two major ones are the runtime size, which is larger than it should be for the standalone version (the team version includes a packager that lets you drop in minimum size from 3+MB to 1.3MB according to IBM), and the lack of source code, which is less than the other Smalltalk environments. There are also smaller nuisances, such as the fact that keyboard clipboard actions don't work as advertised

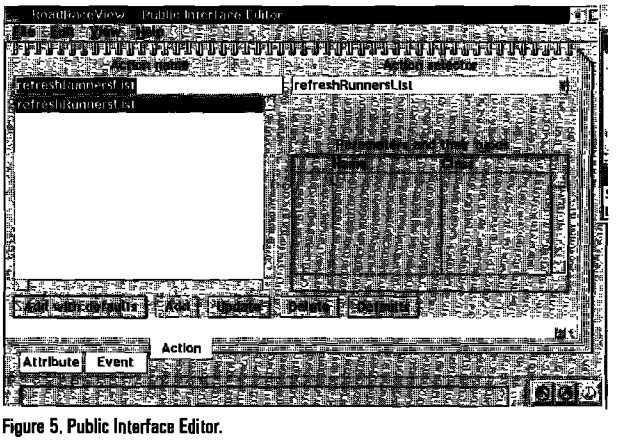


Figure 5. Public Interface Editor.

on the menubar, dialogs that prompt for information don't give you a list to choose from, and global message selector searches list nonmethod symbols with no differentiation.

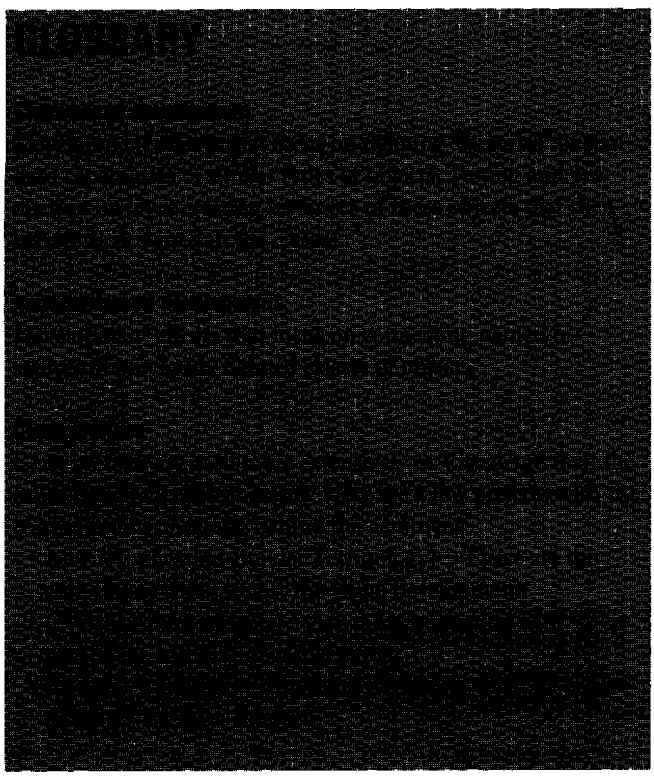
Despite these first release drawbacks, it is possible to do a lot in a little time in VisualAge. The environment has more classes than any of its competitors, including multiple database, communications, and multimedia widgets. IBM is also fostering a components industry around VisualAge, so you can expect to see more and more widgets available for the palettes in the future. I can relate an example of the productivity that can be achieved from a personal consulting effort. I recently worked with a large company in the finance industry using VisualAge. In less than a week, we held some rapid modeling sessions to discover the right objects for their business, and implemented the first scenario script using VisualAge. The scenario involved building a self-validating GUI and interfacing with a SQL Server database.*

Acknowledgement

Thanks to Cynthia McCrickard in the VisualAge support group at IBM Cary for the sample code to pass parameters between view objects as well as for useful review comments.

Mark Lorenz is Founder and President of Hatteras Software, Inc., a company that specializes in helping other companies use object technology effectively. He welcomes questions and comments via e-mail at 71214.3120@compuserve.com or phonemail at 919.851.0993.

* In a future article, I'll present some tips for accessing databases from VisualAge.



continued from page 13

```
compileReferenceIn: aClass
"Compile into aClass a method with the same selector as my name to serve
as a reference back to me."
| sc browserCode |
browserCode := (self name , '
"Autocompiled by ', self name printString, ' ' ,
(Date dateAndTimeNow) printString, '.
By looking at my senders you find the places
modified by it."
true ifFalse: [self Carrier940701
            "The root of it all"].
').
sc := ('
 ! ' , aClass name , ' methods !
 ', (browserCode) , '
 ! !
' ) .
^ (ReadStream on: sc) fileIn
FREEZE
"Produce/autocompile my source-code-returning methods named
<Classname>X<methodName>.
TEST: Carrier940701 FREEZE."
|changedClass changedSource chSelector mySelector carryCode|
self systemChanges
 do: [:c |
            changedClass  := (c at: 1).
            chSelector := (c at: 2).
            changedSource := self codeFor: chSelector
                of: changedClass.
            mySelector := self carrySelectorFor: chSelector
                of: changedClass.
            carryCode  := self myCodeReturning: changedSource
                named: mySelector.
        (ReadStream on: carryCode) fileIn .
    ]

codeFor: aSelector of: aClass
"Return a string of existing source-code in filein-format as it will appear
 inside another filein -format (note the double exclamation marks with no
 space in between them)."
^('
 !! ', aClass name, ' methods !!
 ', (aClass sourceCodeAt: aSelector), '
 !! !!
')

myCodeReturning: aString named: aSelector
"Produce and return source-code for my method aSelector that will return
 aString."
| aComment browserCode |
aComment := '
"Autocompiled by ' , self name, ' ', (Date dateAndTimeNow printString),'.
The string below is in Smalltalk file-out -format."
'.
browserCode := aSelector, aComment, '^', aString printString.
^('
! ', self name, ' class methods !
', browserCode, '
 ! !
')
```

Arbor Help System V2.0

Douglas Camp

Arbor Intelligent Systems has just started shipping version 2.0 of the Arbor Help System (hereafter referred to as AHS), a class library for adding context-sensitive on-line help to VisualWorks applications. AHS is designed to allow anyone (not just developers) to easily create and edit help text in an application under development or in a running application. AHS provides two types of help, a status line facility and a hypertext outline browser.

