THE Smalltalk REPORT

SIGS PUBLICATIONS

Publishers of Journal of Object-Oriented Programming, Object Magazine, C++ Report, The Smalltalk Report, The X Journal, Report on Object Analysis & Design, Objects in Europe, and Objekt Spektrum (Germany)

Table of Contents

February 1995 **Vol 4 No 5**

Features

Columns

Departments

The Smalltalk Report (ISSN# 1056-7976) 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. Second Class Postage Pending at NY, NY and additional Mailing offices. Canada Post International Publications Mail Product Sales Agreement No. 290386.
Individual Subscription rates 1 year (9 issues): domestic $79; Mexico and Canada $104, Foreign $119; Institutional/Library rates: domestic $119, Canada & Mexico $144, Foreign $159. 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 215.785.5996, 215.785.6073 (fax), P00976@psilink.com (email). PRINTED IN THE UNITED STATES.

Editors' Corner

We've spoken a number of times of the different types of applications that you are solving with Smalltalk. The major vendors have made it abundantly clear that their target market is the professional MIS market of the so-called Fortune 500 companies. The other thing that they have made clear is that they no longer see C++ as the major competitor of Smalltalk. Clearly, it is the PowerBuilders and VisualBasics of the world that they are competing against in the corporate board rooms.

This does leave out many other significant domains in which Smalltalk has been, and continues to be, used. There are numerous engineering shops that are making use of Smalltalk both for modeling and for development of actual production systems. Smalltalk has been used in many research arenas because of the power it offers. One area that we know has been using Smalltalk for some time is in the development of real-time systems. In these markets, Smalltalk is very much still competing directly with C++. And the knocks against using Smalltalk seem to remain constant. Comments that it is too slow to use, or the image size is too large to be used are still the complaints, even though most of them are no longer justified. In fact, while attending a talk a few weeks ago we were astounded to hear one member of the audience pose to the speaker the comment "having a garbage collector makes Smalltalk useless"! This to us says Smalltalk has an image problem.

These comments seem to be the right answer to the wrong question. Many have proven that Smalltalk can be used in real-time systems. That Smalltalk runs slower than equivalent code written in Assembler or C is a given. But surely this doesn't lead to the conclusion "therefore it isn't appropriate to ever use it." It just means if you need blinding speed use Assembler. While some real-time systems require exceptionally quick response time, what they all need is predictability, which can definitely be achieved using Smalltalk. Yes, we need performance, but predictable performance. Tuning Smalltalk systems is definitely possible. There are a number of tools available for doing just that. What's more, tuning many applications can be achieved by finding fundamentally better approaches, rather than making a poor design run faster. Since we can build solid, understandable models of our domains, it should be possible to find significant improvements to them, a task that is difficult using traditional approaches. What's more, if you find a part of your Smalltalk system that doesn't perform acceptably, it is always possible to rework that part using another language. As for the garbage collector being a problem, we're no experts in the field, but as has often been explained to us, the garbage collector only works as hard as you make it. That is, though it is not controllable, it is certainly predictable. If you know the garbage you're creating, it should be possible to predict how the garbage collector will behave. What's more, features such as the one ParcPlace has included for controlling the garbage collector's behavior are a step in the right direction. (We'll try to get someone who knows this area better than we do to write about this soon).

As we're just coming back to the reality of facing another cold January here in the "Great White North" it is time for our traditional post-Christmas wish list for the Smalltalk world. Some items are new, some have been on the list forever. But here goes:

1. Build a better browser. This has been number one on our list for many years now, but to little avail. What is required is not just minor changes, but a radical rework of the browser from the ground up. With no good reason, Smalltalk has lost the edge in terms of development environments. It is time to reclaim that title.

JOHN PUGH

PAUL WHITE

2. Fix the name space problem. Classes should not be global. We've argued in the past that this problem is at best an annoyance and at worst a real impediment to building large Smalltalk applications. In particular, the lack of proper name spaces is going to inhibit the growth of third party libraries coming to market. The solution of just adding prefixes to the front of all class names is just a patch rather than a solution to a deficiency in the language. It would be nice if the ANSI committee would have something to say about this one, but the chances of this are extremely slim.
3. Support private methods. In large system development, it is mandatory that the language itself support true private methods. Again, hopefully the committee will solve this one.
4. Provide testing tools. This wish is still fuzzy in our minds. It is clear that testing mechanisms are being created by different organizations using Smalltalk, but this seems to be inappropriate. It certainly goes against the goal of Smalltalk that is to achieve reuse and stop people from reinventing the wheel. The vendors must have testing mechanisms they use themselves, as do many of their customers with which they work closely. Hopefully, these tools (or at least their strategies) will be included in their products.
5. Provide documentation tools. Again, most organizations using Smalltalk realize there is a need to do a better job of documenting what is being constructed, but it is being done for the most part in a haphazard way. Certainly the vendors themselves have not led the way in terms of showing us how classes should be described within Smalltalk. We need to capture not just the descriptions of each of the methods, but the actual design of the class. As has been discussed more and more lately, it is more important that the designer of a class describe how they intended for the class to be used, rather than providing a description of how it was built.
6. Make available a "Smalltalk Lite." We continue to hope that someone will come forward with a $199 Smalltalk for the masses. This version could be nothing more than a return to the original style Smalltalks we had in the past. There is no denying we need the features that each of the vendors have been working so hard to include so that business can get their job done, but the individual working at home in their basement who wants to try Smalltalk out needs very little in terms of features. The importance of such people to the growth of Smalltalk should not be underestimated.

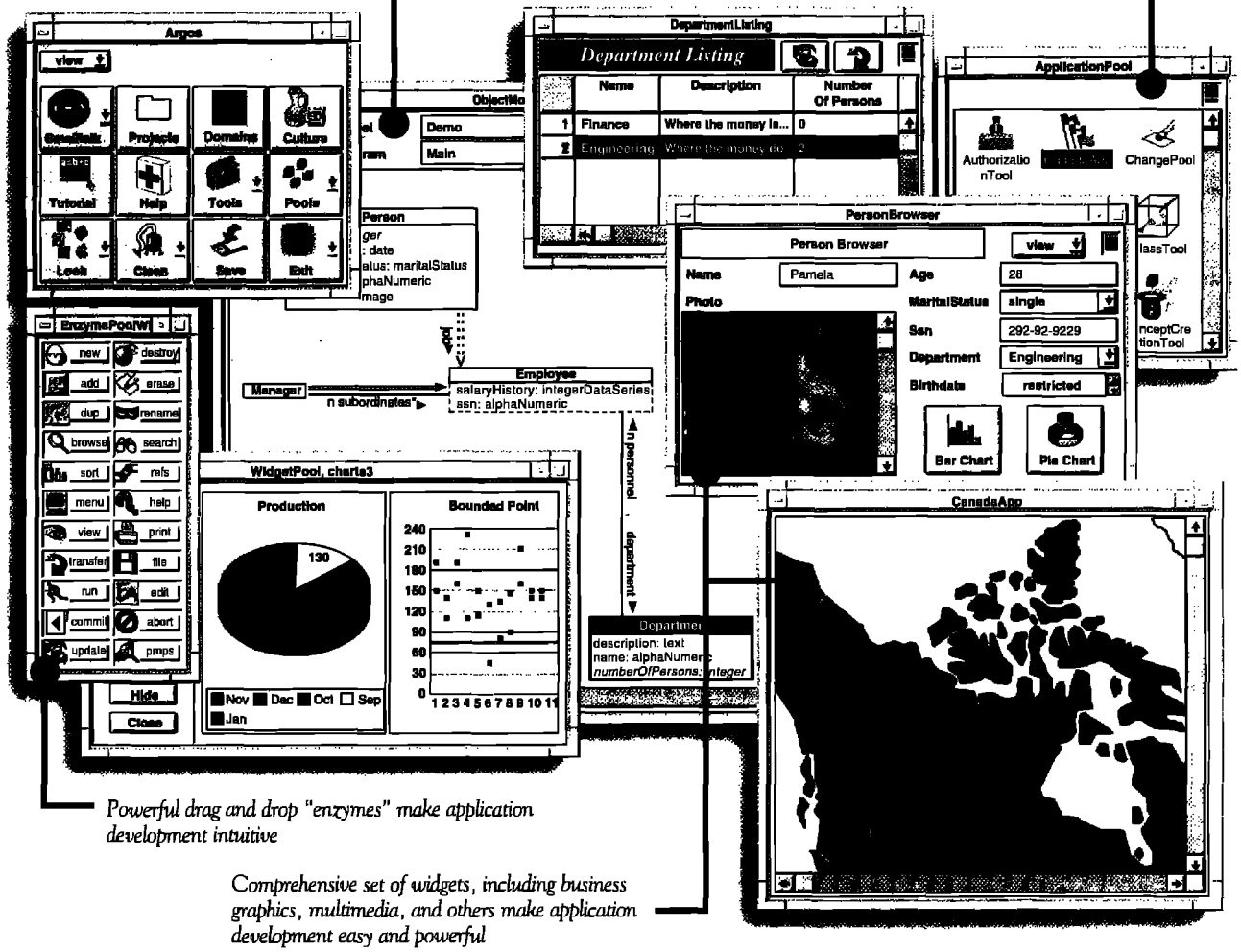
Enjoy the issue, and we hope to see many of you at the Smalltalk Solutions conference at the Omni Park Central Hotel in New York at the end of this month.