VERSIONS

AHS 2.0 is available for any platform for which PPS VisualWorks 1.0 is available. Full source code as well as partial source code versions are available. Single image as well as ENVY/Manager versions are available.

DOCUMENTATION

AHS includes a well-written, concise (65-page) manual. The first line of the manual notes that, in keeping with the company's focus on on-line help, the manual was deliberately kept short . In lieu of a large, printed manual, several demo applications are provided. The manual appears sufficient for most purposes, and you can browse the methods in the demonstration and other classes, however many of these classes are sparsely commented. Three months of free support and upgrades are included.

INSTALLATION

Installation is straightforward, especially for an ENVY application. I installed AHS on a 486/25 with 16-M RAM, under Windows 3.1 using ENVY/Manager in single-user mode, in about an hour. Using ENVY slightly complicates the process, however AHS supplies a diskette with the help system in ENVY application format, ready to import into your repository.

After unzipping the AHS application you import it into your ENVY library. Next, file in a .st file containing changes to system methods. This modifies 11 classes scattered throughout the VisualWorksBase and WindowSystem applications. The MenuTracker subclasses for each widget look policy are modified, as well as the controllers for the basic VisualWorks/ObjectWorks widgets. Some of these modifications are very extensive. For example, the MenuTracker>>startUpAt:KeepOpenIfIn: method grows from an already large 33 lines to around 140 lines nested too deeply to understand easily.

USING AHS

AHS provides two types of help—real-time and full outline. Real-time help is a "status line" facility, while full outline help is a hypertext driven text outline browser.

REAL-TIME HELP

Real-time help is very much like the status lines often found in Microsoft Windows applications—when the user points at a widget with the mouse (no mouse click is required), the help text associated with the widget is displayed in a status line area (a VisualWorks subcanvas).

Adding real-time help to a new or existing VisualWorks canvas is very easy. First, make the super class of the canvas AHS HelpApplicationModel (which is a subclass of the standard ApplicationModel). Next, either create a subcanvas of your own in which to display the help text, or you can use a default subcanvas spec supplied with AHS. If you choose to create your own subcanvas, all that is necessary is to add an InputBox with the id #helpText. Optionally, you can add a CheckBox with an id of #helpFlag—if the check box is checked, help will be displayed in the InputBox, otherwise no help text is displayed.

Any widget for which you want to provide help text must have an Id property. The id is part of the key used to lookup the help text. The help text itself is held in a class instance variable of the canvas class. The other element of the key describes the widget state, one of #(#default #on #disabled). This means you can provide the user of an application with specific help text based on the current state of the widget (the #on state is applicable only to radio buttons and check boxes). Any widget, including the menu bar, can have help text. Also, individual elements in the menu bar, or in selection lists, can also have their own help text.

Full outline help

Full outline help provides a hierarchical text outline browser that is activated by typing F1 while the mouse cursor is over a widget.

The full outline help browser is again reminiscent of the help system found in Microsoft Windows. The top pane lists help topics, which can be nested to form outlines. Topics can be collapsed/expanded to show the subtopics they contain. The bottom pane displays the help text associated with the selected topic. Any widget can be linked to an outline topic. Also, the canvas itself can provide a default help topic for all widgets on the canvas. Users navigate through help text in the full outline browser either by selecting topics and subtopics, or by activating *hyperlinks* embedded in the help text.

Navigating through hyperlinks

AHS supports four different kinds of hyperlink: Hypertext links

jump to another topic when clicked. Hypergraphic links open a new window which displays a graphic. Glossary links open a small modal dialog near the linked text that (typically) contains a definition of the linked text. Finally there is something called a *HyperCam* link. The HyperCam link is used in conjunction with another product from AIS called ArborCam. ArborCam is a tool for capturing and displaying "screen movies." The idea is that when a user clicks a HyperCam link, a screen movie is displayed, perhaps demonstrating some action in detail. We don't have ArborCam, and so can't comment on its usability with AHS.

The implementation of hyperlinks in AHS is very nice. Distinct visual feedback is provided for each kind of link (e.g., hypertext appears in green and is underlined, the cursor also changes to a hand icon when over the link). Creating a hyperlink is a simple point-and-shoot operation: while in edit mode, you simply type text into the bottom pane of the full outline browser, select the text to be hyperlinked, then select the type of link from the operate menu.

Creating help text—The help editor

Help text is entered and edited for real-time help, and links to outlines established for full outline help, via the help editor. One of the most interesting features of the AHS is that the help editor can be activated, and new help text or outlines created, either while the canvas class is under development (i.e., open in the UIPainter) or while the application is actually running. This means that developers can focus on application development and allow other team members (technical writers, prototype users, interface designers, etc.), who perhaps don't even know Smalltalk, to add help to a VisualWorks application. Being able to add help text to an application while it's running is a very natural and productive way to develop on-line help. As you use the application, places in the process that may require help text are readily identified, and it's easier to develop that text as you interact with the running application. In a sense, AHS extends the flexibility, immediate feedback, and ease of use of the Smalltalk environment itself to the task of creating on-line help.

When an application and its help text are complete, the ability to modify help text can be turned off in the run-time image.

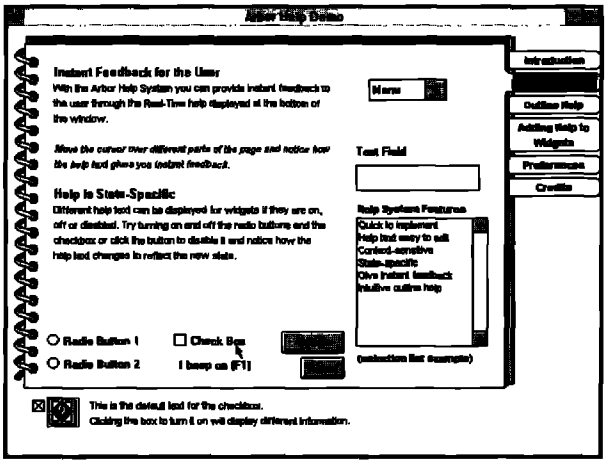


Figure 1. Real-time help.

MEMORY CONSIDERATIONS

Help text can be stored in the image (real-time help only), in a flat file (full outline help only), or in a Versant or Gemstone object database. If your application included lots of real-time help text that was stored in the image the memory requirements could become significant. In this case, AHS provides behavior that the developer can use to selectively load/unload portions of the help text dictionary into memory on demand. There is also some provision for minimizing the memory footprint of deployed applications containing AHS. A list of classes that can safely be stripped from a runtime image is included.

PRICING

The full source version of AHS sells for $1395. Each license entitles the user to use AHS on one system at a time, and to distribute applications developed using AHS free of runtime fees (except of course the PPS VisualWorks runtime fee) provided the ability to edit help text within the running application has been turned off.

OTHER FEATURES

AHS contains several other interesting features that are new in version 2.0: Internationalization of help text is supported. For example, you can implement all help text in Spanish and English,

continued on page 28

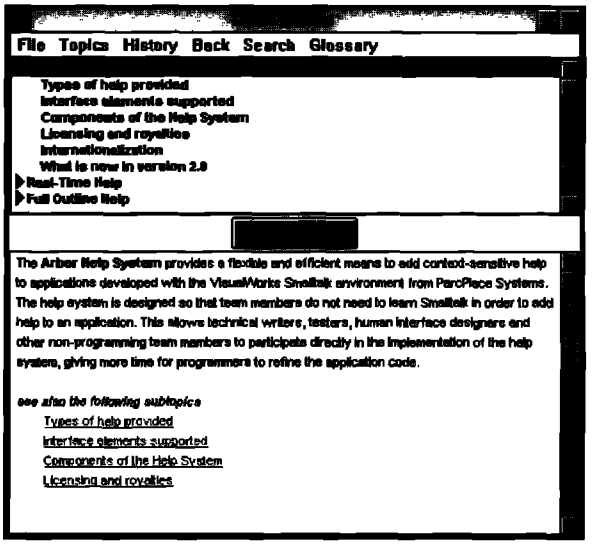


Figure 2. Full outline help.

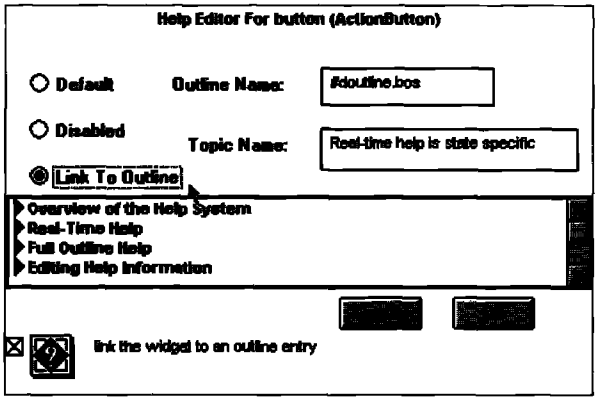
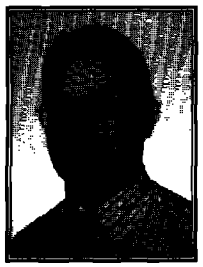


Figure 3. The help editor.



Performance tips

ALAN KNIGHT

This is the first of a two-part series of articles on optimizing performance in Smalltalk. It's very easy to write an inefficient Smalltalk program, and easy to blame the language for the resulting inefficiency rather than spend a little time tuning it. While optimizing performance is a large and complex topic, I hope this can present a few general principles and hints for those seeking to improve the response time of their applications. The discussion from which most of these posts are taken started with a question from **Bill Punch (punch@cps.msu.edu)** who writes:

> **I don't want to start an argument about why Smalltalk is slow/fast or other languages are slower/faster. My question is this: If you write a ST program, and it isn't as fast as you would have liked, what tips, tricks-of-the-trade, advice etc. can the ST community offer to make systems go faster? We have written a small-medium size ST program (12,000 lines, if that means anything) that runs a bit slowly. We rewrote it, started caching results, and using other "algorithmic" approaches to get more speed (and shrunk the code at the same time), but it is still a bit slow. We'd like to go faster but we've exhausted our "algorithmic" approaches. Are there things in Smalltalk we should do/avoid to get improved performance?**

BASIC PRINCIPLES

The most important principle of performance optimization is to optimize in the right places. It only makes sense to tune the code where it will make a difference to the overall performance of your application. In most applications, a great deal of the running time is spent in a relatively small amount of code. Exactly which code depends enormously on your application, and to a lesser extent on the version of Smalltalk and the operating system.

Use a Profiler

How do you find the performance-critical sections in your application? Simple. Use a profiler. These are available from a variety of sources. As far as I know, they all work in roughly the same way. You provide a block of Smalltalk code, which is run in a separate process. The profiler runs another process, and periodically interrupts the process being profiled, taking a snapshot of the stack. Most can show either the calling sequence or the total time spent in a particular method. For the most part, they are quite simple to use. **Jan Steinman (jan.bytesmiths@acm.org)** provides a couple of tips.

> **It's a sampling technique, so it is necessary to make certain you have enough samples, or else you end up with aliasing artifacts. Make sure you loop *lots* of times in the profile. Be aware that lengthy primitives may "pile up" the time on either side of a sample. In general, the bottom few lines should show most of the time in primitives, or at least most of the time should be in base methods. If not, you have a target for tuning.**

For a good introduction to the uses of profilers (and lots of other great articles, on performance tuning and programming in general) see Jon Bentley's books PROGRAMMING PEARLS and MORE PROGRAMMING PEARLS (Addison-Wesley, ISBN 0-201-10331-1, and 0-201-11889-0 respectively).

Don't over-optimize

Performance optimization is a good thing, but too much of a good thing can be bad for you in the long run. Some kinds of optimizations are almost always good because they are the kind of things that we normally strive for in programming. The algorithmic improvements described in the original question usually fall into this category because they also can make the code cleaner and simpler. On the other hand, some optimizations reduce encapsulation, hinder maintenance, and generally make you do things you would otherwise consider bad practice. They are still worth using, but should be employed carefully, with the realization that you are trading maintainability for speed. They should definitely be avoided early in the development process. Many of the most powerful optimizations come from exploiting knowledge of the application's structure. If we are guaranteed that some circumstance cannot possibly occur, we can often make dramatic simplifications and speed optimizations by exploiting that knowledge. The danger here is that next month's addition to the list of requirements may be the ability to handle exactly that circumstance.