Introducing Argos

The only end-to-end object development and deployment solution

An integrated object modeling tool provides model-driven development for enterprise-wide applications

All object models are managed in a shared repository, supporting team development and traceability



Powerful drag and drop "enzymes" make application development intuitive

Comprehensive set of widgets, including business graphics, multimedia, and others make application development easy and powerful

VERSANT Argos™ is the only application development environment (ADE) that makes it easy to build and deploy powerful, enterprise-wide object applications. Easy because Argos features an embedded modeling tool and Smalltalk code generation that ensure synchronization between your models and applications. Powerful because Argos supports full traceability and workgroup development through a shared repository.

Argos automatically generates multi-user database applications that run on the industry-leading VERSANT ODBMS. Argos deals with critical issues such as locking and concurrency control transparently. And only Argos is packaged as a completely visual ADE built on ParcPlace VisualWorks®.

Leading organizations — in industries from telecommunications to finance — are using Argos to deliver business-critical applications. Find out how Argos can help you deliver your critical applications in weeks, instead of years.

Contact us today at 1-800-VERSANT, ext. 415 or via e-mail at info@versant.com

VERSANT
The Database For Objects ™

1380 Willow Road • Menlo Park, CA 94025 • (415) 329-7500

JAY
ALMARODE

Transactions in Smalltalk

IN MY PREVIOUS column, I described the architecture and advantages of multi-user Smalltalk. The key characteristic of multi-user Smalltalk is that a single object identity domain is accessible by multiple, concurrent users. Users share the same objects, not proxies to a remote system or duplicate copies mapped from a persistent store. This means that users share object behavior, as well as state. Rather than duplicating the same behavior in each application (and having to update each application when the behavior changes), the application sends messages to objects that reside in a single, globally shared image.

Since multiple users may be reading and modifying shared objects, the underlying Smalltalk system must make sure that a single user's view of objects is consistent. When a user reads or modifies an object, the user's operations must not be invalidated by other user's changes. For example, suppose an application maintains financial accounts with objects that encapsulate the account balance. A user that wants to transfer funds from account A to account B would cause the value in the account object for A to be decremented by some amount, and the value in account B to be incremented by the same amount. Since multiple users may be allowed to view the account balance in account A, it is important that concurrent users are not allowed to transfer funds based upon a view of the account that has since been decremented (unless we are allowed to make money out of thin air).

The way that a multi-user system maintains a consistent view of objects is with the notion of a transaction. A transaction is a bounded sequence of operations such that either all of the operations are executed to completion, or none of them are executed. This is called atomicity. In the example above, when transferring money between accounts, both the debit of account A and the credit of account B must occur, or neither must occur. Otherwise, the account balances may become logically inconsistent. In a transaction-based system, when a user invokes the "commit" operation, the underlying system guarantees that either all modifications that occurred since the transaction began are made persistent, or none of them are. If a user wishes to discard all modifications, then he or she invokes the "abort" operation. In a limited sense, single-user Smalltalk systems support the notion of a transaction with the operation to save the image (by writing all of object memory to a file). When the image is saved, all modifications that occurred since the last save operation are made permanent, analogous to a commit operation. Correspondingly, if the user quits the image without saving, it is equivalent to the abort operation. If you've ever made low level changes to the user interface or kernel classes, you know the practicality of being able to quit the image without saving. It is a convenient way to back out of changes that have made the system inoperable.

The notion of a transaction has another important ramification concerning object visibility. When a user begins a transaction, the user is presented a view of the world of objects that is based upon the last committed state. This is sometimes called a "transaction's point of view." As a user modifies objects, these changes are not visible to other users until these changes are committed. In addition, any new objects that a user creates are not visible to other users until the transaction is committed. There is another model of object visibility where a user is allowed to see uncommitted modifications performed by other concurrent transactions. In this model, when a transaction views an uncommitted modification to an object, the transaction becomes dependent upon the committal of the other transaction. If the other transaction should abort its changes, then the current transaction must be aborted as well. With this model of object visibility in a transaction, the application may not get "repeatable reads" of an object. Accessing the state of an object depends upon the time that it is accessed within the transaction, and may not yield the same result every time the object is accessed. This is problematic in object-based systems, since complex (and side-effect causing) behavior may be executed based upon the state of an object. This model also leads to the potential problem of "cascading aborts," where the aborting of one transaction causes a domino effect by requiring dependent transactions to abort.

In multi-user Smalltalk, the underlying system is responsible for managing transactions and maintaining logical object consistency. Since objects reside in a single object memory, this task is greatly simplified. The internal object manager has knowledge of which objects have been read or written, and directly coordinates the updating of object memory that is sharable by all users. In SmalltalkDB, the data definition and manipulation language for GemStone[1], the underlying system uses shadowing techniques to provide a transaction's point of view. When a transaction begins, the user is presented a view of objects based upon the last committed state of object memory. This view appears to the user as a private copy of all of object memory. Any modifications that the user makes are not seen by other users. When the user modifies an object, the modification is actually performed on a shadow copy of the object. When the transaction is suc-

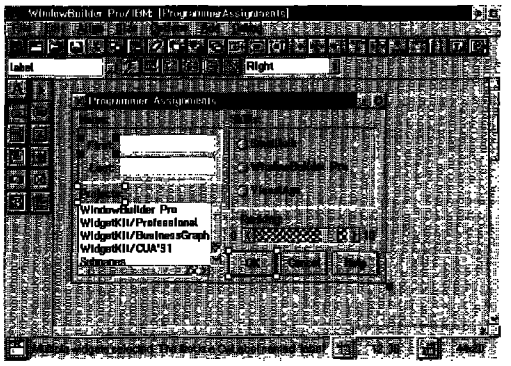
Jay Almarode can be reached at almarode@slc.com.

"The Difference Between Success and Failure in IBM Smalltalk"

New! **WindowBuilder™ Pro** is an interactive tool that lets you build polished user interfaces fast in Smalltalk from Digitalk *and* IBM. WindowBuilder Pro (WBPro) saves you from the job of building UIs in code. It helps simplify maintenance and increase consistency.

Like VB, with Real Objects

Select controls from a palette. Place and edit them interactively. Integrate the controls with your app easily. Build composites of controls to create your own reusable UI components. Place and edit them in WBPro just like the native controls. Get portability of your UIs across all the supported platforms of a Smalltalk family. Includes autosizing, automatic alignment, control of fonts, menus, colors, and more.



Building user interfaces is easy

High-Level Controls for WBPro

When you use the high-level add-on controls like spreadsheets, business graphics, and others, your apps will be more powerful and polished. And you'll save even more time and effort. Inquire about specific offerings and platform availability.

"For most Smalltalk/V programmers, WindowBuilder Pro/V is a survival tool—the difference between success and failure."

— Milan Sremac, President, Medical Software Systems

WINDOWBUILDER PRO/V (VER 2)		
For Digitalk Visual Smalltalk	Windows	$495
	OS/2	$495
For Smalltalk/V Win16 (WBPro/V ver 1)		$295

"Unless you're totally comfortable with the Motif API, a tool like WindowBuilder Pro is the difference between success and failure in IBM Smalltalk."