Avoid recomputing

Many applications compute the same values repeatedly. A very common optimization is to cache values that are repeatedly computed, trading space for speed. This can be implemented by storing dictionaries of values in a class variable. This is particularly valuable when there are relatively few values that are expensive to compute.

Know what's expensive

Smalltalk allows you to program at a high level of abstraction. It's easy to write code quickly, using the most convenient operations rather than the most efficient. For performance optimization it's vital to understand the costs and trade-offs of different operations. This detailed knowledge of the standard class library

Alan Knight is a consultant with The Object People. He can be reached at 613.225.8812, or by e-mail as knight@acm.org.

and how to achieve the best performance with it is an important part of the toolkit of expert Smalltalk programmers.

Collections

The Collection hierarchy is widely used in almost all programs, and wise use of it is vital to writing fast Smalltalk code. There are several issues that can often snag beginners.

Growth

The most basic collection in Smalltalk is an array. It's fixed-size and is accessed by indexing. It's also the fastest collection to access. Fixed sizes are inconvenient, though, and it's much more common to use OrderedCollection, which supports a number of additional operations. The most important is add:, which appends an object to the collection, growing the collection to accommodate it if necessary. This growing process is surprisingly efficient. Peter Epstein (peter@objectime.on.ca) writes:

> **Adding an item at the end of an OrderedCollection is amortized O(1) (in other words, adding n items at the end requires O(n) time) since no shifting is needed and growing is done by a percentage rather than a constant number of slots.**

For those unfamiliar with terms like "amortized O(1)," here's a brief explanation. OrderedCollections are ordinarily implemented using an array that is at least as large as the number of elements in the collection. There may be additional space, in anticipation of elements being added later. Adding one element to an OrderedCollection that is full may require quite a bit of work (it allocates a new, larger collection and then copies all of the elements into it) but this expensive operation is guaranteed to happen rarely. Since the number of extra spaces allocated is proportional to the size of the collection, over a sequence of operations the average cost of each is still small, regardless of the size of the collection.

Even though adding to OrderedCollections is relatively efficient, it still has a cost. Jan Steinman writes:

> **If a collection never needs to grow, make it an Array. (Do the same thing if it grows to a certain point, then becomes essentially read-only.) If it *rarely* needs to grow, make it an Array anyway, and copy it each time, or consider streaming it.**

If a collection does need to grow, it's often useful to guess at its size. The default size of an OrderedCollection is usually around a dozen elements. If you have a pretty good idea that a collection will hold hundreds of elements, use OrderedCollection new: 500 rather than OrderedCollection new. If you're right, you'll save several grow operations, and if you're wrong you only waste a bit of space. Overestimating may also be a good idea if you're not quite sure what the system code does.

This becomes even more important with other kinds of collections. When an OrderedCollection grows, it only needs to copy a bunch of object pointers. Sets and Dictionaries are hashed, and must rehash all of their elements when they grow.

In extreme cases, growing collections can be a major factor in application performance. Andy Choi (andyc@hprnlac.rose.hp.com) writes:

> **Using the VisualWorks APOK time profiler, we've found that in one of our applications, up to 70% of the time was spent adding elements to collections... We've found that in general, growing collections is one of the most time consuming parts of VW.**

Use the right collection for the job

There are many collection classes to choose from, and the choice can affect performance substantially. We've seen that OrderedCollections are reasonably efficient using operations like at:, at:put:, and addLast:. There are many other operations that are not at all efficient, requiring a linear search or moving many of the elements in the collection. These are operations like addFirst:, indexOf:, add:before:, after:, and removeIndex:. If you find yourself using operations like this on large collections, consider changing your representation.

SortedCollections are like OrderedCollection, but maintain their elements in sorted order. They therefore behave much like OrderedCollections except that it is very slow to add objects (since they must re-sort). If you must add objects to a SortedCollection, try to group them together and use addAll:, which adds several elements but does only one sort.

Sets, Bags, and Dictionaries are collections implemented as hash tables, and as such don't maintain an order for their elements. They are only a little less efficient for iterating than sequenceable collections but are much more efficient for inserting, deleting and testing membership. IdentityDictionaries (and IdentitySets) can be used for greater efficiency in some situations, but you must be careful that an identity-based collection is really what you want. It's also important to ensure that the objects used as keys have a good hash function. These issues, and others associated with hashed collections were discussed in detail in The Smalltalk Report, 2(8).

It's not necessary to confine yourself to the collections available in the base image. A radically different collection implementation may be ideal for your application, making it worth the extra effort of finding an implementation (hint: refer to some of my previous columns on freely available "goodies") or of implementing it yourself. This is particularly likely to be useful if you have very large collections or very sophisticated operations. David Siegel (dsiegel@panix.com) writes:

> **Another approach is to build data structures that grow incrementally (like self-balancing trees) rather than Dictionaries. This approach is a win for large data structures (I've seen crossover between 5,000 and 50,000 elements, depending on the problem and the data structure).**

If you don't need to iterate, the best solution may be not to use an explicit collection at all. Eliot Miranda (eliot@ircam.fr) writes:

> **Another approach might be to make membership a property of the objects themselves, rather than implement it by placing them in a collection. For example, posit you have a set of objects S, and wish to implement subsets S_a and S_b, so that an object in S can be in either S_a or S_b, in both S_a and S_b or neither. If it's possible to add an instance variable to all objects in S then membership of S_a or S_b could be recorded in this instance variable. The implementation can be hidden behind an interface such as $isInS_a$, $isInS_b$, $joinS_a$, $leaveS_b$, etc.**

Objekt-orientiertes Programmieren

OOP '95

MÜNCHEN

Don't Miss Germany's Most Attended Object Technology & C++ Conference!

Moving Forward with Object Technology

Here's What Prominent German Publications Said About OOP'94...

*"**OOP '94**: A lot of new exhibitors with new products..."*

—*Computerwoche*, Nr. 7, February 18, 1994

"For companies who want to give further education to their employees by participating in a conference or who are making important buying decisions...it is the only alternative."