— Gordon Sheppard, Senior Technologist, American Management Systems

WINDOWBUILDER PRO (FOR IBM SMALLTALK)			
OS/2 std.	$495	Team	$695
Windows std.	$495	Team	$695

No runtime fees are required for applications developed with WBPro. Free support for the first 90 days. All products include complete documentation. Support subscription available. WindowBuilder Pro/V is compatible with Team/V. Code generation in IBM Smalltalk is totally Motif compliant. © Objectshare Systems, Inc. 1994

Visual Smalltalk

WindowBuilder™ Pro/V lets you build UIs interactively, save time, simplify maintenance. Version 2 is fully compatible with Visual Smalltalk and Visual Smalltalk Enterprise. Generate ViewManager subclasses, ApplicationCoordinator subclasses, or PARTS windows.

WINDOWBUILDER PRO/V			
Windows	$495	OS/2	$495

Upgrade WindowBuilder Pro/V to version 2. We have special upgrade pricing to registered users. Please inquire.

Subpanes/V provides columnar list box, hierarchial list box, table pane, bitmap pane, bitmap button, 3-D frames, and more. Requires WindowBuilder Pro/V ver 2 and Visual Smalltalk or Visual Smalltalk Enterprise.

SUBPANES/V			
Windows	$235	OS/2	$235

VisualAge

Spreadsheets, business graphics, and other high-level components are easy to add to your VisualAge™ based applications.

WidgetKit™/Professional has powerful spreadsheets and more. You get virtual spreadsheets, multi-column list boxes, table editor, graphic viewers for BMP, PCX, and GIF, input validation, file system widgets, and more.

WIDGETKIT/PROFESSIONAL			
Windows std.	$495	Team	$795

WidgetKit/Business Graphics has versatile graphs and charts. You get bar, pie, area, line, gantt, high-low-close, scatter, and more basic types. Options include 2-D and 3-D, fonts, colors, control of printing, and more.

WIDGETKIT/BUSINESS GRAPHICS			
OS/2 std.	$495	Team	$795
Windows std.	$495	Team	$795

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 (408) 970-7280

Or call for free info. 9 AM to 5 PM PST, M-F. 30-day money-back guarantee

Getting Real

cessfully committed, the shadow copy is merged into shared object memory by the underlying object manager. At this time, other users gain visibility of the transaction's modifications and any new objects that were created during the transaction. In addition, the user's view of objects is refreshed to include any modifications committed by other transactions in the interim. When a transaction is aborted, any modifications that were made to objects are lost, and the user's view of objects is refreshed. However, the user does not lose any new objects that were created before the abort occurred. As long as the application retains a reference to the newly created objects, it can continue to access them, and possibly commit them at a later time.

The task of maintaining logical object consistency is slightly more complex for other architectures where a relational database (or other persistent store) or remote object messaging is used to share objects in single-user Smalltalk systems. In applications where a relational database is used to store an object's state, the application must transfer modifications that are performed on an object into updates to a relational table. Since the Smalltalk image exists independently from the database, an application developer

> *The underlying system manages transactions and maintains logical object consistency in multi-user Smalltalk. Objects reside in a single-object memory and this task is greatly simplified.*

must decide upon some means to keep object memory in synch with that state of the database. This problem is commonly called the "two-space problem."

When using a relational database or other persistent store to share objects, the Smalltalk application must make sure that when modifications are flushed to the database (for example, by causing the execution of SQL update commands), the modifications are atomic. This usually means utilizing whatever transaction mechanism is provided by the database. In the earlier example where the Smalltalk application has objects that represent account A and account B, there are corresponding rows in a relational table that holds the account balances for both of these objects. An application developer must make sure of at least two things when the modifications to the two objects are flushed to the database: 1) the state of the corresponding rows have not changed from the time they were initially read when constructing the account objects (or at least have not changed in such a way as to invalidate the fund transfer), and 2) the two SQL update operations are per-

20,001

AND COUNTING

As we rapidly move beyond the 20,000 mark in the number of students we've enrolled, more and more people like you are choosing Semaphore as their primary source for object technology training and consulting. It's a record no one can match.

Focused on object technology and only object technology, Semaphore's staff of more than fifty highly effective real-world specialists train you to maximize your object skills. No matter what your level of experience, no matter where you are in your software development, Semaphore can help you reach your goals.

Semaphore offers over thirty courses, on-site and open enrollment programs, worldwide training, and comprehensive consulting that supports your entire software development cycle. The bottom line? Semaphore is achieving the most impressive numbers in the field.

Object Technology Specialists

SEMAPHORE

Semaphore, 800 Turnpike Street, North Andover, MA 01845, USA
(508) 794–3366 • Toll Free: (800) 937–8080 • Fax: (508) 794-3427 • Email: 505.4433@mcimail.com

Call for your FREE Semaphore Object Technology Solutions Kit: 1–800–937–8080

Getting Real

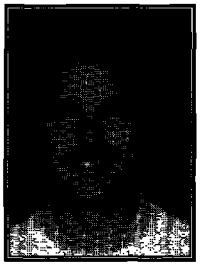
formed atomically. The first problem is solved by acquiring locks on the rows of the table or by re-reading the rows prior to the update to validate that they have remained unchanged. The second problem is solved by placing both update operations in a database transaction.

In applications where objects in one Smalltalk image can send messages to remote objects in another Smalltalk image, these same issues must be addressed. The developer must design the application so that when changes are committed in one Smalltalk image (i.e. the image is saved), any modifications that occurred to remote objects in other images are also committed. For example, if the object for account A resides in one Smalltalk image, and the object for account B resides in another, both images must commit their changes, or the objects may become logically inconsistent. If both account objects reside in the same image, but their modifications are caused due to a message sent from a remote image, their changes cannot be committed unless the remote sender notifies them that it expects them to commit. This is because the remote sender may have determined that the changes should not occur after all. This problem is solved using two-phase commit protocols. In this scheme, a Smalltalk image must ask all remote images in which it caused modifications if they can commit their changes. If a remote image answers yes, then it must guarantee that if asked to do so, it can commit its changes, even in the face of hardware failure. This is typically done by writing some logging information to disk before answering affirmative to the request. If all remote images answer yes, then the coordinating Smalltalk image can send a second command to the remote images, telling them to commit their changes. Note that this scheme does not allow a Smalltalk image to execute messages from more than one remote transaction at a time and maintain logical object consistency. This is because one remote transaction may request that the local Smalltalk image commit its changes, while another remote transaction might request it to abort. Since a save operation will write all of object memory, an image cannot selectively commit modifications to some objects and not others.

To build industrial strength multi-user applications in Smalltalk, the system must support the notion of transactions. Sometimes a transaction may not be allowed to commit to ensure that objects remain logically consistent. The inability to commit a transaction is necessary when other transactions have performed operations that invalidate the operations in the current transaction. My next column will discuss concurrency conflicts in multi-user Smalltalk and how application developers can avoid them.

Reference

1. Bretl, B., *et al.* The GemStone Data Management System, OBJECT-ORIENTED CONCEPTS, DATABASES, AND APPLICATIONS, W. Kim and F. Lochovsky, Eds., ACM Press, 1989.



KENT BECK

Garbage Collection Revealed

THIS MONTH I'LL talk about garbage collection. To paraphrase Mark Twain, everybody talks about the garbage collector, but nobody does anything about it. All of the commercial Smalltalks provide some ability to tune the garbage collector, but without knowing what's going on and why, you are unlikely to be able to know when these features are applicable or how to use them. This article discusses the common vocabulary of modern garbage collection. Later, we'll explore what you can do to tune the garbage collector in the various Smalltalks.

THE IDEA

In the early days of programming languages, programmers had to decide at compile time how much memory they needed. Languages like FORTRAN and COBOL had a simple runtime model as a result, but they aren't very flexible. Along came LISP, which let you allocate storage at runtime. LISP was very flexible, but what got allocated needed to get deallocated. The first LISP implementations would run until they filled memory, then die. It was clear that when the system filled memory, much of the storage was no longer in use. It had been used for a while, but then it could be safely reused, because it would never be used by the program again. Rather than make the programmer responsible for deallocation, early Lispers decided to have the system deallocate memory for them.

At first, automatic storage deallocation was considered an artificial intelligence problem. After all, how could you possibly know that a piece of memory would never be accessed again? Only a trained programmer could tell with any certainty, and even they weren't very accurate.

It wasn't long before someone noticed that in a type safe language (that is, one where you can't arbitrarily create pointers to memory) the problem is conceptually quite simple. Once the last pointer to an object is lost, there is no way to get another pointer to it. Therefore, you can't possibly harm the execution of the program by reusing that memory.

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, 408.338.4649 (voice), 408.338.3666 (fax), or by email at 70761,1216 (Compuserve).

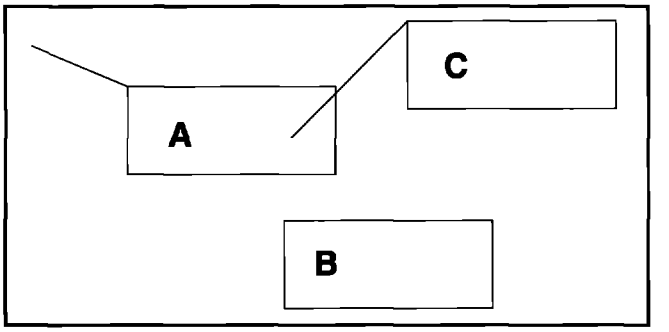


Figure 1. Object B's memory can be safely reused.

In Figure 1, since there are no references to object B, the program is free to reuse the memory it occupies, safe in the knowledge that no part of the program can possibly refer to it again. Object C cannot be reclaimed, because it is referred to by object A. Object A cannot be reclaimed because it is referred to from outside the object memory.

The code that finds objects that are no longer referenced is called the "garbage collector." Your Smalltalk contains a garbage collector. While most of its workings are beyond your control, it will occasionally become a most important part of your life. When you are trying to squeeze performance out of a running system, or reduce its memory footprint, you will have to understand what's going on "under the hood."

One common mistaken impression is that the garbage collector runs "occasionally," almost of its own volition. The garbage collector always runs in response to a request for memory it cannot fulfill. The memory allocator looks for the requested memory, but can't find it. It invokes the garbage collector, which reclaims some memory. The memory allocator runs again, returning some of newly freed memory.

The presence of a garbage collector is an integral part of the Smalltalk programming experience. When you have to explicitly deallocate memory, you program in a very different style. The hard cases are where several parts of the system share an object, and all of them must agree before it can be deallocated. This introduces a pattern of communication to the system that likely wouldn't exist if not for the deallocation problem. A garbage collector, **because it needs to have** a global view of the system, frees you from having to take a global view. The connections between the parts of a program can be much looser, because they never have to communicate about deallocation. You never have to write otherwise irrational code just to make sure memory gets deallocated correctly.

Your Smalltalk implementation (the virtual machine) provides you with two main resources- message sending and object allocation (and hence garbage collection). The right attitude 95% of the time is to assume that both are free. The right time to stop this charade is when you have gotten the design as clean as you possibly can at the moment and it is obvious that limited machine resources are going to pose a problem for your user. Then you need to have a model in your head of what is going on.

BAKER TWO SPACE

Here's a simple garbage collection algorithm: allocate twice as much space for objects as you think you'll need. Divide the memory in two equal sections, called Old and New. When you allocate an object, allocate it in Old space. (See Fig. 2.)

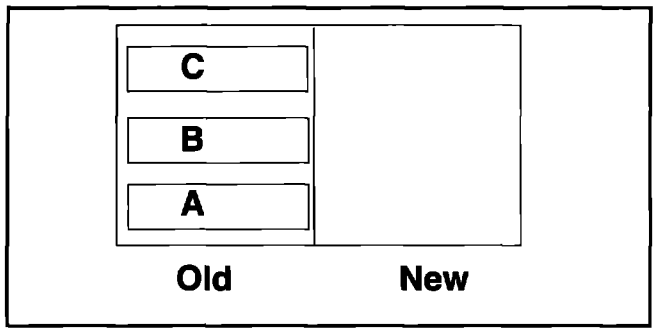


Figure 2. Allocating objects in old space.

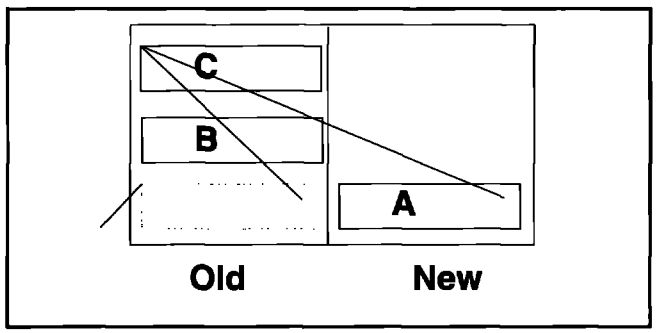


Figure 3. Copying a known object to new space.

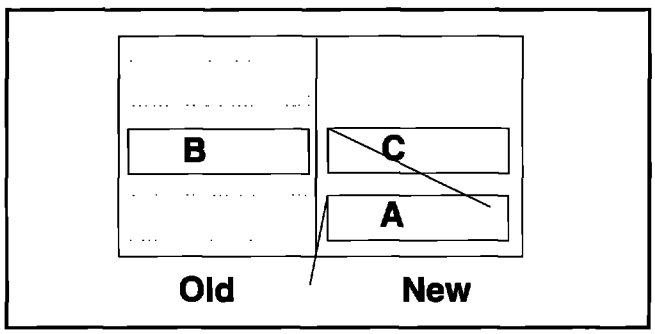


Figure 4. Copying a refered-to object to new space.

When you want to allocate an object, but Old space is out of room you have to invoke the garbage collector. The collector runs by starting with a known live object in Old space (in this case A) and copying it to New space. (See Fig. 3.)

Any object that gets copied to New space has all of its objects copied to New space, too (in this case C). (See Fig. 4.)

When no more objects remain to be copied, any objects remaining in Old space are not referenced anywhere. In this example, B can be safely ignored. Swap the identities of Old and New space. New objects will be allocated in the same space as the surviving objects. (See Fig. 5.)

This algorithm is called Baker Two Space after its inventor, Henry Baker. It advantages are:

- it is simple
- it automatically compacts surviving objects together, leaving the remaining free space in one big chunk

Its disadvantages are:

- it takes twice as much memory as the object actually occupies
- the copying operation takes time proportional to the number of surviving objects

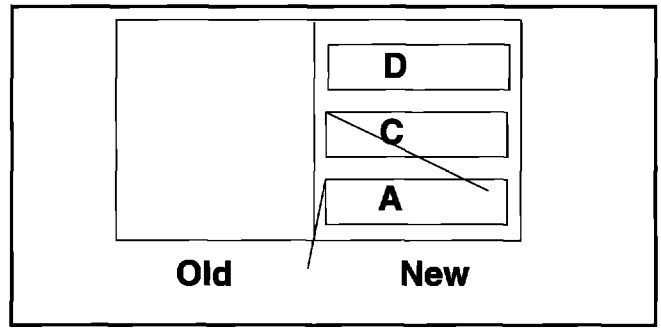


Figure 5. Objects are allocated in old space.

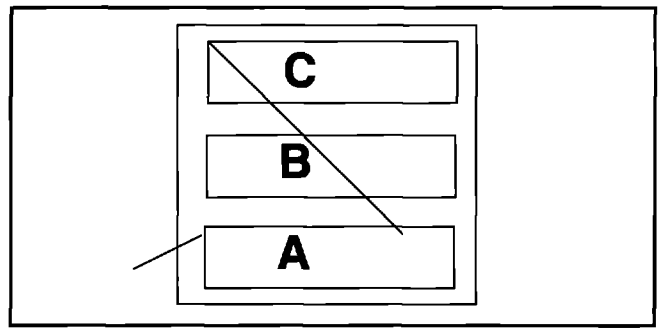


Figure 6. Mark and sweep objects in a single space.

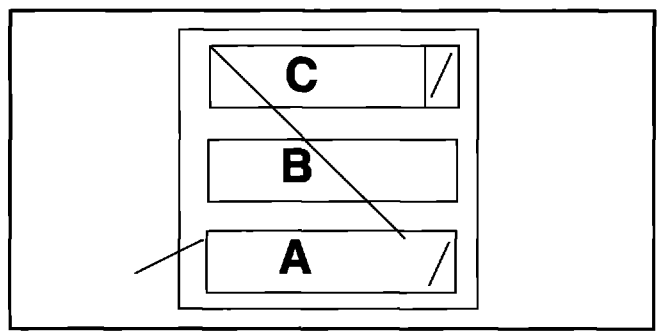


Figure 7. After marking.

MARK AND SWEEP

The mark and sweep algorithm addresses the disadvantages of the Baker Two Space algorithm (it actually appeared many years before Baker Two Space). All objects are allocated in a single space. (See Fig. 6.)