—*iX Magazin*, March, 1994

"The most important thing for the attendees was the quantity and the quality of the talks and the seminars...It was a positive experience in contrast to mass-events like Cebit... For software developers it is recommended to visit this conference."

—*mc Magazin fur Computerpraxis;* Jan. 1994

FOR INFORMATION ON EXHIBITING OR ATTENDING OOP'95 FEATURING C++ World CONTACT:

(In the USA) SIGS Conferences, Inc.....v. 212.242.7515...f. 212.242.7578

(In Germany) SIGS Conferences GmbH.....v. 089.957.9517...f. 089.957.9125

Sponsored by:

Journal of Object-Oriented Programming

C++ Report

OBJEKTspektrum

Object Magazine

iX Multiuser Multitasking Magazin

Presented by:

SIGS Conferences

Concatenation

Of course, collections don't grow unless things are being added to them. If we can reduce the number of unnecessary appends we also will get significant savings. One major source of unnecessary collection operations is copying collections, particularly strings. The biggest culprit is the comma operator "," which concatenates two strings. It's very common to see code like:

```
Transcript show: 'My favorite variable = ', myFavoriteVariable printString,
    'at step ', stepNumber printString; cr.
```

There's nothing inherently wrong with this kind of code. It does what it's supposed to, it's easy to write, easy to understand, and debugging code doesn't have to run fast anyway. However, if your application uses this type of code a lot, you should be aware what's going on to execute this. First of all, printString is implemented as something like:

```
|aStream |
aStream := WriteStream on: String new.
self printOn: aStream.
^aStream contents.
```

For each call to printString, we're creating a stream, printing the object on it, then copying the contents and discarding the stream. The stream will almost certainly have to grow the string at least once.

So, to execute the previous debugging statement we create and discard two streams, copying their contents and then performing three successive concatenations, each of which allocates a new string and copies both of its arguments into it.

For small strings, and small numbers of strings this inefficiency is not a big problem, and may be worth it for the convenience of the code. If necessary, however, this can be written much more efficiently (if less compactly) as

```
| aStream |
aStream := WriteStream on: (String new: 100).
aStream nextPutAll: 'My favorite variable = '.
myFavoriteVariable printOn: aStream.
aStream nextPutAll: ' at step '.
stepNumber printOn: aStream.
Transcript show: aStream contents; cr.
```

It's also possible to improve on this if you're willing to go out of the realm of reasonable code transformations and into efficiency hacks. The Stream>>contents message normally copies its contents. If you know you're not going to modify the contents, you may be able to access the collection being streamed over directly,

> *"We've found that in one of our applications up to 70% of the time was spent adding elements to collections."*

saving a copy. If you're writing to the Transcript, you can treat it as a stream and use nextPutAll: repeatedly (in some versions you have to add a few extra methods in order to treat the Transcript as a WriteStream). Only do a show: on the last operation, since that will force a screen update, which is much slower than any reasonable number of string operations.

On the general subject of streaming, Jan Steinman writes:

> **Streaming is more efficient the larger the number of items you want to concatenate. My rule of thumb is that I use streams whenever I need to concatenate three or more Strings, especially if I know about how big the result will be, or if any of them needs to be sent #printString. At least in PPS ST-80, #nextPut: is a primitive, and #nextPutAll: is little or no more expensive than an ordinary copy, leading one to intuit that two concatenated Strings is close to break-even, and any over that is a win for streaming.**
>
> **A big win is to pre-allocate your stream. If you know your TextView will have about a thousand characters, using**
>
> **(String new: 1000)**
>
> **writeStream means you have a grand total of two copies per Character, no matter how many little bits and pieces you're putting together: once to put them on the Stream, once to take them off. In general, try to make it slightly bigger than the most likely result size.**

That's all for collections. In the next column, I'll look at some hints regarding blocks, numbers, special classes and messages, graphics operations and some miscellaneous tips. ♀

KENT BECK

Using patterns: Finishing the design

Before I begin, I'd like to comment on the code in the June issue by Bill Cole and Tim Howard. I loved the simplifications to VisualWorks' ApplicationModel component protocol. I use them now whenever I can. I did have a bone to pick with their programming style, though.

I didn't like it that many of their methods would take either a Symbol or a Collection of Symbols as arguments. There are at least two conflicting forces at work here. The first is the desire for simplicity, to add as few selectors as possible. I certainly appreciate simplicity. I have too many selectors memorized as it is. However, a method that takes either one or many arguments is a violation of a more important principle-clarity of expression. While their simple examples of usage weren't confusing, down the road I think it is much clearer all around to have two flavors of the message, one that takes a single argument and one that takes a collection. Here is the kind of thing that has happened to me in similar situations.

First I start out with code like this:

```
...
self hide: #okay.
...
```

Then I realize I have the same symbol in many different methods, so I factor this into two methods:

```
...
self hide: self hiddenButtons.
...
hiddenButtons
    ^#okay
```

Later on, I find self hide: self hiddenButtons and say, "Oh, great. I can iterate through all the hidden buttons." So I write:

```
self hiddenButtons do: [:each | ...]
```

Boom! Not only does my code not work as expected, the error occurs somewhere deep inside the block, because Symbols respond to do: just fine, but the loop variable each gets set to a Character instead of a Symbol.

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

As I said, I prefer to have two variants of the message, in this case hide: aSymbol and hideAll: aCollection. This strategy gains in clarity, but doesn't it lose simplicity? Not really, because I know that if I have a message, foo:, which takes a single argument, if I have a version that takes multiple arguments it is always called fooAll:. I don't have to remember two messages, just one message and one rule that applies to all messages.

I don't claim any originality in this. I'm just copying the collection protocol, which has add: and addAll:. I have found that applying the same pattern consistently leaves me with code that is easy to read and use, even months or years after I first wrote it.

INTRODUCTION

In the previous column we began exploring what it would be like to use patterns. I mentioned three ways I've found to use patterns: as documentation for designers, as documentation for reusers, and as a design aid. I started presenting an example intended to simulate the use of patterns in design.

The example problem is to design the objects to run a television with a remote control. We had applied three patterns—Objects from the User's World, Objectified Library, and Event, to come up with five objects:

Television
- change channels

RemoteControl
- translate user input into commands
- read keyboard events

Keyboard
- create events from keystrokes

Event

To change a channel, the Keyboard creates an Event, which is read by the RemoteControl and interpreted to send a message to change the channel to the Television.