As before, when the allocator runs out of space, it invokes the garbage collector. This time, instead of moving surviving objects, they are merely marked as being alive. Objects referred to by marked objects are also marked, recursively, until all the objects that can be marked have been. (See Fig. 7.)

After all the surviving objects have been marked, the sweep phase goes through memory from one end to the other. Any object that isn't marked is put on a list of memory available for allocation. While sweeping, the marks are erased to prepare for the next invocation of the garbage collector. (See Fig. 8.)

The mark and sweep algorithm has the following advantages:

- it doesn't require extra memory
- it doesn't need to move objects

However, it has some serious shortcomings:

- the marking phase takes time proportional to the number of surviving objects
- worse, the sweeping phase takes time proportional to the size of memory

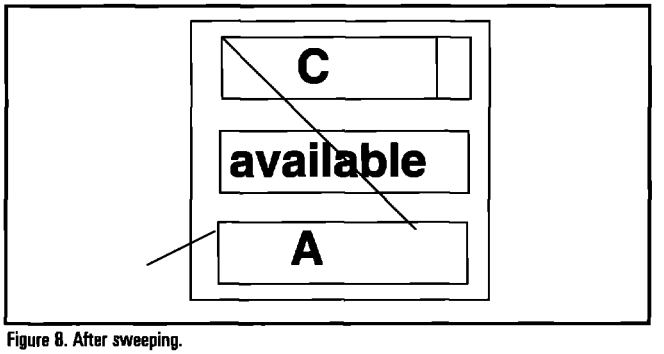


Figure 8. After sweeping.

- the resulting available memory is fragmented, possibly requiring a separate compaction step to pack the surviving objects together

GENERATION SCAVENGING

While a graduate student at Berkeley, David Ungar combined the two space and mark and sweep algorithms to create a collector which usually exhibits none of the weaknesses of either, and has some important new properties. He called it generation scavenging.

The observation that makes generation scavenging work is that as a rule objects die young or live forever. That is, many objects are used temporarily during computations. For example, here a Rectangle creates a Point to calculate its extent.

```
Rectangle>>extent
    ^self corner - self origin
```

Similarly, a Point creates a new Point to hold the maximum of its coordinates and the parameters coordinates.

```
Point>>max: aPoint
    ^(self x max: aPoint x) @ (self y max: aPoint y)
```

A client might use extent to compute the merged size of several Rectangles.

```
Client>>extent
    ^self rectangles
        inject: 0@0
        into: [:sum :each | sum max: each extent]
```

The Points created by invoking extent only live long enough to get passed as parameters to Point>>max:. The Points created by Point>>max: live over two invocations of the block, one where they are created, the next when they are replaced. If Client has a 100 Rectangles, Client>>extent creates 200 Points which are all garbage even before the answer is returned.

Generation scavenging uses the demographics of objects to

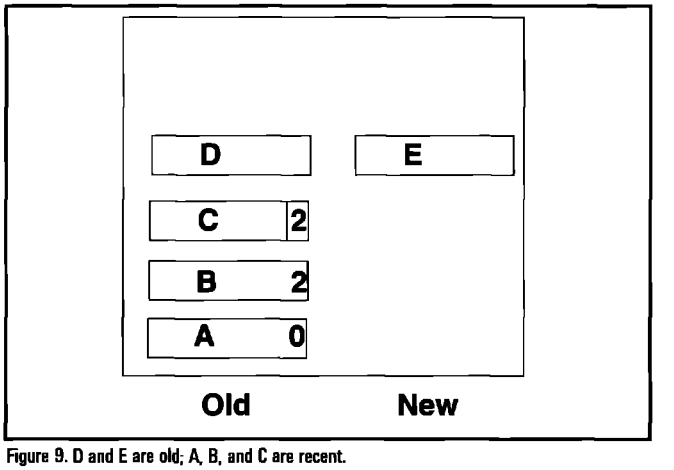


Figure 9. D and E are old; A, B, and C are recent.

advantage. The relatively expensive two space collector is lavished on newly created objects. The copying operation of the two space collector is called a "scavenge."

The generation scavenger keeps track of the age objects by incrementing a count every time an object is copied by the two space collector. When the count exceeds a threshold, the object is copied not into New space, but into Tenure space. Tenure space is managed by a mark and sweep collector.

This has the effect of concentrating the collector's efforts on newly created objects, the ones that are likeliest to be collectable. After an object has demonstrated a little longevity, the collector effectively ignores it. Only when tenure space fills or you take a snapshot, will the mark and sweep collector examine tenure space.

By concentrating its efforts where garbage is most likely to be found, generation scavenging garbage collectors end up taking only a small fraction of the total time of the system. In general, the collector only takes a few percent, compared with 20–30% for earlier algorithms.

The other valuable property of generation scavenging is that it is insensitive to the number of objects in the system. Recall that the two space algorithm takes time proportional to the number of surviving objects. Since most of the objects in the system are in tenure space, generation scavenging takes time proportional to the number of recently created surviving objects. Limiting the size of New and Old space keeps that number small.

A TENURING MONITOR

All of this is fine in theory, but what about practice? The collector is like a pair of shoes. You don't really notice it unless it is causing you pain. Then you have a serious problem.

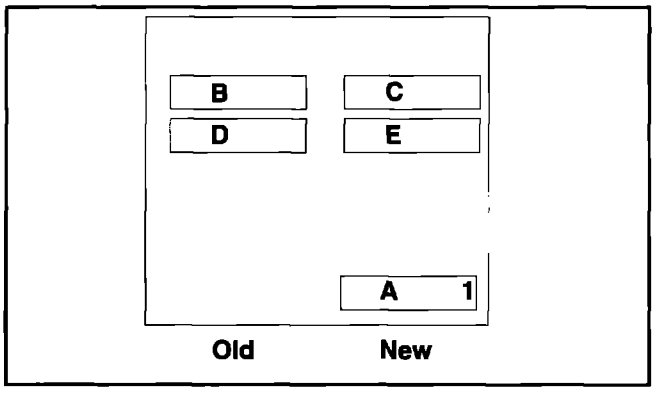


Figure 10. B and C have been tenured.

I'm running out of space this month, so I'll have to cover garbage collection tuning in future columns. I'll leave you with a little utility that will to help you begin to understand the interaction of your program with the collector.

The most serious breakdown of a generation scavenger is when it acts like a mark and sweep collector. If objects live just long enough to be tenured, then die, all the efforts spent on scavenging are wasted.

In old versions of Smalltalk/V the execution of the mark and sweep collector was accompanied by a cursor shaped like a vacuum cleaner. This lead to the use of "hoover" as a verb, "I was creating lots of objects, and boy, was I getting hoovered."

The new version of Smalltalk/V, Visual Smalltalk, provides hooks for watching the collector. In particular, the global object Processor posts the event flip when a scavenge takes place. You can send the message bytesTenured to find out how many bytes worth of objects were moved to tenure space.

I built the tenuring monitor with Parts. I know of no good way to typeset a Parts application, so I'll just try to sketch it out well enough for you to reproduce it if you want to.

The design of the user interface is a window with a static text in it. The text displays the number of bytes tenured with every scavenge.

First, we create a window and put a static text into it. Then we need to have the static text notified when a scavenge happens. Give the static text the following script (Digitalk calls them flips) and link it to the open event of the window:

```
SetDependencies
    Processor
        when: #flip
        send: #UpdateBytes
        to: self
```

When the window closes, the static text should stop getting notified, so define the following script and link it to the aboutToClose event of the window:

```
BreakDependencies
    Processor
        removeActionsWithReceiver: self
        forEvent: #flip
```

Finally, when the static text gets UpdateBytes, it needs to display the number of bytes tenured by the latest scavenge. It gets

continued on page 30

A sample pattern language—Concatenating with Streams

Bobby Woolf

I WOULD LIKE to elaborate on Alan Knight's "Performance Tips" article in THE SMALLTALK REPORT, 4(1). In his article, Alan briefly discussed using streams as a more efficient technique for performing concatenation.* I would like to show how to document this technique more thoroughly using patterns. This example will also show how one pattern can easily lead to others and form a pattern language.

WHAT IS A PATTERN?

Regular readers of THE SMALLTALK REPORT have seen numerous pattern examples in Kent Beck's "Smalltalk Idioms" column. In each column, Kent describes at least one commonly used technique and documents it using a pattern.