In this column we'll finish the design by showing how the objects get divided between the address space in the remote control's processor and the address space in the television set.

HALF OBJECT

There is one common approach to giving an object a presence in more than one address space. I call it *Remote Proxy*. In it, you put the full object in one address space, and you put a minimal object in all the others. The minimal object, the remote proxy, only has enough information to forward any message it receives across the network to the real object. This pattern relies on most of the processing being done in one address space, and only requests for processing originating from other address spaces.

In our example, the object that I want to make accessible in both address spaces is the RemoteControl (you might also distribute the Television, but there are other good reasons we'll get into later for not distributing the Television). Looking at its responsibilities, one of them, "read keyboard events," has to take place in the remote control. The other, "translate user input into commands," needs to take place in the television so the commands can be executed. We cannot use Remote Proxy to distrib-

ute the RemoteControl, because the processing is balanced between the two address spaces.

Here is a second pattern for distributing objects I learned on a telecom consulting assignment:

Half Object
How can you distribute an object that has substantial processing responsibilities in more than one address space?

The usual pattern for distributing objects is Remote Proxy, which puts a minimal stub object, the proxy, in all but one of the address spaces. Messages to the stubs are forwarded to the "real" object. This works well for "client/server"-style interactions, where there is little dialog between the sender of the message and the receiver. When the object maintains an active role in more than address space, though, it leads to excessive communication overhead. No matter where you put the object, many of its messages are remote.

Another case where Remote Proxy is not appropriate is where part of the functionality of an object is implemented in a system that you do not control. To maintain a legacy, you may not be able to concentrate all of the processing in a single address space.

Giving an object a presence in more than one address space leads to its own set of problems. You must maintain the object's identity, even across communication failures or system recovery. You must keep the parts of object synchronized if there is duplicate information. You must also design the split between the parts of the object to minimize long-distance communications.

Even with these drawbacks, where necessary, splitting an object across address spaces is a reasonable design decision. Therefore:

Split one conceptual object across address spaces. For symmetrical splits (where the same code is running in all address spaces), prepend "Half" to the object's name. For asymmetrical splits, prepend a word describing the address space to each objects' name.

Once you have divided the object, you will need to design a protocol to go between the parts.

Here's how we'll use Half Object on RemoteControl. We put the responsibility for reading keyboard events in remote control, and call it RemoteRemoteControl. We put the responsibility for mapping input to commands in the television, and call it TelevisionRemoteControl. As is often the case with names generated blindly (or it may be a weakness in the above pattern), these names don't read well. We'll simplify them to RemoteControl and TelevisionControl.

RemoteControl
- read keyboard events

TelevisionControl
- map user input to commands

DESIGNING THE PROTOCOL

Notice that Half Object has a little coda that reminds us that we need to design a protocol to go between the two halves of the RemoteControl. Many patterns don't solve a whole problem by themselves. Solving one problem leads to others, and so on until you have running code (heck, until the product finally dies). How do we design the protocol?

The problem statement in the last issue specified that we were given a library for infrared communications. We can use Objectified Library to make this into an object, InfraredStream.

InfraredStream
- read and write bytes

This takes care of how the bytes will get transmitted, but there is still a big gap between "the two halves of the object need a protocol" and "read and write bytes." There are two parts to this question: what gets sent between the RemoteControl and the TelevisionControl, and how does it get formatted. Fortunately, the answer to the first question is easy- Events. Once a TelevisionControl gets ahold of an Event, it knows what to do. Now we're left with the problem of how to format Events. What we need is an object whose responsibility is to read and write Events, bridging the gap between what the RemoteControl and TelevisionControl need, and what the InfraredStream provides. Here is a pattern I use fairly often for this:

Formatting Stream

How should you translate objects to and from lower level representations, like characters or bytes?

One simple solution is to have the object that needs to read or write another object do the formatting. This has the advantage of simplicity, as you don't need to introduce any new objects and you don't need to add to the behavior of the object being formatted. However, this means that the object doing the formatting has to know the details of the object being formatted, making it less flexible. Also, if one object wants to write and a different kind of object wants to read, you are stuck either with two copies of the formatting code to synchronize, or you have to waste inheritance on sharing the formatting code.

Another solution, more idiomatically object, is to have the object being read and written do the formatting. This insulates the reading and writing objects from all the details of what object is being read or written, and centralizes the formatting code. However, you may not have authority to change the object being read and written, or you may want to have multiple formats. Therefore:

Create a stream for the objects you wish to read and write. The name of the stream is the kind of object being written concatenated with Stream. The external protocol is next for reading the next object, nextPut: anObject for writing an object, and atEnd to test whether there are any objects for reading. Create the stream on a lower level stream.

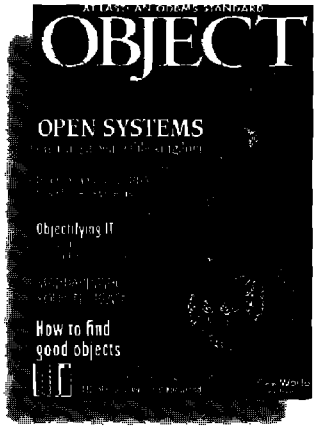
Using Formatting Stream on our problem causes us to create EventStream, which we will create on an InfraredStream. This gives us the flexibility to change the transport mechanism (EthernetTV?), or the format of the events without affecting either the RemoteControl or the TelevisionControl.

Tying it together

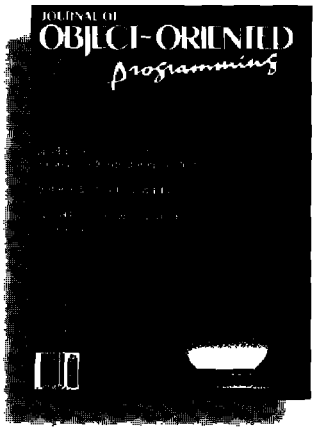
Let's follow a scenario all the way through from the user pushing a button on the remote control to the channel changing.

continued on page 28

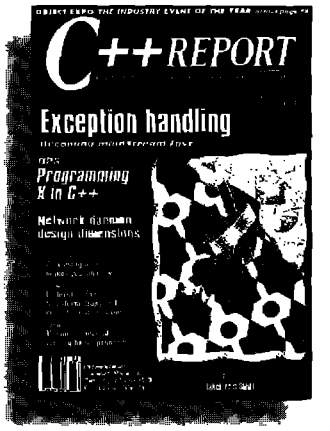
Your best tools for object-oriented software development...