A PATTERN DOCUMENTS EXPERTISE

The concept of patterns was first described by Christopher Alexander, an architect who theorized about how best to design buildings and towns. He describes a *pattern* as the documentation of a common problem and its solution.† This can also be phrased as "a solution to a problem in a context."‡ It is a mechanism through which an expert in a field can document his expertise, the various tricks and techniques he has learned which make him an expert. Thus a pattern is only as good as the person who wrote it. In fact, a pattern is frequently less complete than the author's understanding of the problem because even an expert is often unable to completely express in words all of his understanding.

A pattern is much like a scientific theory: As its accuracy is confirmed through repeated use, its acceptance grows. But when it fails to accurately predict results, it must be revised to include these new circumstances. A theory can never be proven to be fact, and a pattern can never be proven to be right. For this reason, a pattern is never really finished. It evolves to reflect further experience gained through its use.§ A pattern can only be considered finished when the writer understands a problem completely and documents it perfectly.

A number of people in the computer software industry have discovered Alexander's work and found his concepts of patterns to be useful when applied to software engineering tasks. The Hillside Group formed a few years ago to investigate and promote the use of patterns in the software industry.|| It recently held a conference, PLoP '94 (The First Annual Conference on the Pattern Languages of Programs) to further coordinate this effort.#

A PATTERN FOLLOWS A FORMAT

There is considerable debate among software engineers about what the format or template for a pattern should be. Alexander describes a format for his architecture patterns,** but it is not easily applied to software patterns. Whatever format is used, a pattern consists of at least four discrete parts††:

- A *title*. This is *who* the pattern is, a name for easy reference. While many authors prefer to give each pattern a name that describes the overall pattern, I prefer a name that summarizes the solution in a sound bite.
- An explicit *problem* statement. This describes *what* the entire pattern is about, what problem it will demonstrate how to solve. The problem statement is specific enough to accurately describe the dilemma, but general enough to apply to the widest possible range of examples.
- A discussion of the *forces* or *constraints*. This section describes *why* the problem is difficult to solve. It defines the various obstacles that must be overcome and explores alternatives for doing so. The forces/constraints set the boundaries of the problem and guide the reader to the solution.
- An explicit *solution*. This shows *how* to solve the problem. It is stated as a clear recommendation of a course of action to be taken by the reader. The solution has the same level of specificity as the problem.

It is possible and often preferable for a pattern to have additional parts, but only the four listed previously are required. I prefer to combine the forces and constraints together into a Context section. I also include an Example section in my patterns, but that is not a requirement a pattern must meet.

The title should be just a few words that name the pattern, one that people will easily associate with the pattern. The problem statement should be short and simple. It is what the reader

* I first saw this technique documented in Ken Auer's "Efficient Smalltalk Programming" tutorial at OOPSLA '92 in Vancouver, BC, Canada.

† See page x of Alexander.[1]

‡ see Coad.[5]

§ See page xv of Alexander.[1]

|| Beck[2] contains details about the Hillside Group's origins.

A book by The Hillside Group,[6] due this year, will contain 30 pattern languages from the proceedings of PLoP '94, the first annual Pattern Languages of Programs conference. PLoP '95, which will be held September 6–8, 1995, in Monticello, IL, has issued its preliminary call for papers. For more information, contact Richard Gabriel at rpg@parcplace.com.

** See pages x–xi of Alexander.[1]

†† The four parts I have listed are my opinion. Other opinions on which sections are key include: 1) Gamma *et al.*[4] lists four elements—pattern name, problem, solution, and consequences; 2) Beck, page 20,[2] lists three parts—problem, context, and solution; 3) Coplien[3] discusses four parts: the problem the pattern solves, the trade-offs it resolves, the context in which it applies, and the particulars of its implementation. There appears to be less debate about whether the section/parts should be explicitly labeled. Although Alexander did not label his sections, the aforementioned authors and I all do.

will review to quickly find whether a pattern meets his current needs. The solution should also be concise, but long enough to describe all of the steps the reader should take and any exceptions to the rule that he may encounter. The real meat of the pattern is the discussion of forces/constraints. This section teaches the reader about the problem and documents the writer's conceptualization of it. It considers alternate solutions and shows why they were rejected. In the end, it justifies the solution.

For a couple of examples of patterns, see the sample pattern language included in this article.

A PATTERN IS REUSABLE

Ideally, a pattern describes a solution to not just one problem but rather a range of related problems. Thus the reader can encounter several seemingly unassociated problems that fall into this range. The context section will show that the pattern applies to each of these "different" problems such that all of them have the same solution. In this way, the solution to one problem can in fact be reused to solve many.

Because patterns have this reusability, once an author has documented the solution to a problem using a well-written pattern, he should never have to document that solution again. (On the other hand, as mentioned earlier, a pattern is never really finished. Both the author's understanding of the problem and his ability to express his understanding will evolve. As they do, he should update the pattern accordingly. However, the pattern is available for reuse throughout its evolution.) Any time another problem touches upon this one, he will be able to simply refer back to this pattern as the ready-made solution.

A PATTERN ENCAPSULATES A SOLUTION

Each pattern must be a small, self-contained chunk that is relatively easy to understand on its own. If a pattern becomes too long and complex, it will lose its focus of presenting a specific solution to a specific problem. Should this happen, the pattern must be refactored into a series of smaller patterns.

Thus a complex problem requires more than one pattern to derive its solution. Each pattern will describe a specific problem and its solution, and the patterns will build on and reinforce each other. The solution offered by the pattern family whole is greater than the sum of its pattern parts, therefore the family will present a more elaborate solution to the complex problem. Alexander called such a collection of collaborating patterns a pattern language.

WHAT IS A PATTERN LANGUAGE?

Individual patterns document individual techniques, but an expert in a topic has numerous techniques at his disposal. His art is knowing how to combine these techniques to form a methodology for solving a range of problems within a domain. When coupled in certain ways, his techniques form a structure of solutions far more useful than the sum of the individual parts. Yet when mixed together haphazardly, the guidelines cancel out each other's value. This can leave the reader at a loss as to how to apply small patterns to solve large problems.

A *pattern language* is a collection of patterns that reinforce each other to solve an entire domain of problems. Each pattern in a language leads to others. Large, broad patterns contain smaller, specific patterns. A language's shape is a multidimensional web of patterns referring to one another. But paper is two-dimensional and a reader's attention is one-dimensional, so a pattern language is written as a list. This list guides the reader, starting with an overall problem; through subsequent patterns, the language explores the various issues involved and discovers the specific solutions that will be required.

Pattern languages can be nested, forming a language consisting of sub-languages consisting of sub-sub-languages. Each of these is a pattern language of its own that just so happens to be part of a broader pattern language. Just as a tree may actually be a branch in a larger tree, a pattern language is a sub-language in one or more larger languages. In theory, a pattern language describing a feature in Smalltalk is part of "the" Smalltalk pattern language. The Smalltalk pattern language is part of the object-oriented pattern language (as would be parallel languages for C++ and other object-oriented languages). Furthermore, the object-oriented pattern language is, in turn, part of the software engineering pattern language.

A SAMPLE PATTERN LANGUAGE: CONCATENATING WITH STREAMS

This is an example of a simple pattern language. It is very Smalltalk specific. As in the "Performance Tips" article, it teaches the reader that it is more efficient to use streams for string concatenation than to use the concatenate message. Because it is written in pattern form, it clearly describes why the solution works and when to use it. For example, it notes that streams can be used to concatenate any SequencableCollection, not just Strings.

What makes this a language is that the overall solution is presented not in one pattern but in three. The first pattern is the main one and discusses the most important issues documented by the pattern language. In the process, it touches on two other problems and refers the reader to other patterns that resolve them. Because the other two patterns are referred to by the main pattern, they are included in the language (otherwise it would be a one-pattern "language").

Notice that these three patterns could also refer to even more patterns. The reader might not know Smalltalk and thus would need patterns describing problems that are solved using strings, streams, and concatenation. Pattern 2 refers to unnecessary garbage collection; the reader may require a whole separate pattern language on problems encountered in memory management and why Smalltalk's dynamic garbage collection is a good solution. The reason these patterns are not included in this language is that I, the author, decided that they were outside the scope of this language. Although they probably belong in a larger-context language that describes Smalltalk in general, they do not belong in a specific sub-language that discusses concatenation using streams.