The manager's guide to implementing object technology. The "point of entry" for software management infusing objects into their work environment. Filled with how-to advice, usable strategies, and real world experiences.

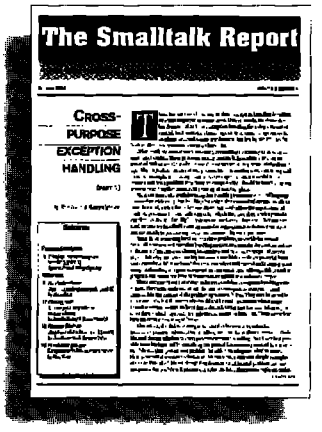


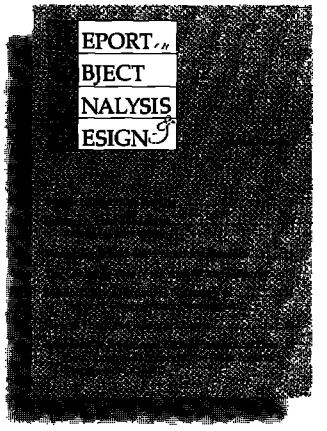
Written for programmers and developers using OOP techniques. International in scope. Code intensive, practical, technical. Breakthrough peer-reviewed papers and invited columns. Now in its 7th year.



Informs C++ developers on how to get the most out of the language. Ideas and techniques for increasing your productivity with C++. Code-intensive, functional tips and tricks for C++ users on all levels and platforms.

Filled with "how-to" advice for Smalltalk users at all levels and in all dialects. The best way for Smalltalk programmers to maximize the language's potential.





Addresses language-independent, architectural concerns about O-O analysis, design and modeling. Platform and system independent, ROAD is written for software developers and project leaders.

SIGS PUBLICATIONS

Yes, I want my subscription to the following publications to begin immediately. If not completely satisfied, I may cancel at any time and receive a full refund of the unused portion.

- □ *Object Magazine* (1 year, 9 issues) $39
- □ *JOOP* (1 year, 9 issues) $59
- □ *C++ Report* (1 year, 9 issues) $69
- □ *The Smalltalk Report* (1 year, 9 issues) $79
- □ *ROAD* (1 year, 6 issues) $99

TOTAL _____

Method of Payment:

□ Bill me, Attn: _____

□ Check Enclosed (Payable to *SIGS Publications*)

□ Charge My: □ Visa □ MasterCard □ American Express

Card# _____ Exp ____

Signature _____

Name _____

Title _____

Company _____

Address _____

City _____

Province/State _____ Postal Code/Zip _____

Country _____

Phone _____ Fax _____

Return this coupon by mail or fax, or call to start your subscriptions.

Mail: SIGS Publications, Inc., P.O. Box 2027, Langhorne, PA 19047

Fax: 215-785-6073

Phone: 215-785-5996

Important: Non-U.S. orders must be prepaid. Please add $35 per subscription year for air service. Checks must be in U.S. dollars drawn on a U.S. bank.

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 THE SMALLTALK REPORT, Product Announcements Dept., 885 Meadowlands Dr. #509, Ottawa, ON K2C 3N2, Canada, 613.225.8812 (v), 613.225.5943 (f).

KnowledgeWare and Digitalk Ship PARTS Wrapper for ADW

KnowledgeWare Inc. and Digitalk Inc. have announced that they have begun shipping a client/server link between KnowledgeWare's Application Development Workbench (ADW) and Digitalk's PARTS.

The PARTS Wrapper for ADW makes the ADW/Planning and Analysis models of business requirements available as components for quick assembly into applications, which is done using Digitalk's object-oriented PARTS assembly and reuse toolset. Developers can reuse and combine models of business requirements developed with ADW, and assemble them with other new or legacy application components into enterprise applications.

The PARTS Wrapper for ADW creates Smalltalk classes that relate to the entities in ADW business models, and methods that implement the attributes and relationships of those entities. The resulting Smalltalk objects are used in PARTS Workbench to create fully functional client/server applications based on business requirements identified in ADW. Also, methods contained in SQL statements for database access are automatically created from information contained in ADW data models. These methods are automatically maintained as the business model changes. Automatic SQL generation and maintenance eliminates hours of manual effort for developers.

The PARTS Workbench also allows developers to combine the reusable objects created from ADW/Planning and Analysis models and assemble them with other new or legacy application components for enterprise applications.

KnowledgeWare, 404.231.3510, ext. 235 (v).
Digitalk, 5 Hutton Center Dr., 11th floor, Santa Ana, CA 92707, 714.513.3000 (v), 714.513.3100 (f).

Servio Announces GemStone Version 4.0

Servio Corporation, a major supplier of advanced database and development technology, has announced GemStone Version 4.0 and in the process set the stage for deployment of large-scale commercial applications based on object technology.

In developing GemStone Version 4.0, Servio worked closely with its major customers to determine their application requirements and to incorporate those requirements into GemStone. It provides a transparent link to the Smalltalk programming language coupled with an underlying ODBMS architecture that meets corporate MIS's definition of a "production quality" multiuser database management system for heterogeneous distributed computing environments. GemStone Version 4.0 also supports applications written in C or C++ and has tools for integrating legacy data.

The major enhancements in GemStone Version 4.0 include the addition of a shared memory architecture, transaction processing and I/O optimization, improved concurrency, improved operational support for 24-hours-a-day, seven-days-per week operations, and new facilities for administration and tuning of GemStone databases in production environments.

Servio Corp., 408.879.6214 (v).

Softlab and Digitalk Integrate Product

Softlab and Digitalk, Inc., announced a long-term partnership agreement under which Softlab will provide a comprehensive redevelopment solution by integrating Digitalk's PARTS into Softlab's Maestro II product line. Softlab also has signed a worldwide non-exclusive agreement to market and distribute PARTS along with its own Maestro II products.