CONCATENATING WITH STREAMS

Pattern 1: Use a stream for multiple concatenations

Problem: What is an efficient way to concatenate together a number of strings (or other collections) into a larger string?

Context: Concatenation (which is implemented in ParcPlace

Smalltalk by the method in SequenceableCollection whose name is a comma) is convenient, but somewhat inefficient. To concatenate two lists (a list being some kind of sequenceable collection), *a* and *b*, a third list, *c*, is created, then *a* and *b* are copied into it. To then concatenate *c* and *d*, a new list *e* is created to hold copies of *c* and *d*. Thus each concatenation requires creating one new object plus iterating through and copying each of the elements in both of the arguments. This is necessary for the first concatenation, but a series of concatenations creates a number of intermediate objects and involves copying the same sublists repeatedly.

A better solution would create fewer new objects and copy the sublists as few times as possible. The solution should work for any pair of sequenceable collections, but will most commonly be used to concatenate strings.

To quickly concatenate a couple of short lists, the comma message is simpler. A more complex technique would be appropriate for concatenating together numerous and/or long lists.

Solution: Use a WriteStream to perform multiple concatenations. Create a stream that contains what will be the result list, add each of the lists to be concatenated into the stream, and then return the resulting list.

A tip when concatenating strings: One common source of strings to concatenate is the method printString. Pattern 2 suggests using printOn: instead of printString, and Pattern 3 gives preference to print: over printOn:. So use print: instead of printString.

Examples

Here's a simple way to concatenate several strings:

```
descriptionString
    ^ 'I am a ', self class name, ' whose name is ',
self name, ' with a value of ', self value
printString, '.'
```

Using a stream is a more efficient way to compute the same string. The general technique is to replace every concatenation comma message with nextPutAll:, which will add the string to the stream. Other WriteStream messages, such as print: and nextPut:, are also helpful:

```
descriptionString
    | stream |
    stream := (String new: 100) writeStream.
    self descriptionOn: stream.
    ^ stream contents

descriptionOn: aWriteStream
    aWriteStream
        nextPutAll: 'I am a ';
        nextPutAll: self class name;
        nextPutAll: ' whose name is ';
        nextPutAll: self name;
        nextPutAll: ' with a value of ';
        print: self value;
        nextPut: $.
```

Notice that I broke the implementation into two methods. This way, if the description string is going to be concatenated with another string, a stream can be used directly.

Some subtle efficiencies to note in the transformed method:

The message nextPut: was used to add a single character; that is more efficient than using nextPutAll: to add the one character string '.'. And, as mentioned in the solution, I used "print: self value" instead of "nextPutAll: self value printString".

This example shows strings being concatenated but this technique can be used to concatenate any series of SequenceableCollections.

Pattern 2: Avoid creating intermediate objects

Problem: How can I avoid creating intermediate objects—ones that are not needed by the methods that obtain them—in my code?

Context: Intermediate objects waste memory by taking up space. They waste CPU time, first when being created, then when their memory is reclaimed (during garbage collection).

Often the reason a method receives an intermediate object is because what it really wanted was a similar object, so it takes the one it received and converts it into the one it wanted. The method should be more specific and ask for the object it wants so that it will not need to convert it.

Code with a series of message sends is less encapsulated because each message send assumes it will be understood by the answer returned by the message before it. By replacing a series of message sends with a single one, the code is both better encapsulated and easier to read.

A single message send is not always more efficient than multiple ones because the single message's implementor may create more intermediate objects than multiple explicit message sends would.

Solution: Avoid creating intermediate objects by sending an object a message that will return the answer object you want, rather than an intermediate object to which you have to send further messages to get the object you want. If the message you're sending returns an object that requires conversion, look for and use another (usually in the same receiver's public protocol) that will return the final resulting object.

Be aware, however, that the implementor of the message that returns you the object you want may in turn create numerous intermediate objects. It may create those unwanted objects itself or use other messages that create them. The goal is not just for your code to create as few unwanted objects as possible, but for your code and all of the code it uses to minimize such objects. In general, though, if everyone writes efficient methods that minimize intermediate objects, all code that uses those methods benefits.

Another technique: It is often tempting to ask an object to return an answer so that you can use it to perform a certain task. Instead, ask the object to perform that task for you. To do so, it can use the answer you would have received without creating a new object to return the answer to you.

When you make these simplifications to your code, it will become more efficient, and will often make it easier to read as well.

Examples

Example 1: One way to determine the height of a rectangle is:

```
rect extent y
```

However, extent computes and returns an intermediate object, a Point, which is then sent y and thrown away. To avoid creating this unneeded object, do this:

```
rect height
```

where height is implemented as:

```
Rectangle>>height
    ^ corner y - origin y
```

This message will perform two accesses and a simpler calculation than extent performed (subtracting Numbers instead of Points). It will return the object you want, with no more accessing or conversion required.

Example 2: Similarly, when adding strings to a stream, the intermediate string is usually avoidable. This will print an object on a stream:

```
myStream nextPutAll: anObject printString
```

The problem is that printString returns a String tha4t is thrown away after nextPutAll: is through. This is unnecessary; printString is implemented to use printOn: which takes a stream as a parameter. To accomplish the same task without creating the unwanted string, ask the object to do it:

```
anObject printOn: myStream
```

Whenever practical, use transformations like these on your code to avoid intermediate objects.

Pattern 3: Use cascading to increase readability

Problem: When one object is being sent a series of messages, how can I format my source code to make this obvious to the reader?

Context: A message expression has at least two parts: the message and the receiver. Thus to understand an expression, the reader must digest not only what the message is but what object it's being sent to. When multiple messages are being sent to the same object, it simplifies the reader's understanding to explicitly show that all of these messages are being sent to the same object. That way, the reader need only determine the receiver once, and can then concentrate on the messages being sent.

Separate code statements are divided by periods. Each is usually placed on a separate line to clearly show the reader that they are separate statements. When multiple statements are appended together into a single sequence, it is tempting to place them all on the same line as one statement. This, however, makes it difficult for the reader to recognize that the statement is really a series of separate sub-statements.

If a substatement starts in the first column of a line, it is difficult for the reader to recognize that this is a sub-statement (a continuation from the previous line) and not a complete statement.

Solution: Use message cascading to send multiple messages to the same object. Cascading will explicitly show the reader that all of the following messages are being sent to the same receiver.

Try to avoid interrupting the cascade to send a message to another object. The more pieces you break the cascade into, the less helpful it will be to the reader.

Format a cascade to indicate to the reader that this is a cas-

cade. Put the receiver on the first line in the first column. Then put each sub-statement sent to the receiver on its own line, indented a set amount from the first column (such as one tab).

Cascading won't make your code any more efficient, but it will make it easier to read.

Examples

Example 1: This code is typical for creating a new object:

```
| layout |
...
layout := LayoutFrame new.
layout leftFraction: 0 offset: 10.
layout topFraction: 0.1.
layout rightFraction: 1 offset: -10.
layout bottomFraction: 0.9.
...
```

To create the same object the same way, but make the code easier to read, use cascading:

```
| layout |
...
(layout := LayoutFrame new)
    leftFraction: 0 offset: 10;
    topFraction: 0.1;
    rightFraction: 1 offset: -10;
    bottomFraction: 0.9.
...
```

The cascading shows the reader more clearly that all four messages are being sent to the same object.

Example 2: Be careful not to assume that all messages return self; many don't, and they could cause you to set your variables incorrectly. This code:

```
^ (Set new)
    add: 1;
    add: 2
```

will return 2, not a Set; use the message yourself to fix this problem:

```
^ (Set new)
    add: 1;
    add: 2;
    yourself
```

Example 3: Avoid writing code that interrupts the cascade. The code:

```
writeStream nextPutAll: 'My class has'.
self class subclasses size printOn: writeStream.
writeStream nextPutAll: 'subclasses.'.
```

can be written to use cascading without interruption as:

```
writeStream
    nextPutAll: 'My class had';
    print: self class subclasses size;
    nextPutAll: ' subclasses.'.
```

References

1. Alexander, C., *et al.* A PATTERN LANGUAGE, Oxford University Press, New York, 1977.
2. Beck, K, Patterns and software development, DR. DOBB'S JOURNAL, 19(2): 18–20 and 22, Feb. 1994.
3. Coplien, J.O., Pattern languages for organization and process, OBJECT MAGAZINE, 4(4): 46–51, July/Aug. 1994.
4. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1994.
5. Coad P., D. North, and M. Mayfield, OBJECT MODELS: STRATEGIES, PATTERNS, AND APPLICATIONS, Prentice Hall, forthcoming.
6. The Hillside Group, PATTERN LANGUAGES OF PROGRAMMING, Addison-Wesley, Reading, MA, forthcoming.