By combining strengths, Softlab and Digitalk are able to provide a migration path to bring legacy systems to the desktop in an integrated LAN workgroup environment. The combination of products allows users to identify reusable components of existing legacy systems using Maestro II and then use Digitalk's PARTS Wrapper technology to create GUI-based desktop applications.

A product of the relationship is an integrated link, which will be called the PARTS Wrapper for Maestro II. It will enable developers to identify reusable components of mainframe applications that can be migrated to client/server environments in Digitalk's fully object-oriented PARTS Workbench. This automatic mapping eliminates hours of manual effort for developers and results in a rapid, visual development of object-oriented applications. By utilizing PARTS Wrapper technology to integrate Maestro II and PARTS , organizations now can have a smooth migration path from mainframes to object-oriented client/server architectures.

Softlab's Maestro II solution is an application maintenance and redevelopment environment that allows large teams of users to easily maintain mainframe legacy systems and identify existing mainframe components that are candidates for reuse in client/server or other architectures.

Softlab, 404.668.8856 (v).
Digitalk, 5 Hutton Center Dr., 11th floor, Santa Ana, CA 92707, 714.513.3000 (v), 714.513.3100 (f).

ParcPlace Announces VisualWorks ReportWriter

ParcPlace Systems, Inc. has announced VisualWorks ReportWriter release 1.0, an extension to its VisualWorks product line. ReportWriter is a client–server database reporting tool that allows corporate developers to visually create sophisticated reports without Smalltalk or database programming. Reports can be built using a point–click approach, then deployed on Windows, OS/2, Macintosh, and UNIX platforms.

ParcPlace's visual layout approach let's you create simple and complex reports while providing developers with full control over WYSIWYG report presentation. VisualWorks ReportWriter also provides MIS departments with concurrent access to a wide range of data sources including SQL relational databases like Sybase and Oracle.

ParcPlace licensed VisualWorks ReportWriter from Synergenics Solutions, Inc. ReportWriter was developed by Synergenics Solutions using ParcPlace's VisualWorks client/server development tool.

ParcPlace Systems, Inc., 408.720.7514 (v) snichols@parcplace.com (email)

Recruitment

For
information
on
advertising in the
Recruitment Section,

contact
Michael W. Peck

at 212.242.7447

Smalltalk

Programmer Analysts

Electronic Data Systems Corporation, the world leader in applying information technology, currently has opportunities in New York, London and Tokyo for experienced OOP professionals. Qualified Programmer Analyst candidates must have hands-on experience with Smalltalk. Experience with UNIX and exposure to the Securities Industry preferred.

EDS offers salaries commensurate with experience and excellent benefits. If you are interested in further developing your professional skills working with an industry leader, mail your resume to:

EDS Staffing, Dept. 2410
1251 Avenue of the Americas
41st Floor
New York, NY 10020

EDS

EDS is an equal opportunity employer, m/f/d/v.
EDS is a registered mark of Electronic Data Systems Corporation.

The American Funds Group®

The American Funds Group is one of the most successful mutual fund organizations in the world. Since 1931, we have provided our shareholders with consistently superior investment results and outstanding service. Share in the continued growth of our Norfolk, VA Office.

We have been a financial industry leader in Smalltalk development for over 5 years. We are currently developing a large client server based customer service system. This application is being created using the latest object oriented methods and is in the beginning stages of development. Ideal candidates will have the opportunity to be a part of the design team whose responsibilities will include these initial phases of development.

We offer a competitive salary and excellent benefits package including:

- Medical, dental and vision care coverage
- Educational assistance
- An outstanding company-paid retirement plan

Positions are currently available for:

Senior Smalltalk Programmer

This positon requires 2 to 5 years of Smalltalk experience including OOA and OOD. Job responsibilities will include leading in the overall design and creation of class and object hierarchies.

Object Oriented Supervisor

In this position you will supervise a team of 5 to 6 Smalltalk developers. Five years experience managing client server development is required for this position. Ideal candidates must have at least one year of object oriented experience including OOA and OOD methodologies.

Smalltalk Programmer

In this position, you will develop GUI based client server applications. At least one year of Smalltalk experience is required.

If you are interested in applying for any of the positions listed above, please send your resume and salary history to:

The American Funds Group
(Please specify position)
5300 Robin Hood Road
Norfolk, Virginia 23513

EQUAL OPPORTUNITY EMPLOYER

Product Report

continued from page 18

then allow the user to switch between languages throughout the application with one button click. Many of the AHS classes (e.g. scrollable input fields, text outline classes, notebook widgets) have been redesigned so that they can be more readily reused in your own applications. Also, the full outline browser has been enhanced to support glossaries, a history list, and text searches.

Summary

The AHS is a well designed, very useful framework for adding online help to VisualWorks applications. The real-time and full outline help facilities integrate well into the VisualWorks environment, and the ability to easily allow nondevelopers to create and edit help text in a running application is very nice. For a demo disk or further information on the AHS contact Robert Royce at Arbor Intelligent Systems, 313.996.4238, or roycer@umcc.umich.edu. ♀

Acknowledgments

The author would like to thank Sivaram Hariharan of Knowledge Systems Corporation for assistance with this report.

Doug Camp is a Senior Member of the Development Services Staff at Knowledge Systems Corporation. He can be reached at 919.677.1116 or dcamp@ksccary.com.

Smalltalk Idioms

continued from page 24

In the remote control:

1. the user presses a key
2. the Keyboard creates an Event for the key press
3. the RemoteControl reads the Event, and write it on its EventStream
4. the EventStream writes the Event on the InfraredStream
5. the InfraredStream transmits the bytes

Then in the television:

6. the InfraredStream reads the bytes
7. the EventStream reads the bytes and creates an Event
8. the TelevisionControl reads the Event, decodes it, and sends the "channel: anInteger" message to the Television
9. the channel changes

This design still leaves many decisions fuzzy. Exactly what control structures are being used to read and write? Polling or interrupt driven? How would the system work with a more complicated, realistic keyboard that has many different kinds of keys?

When I have a design this close, I like to build a prototype that has all of the pieces in place, but none of them finished. I sometimes call this a "spike." In the next issue I'll present the architectural prototype for the remote control television in literate programming style. ♀