Further Reading

7. Alexander, C. THE TIMELESS WAY OF BUILDING, Oxford University Press, New York, 1979.
8. Johnson, R. E. Documenting frameworks using patterns, OOPSLA '92 CONFERENCE PROCEEDINGS, The Association for Computing Machinery, New York, 1992, 63–76.
9. Gabriel, R.P. The bead game, rugs, and beauty, JOURNAL OF OBJECT ORIENTED PROGRAMMING, Part 1—7(3): 74–78, June 1994 , and Part 2—7(5):44–49, Sept. 1994.
10. Beck, K. A short introduction to pattern language, SMALLTALK REPORT, 2(5): 17-18.

Bobby Woolf is a member of the Technical Staff at Knowledge Systems Corp., where he is developing patterns and pattern languages on topics such as VisualWorks frameworks and Smalltalk configuration management. He also participated in PLoP '94, the first annual Pattern Languages of Programs conference. Comments are welcome at woolf@acm.org.

Processes

Alec Sharp and Dave Farmer

THIS ARTICLE TALKS about Smalltalk processes, the threadsafeness of shared resources, and communication between processes. Because the implementation of processes varies quite a bit between Smalltalk vendors, we want to note up front that in this article we are describing VisualWorks 2.0 from ParcPlace Systems.

CREATING PROCESSES

Smalltalk allows you to create separate processes so that your application can do several things in parallel. For example, our application creates a process for handling input from sockets, another process for handling output to sockets, and separate processes to handle I/O to each robot tape library that is connected to our UNIX server. These processes all run in a single Smalltalk image.

The Smalltalk image is a single process being run by the operating system, but internal to Smalltalk is another process scheduler that allocates time among the various Smalltalk processes. So the operating system scheduler allocates time to Smalltalk, and the Smalltalk scheduler allocates time to the various Smalltalk processes.

Smalltalk processes can be forked at different priorities, with higher priority processes being given preferential treatment if they have anything to do. To fork a process, you send a fork or forkAt: message to a BlockClosure. For example,

```
[SocketInput new start] forkAt: Processor userSchedulingPriority.
```

When assigning priorities, it's a good idea to use names to avoid problems when values change in new software releases. For example, in VisualWorks 1.0, user background priority was 3, but in VisualWorks 2.0 the number of priorities has been significantly increased and user background priority is now 30. The priority names can be found in the "priority names" instance protocol of ProcessScheduler.

You can also create a process that does not immediately run, using the newProcess or newProcessWithArguments: message to a BlockClosure. A process created this way does not run until you send it a resume message. (Interestingly, the fork message is actually implemented as a newProcess message followed by a resume.) We won't go into this aspect of processes except to say that you might use this capability if you wanted to gain more control over process scheduling.

PROCESS SCHEDULING

VisualWorks does not have a preemptive scheduler, which means that a process will continue execution until either it explicitly gives up control, using Processor yield, or it does an operation that yields the processor, such as reading a file or waiting. So, for example, in the following code, process1 will never give up control and so process2 never runs. In fact, if we had used fork to create the new process, it would have inherited the priority of the creating process, and would have never given up control to the parent, so the parent would not be able to terminate it. Unfortunately, you won't be able to terminate it with ctrl-C; try it and see!

```
| process1 process2 |
process1 := [[Transcript show: '1 '] repeat]
    forkAt: Processor userBackgroundPriority.
process2 := [[Transcript show: '2 '] repeat]
    forkAt: Processor userBackgroundPriority.
(Delay forSeconds: 7) wait.
process1 terminate.
process2 terminate.

Results: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ....
```

In these examples we don't want the processes to run forever, so we terminate them after seven seconds. We have shown all the code above, but in future examples we will only show the process1 and process2 code to save space. We will also use a tighter formatting than we would use in production code. Another thing to note is our use of the Delay class. Every time we want to wait, we create an instance of Delay then immediately ask the instance to wait. In a production system, it might be more appropriate to create the instance in a separate operation from the wait, especially if the wait occurs inside a loop. For example,

```
delay := Delay forMilliseconds: 100.
[...
delay wait] repeat.
```

In the next example, we yield the processor and now process2 gets a chance to run. Similarly, we could have done an operation that caused a wait, such as (Delay forMilliseconds: 10) wait, and this would have the same result:

```
process1 := [[Transcript show: '1 '. Processor yield] repeat]
            forkAt: Processor userBackgroundPriority.
process2 := [[Transcript show: '2 '.
             Processor yield] repeat]
                forkAt: Processor userBackgroundPriority.

Results: 1 2 1 2 1 2 1 2 1 2 1 2 1 2 .....
```

Let's now give process2 a higher priority (userSchedulingPriority) than process1. Even though process2

does a Processor yield, process1 is never scheduled after process2 gets into the picture because process2 always has work to do. This is unlike timesharing processes in a UNIX system, where each process in effect has two priorities: a base priority and a current priority. The UNIX process scheduler computes the current priority from the base priority, how much time the process has been sleeping, how much CPU time it has used, and other factors. This way, all processes have a chance to run, even if they have a low base priority. Smalltalk processes are more like UNIX real-time processes, where the highest priority process always gets the CPU if it has something to do.

```
process1 := [[Transcript show: '1 '] repeat]
            forkAt: Processor userBackgroundPriority.
(Delay forMilliseconds: 100) wait.
process2 := [[Transcript show: '2 '. Processor yield] repeat]
            forkAt: Processor userSchedulingPriority.

Results: 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 .....
```

Let's take a brief look at how VisualWorks itself uses some of the different priorities; we'll specify the priority by the message that you send to Processor. The incremental garbage collector runs at systemBackgroundPriority, so it only gets activated if there is nothing else going on. Once running, if it decides that memory needs compacting, if forks a process to do so at userInterruptPriority, which is a higher priority than the typical user application running at userSchedulingPriority. The Profiler also runs at userInterruptPriority, since it needs to periodically interrupt the application it is profiling.

Keyboard and mouse input are done at a higher priority still, lowIOPriority, as is the process that handles low space conditions. An example of a highIOPriority process is a C routine calling back into Smalltalk. The highest priority, timingPriority, is used by system processes that handle delays and process termination.

TERMINATING PROCESSES

Once a process has been forked, how does it terminate? There are two ways this can happen. It can simply finish what it was doing, or it can be terminated. This example shows a process finishing up its job then terminating. We print out the value of the process twice, once while it's still doing work, and again after it's finished and the garbage collector has done its thing.

```
proc := [(Delay forSeconds: 1) wait.] fork.
Transcript cr; show: proc printString.
(Delay forSeconds: 2) wait.
ObjectMemory garbageCollect.
Transcript cr; show: proc printString.

Results:    a Process in [] optimized
            a Process in nil
```

To terminate a process, we send it a terminate message. Generally this message will be sent by another process, but there's no reason why a process can't send itself a terminate. Of course the process will need a handle to itself if it wants to send a terminate message to itself. Generally you should be able to

structure the code that is executed in a forked block to simply finish, but it's certainly possible that the termination condition may be buried deep in your code, and rather than filtering up the condition it's easier to terminate the process when the condition is found (alternatively, you could raise an exception). A process could also terminate itself by sending the terminate message to the active process (i.e., itself):

```
Processor activeProcess terminate
```

The example that follows shows a process being terminated by another process. The main difference between this example and the previous one is that in line one, the process waits for ten seconds, then in line three we terminate the process. The Transcript shows that the process is nil long before the 10 seconds are up.

```
proc := [(Delay forSeconds: 10) wait.] fork.
Transcript cr; show: proc printString.
proc terminate.
(Delay forSeconds: 1) wait.
ObjectMemory garbageCollect.
Transcript cr; show: proc printString.

Results:    a Process in [] optimized
            a Process in nil
```

SHARED RESOURCES

Sometimes we have resources that the various processes need shared access to. For example, in our application, we log information from the various processes and we need to make sure that we don't get interleaved data. We also keep a ThingsToCleanUp object in a pool dictionary, in which we store all the opened files and external devices, and the forked processes. We want to make sure that we provide threadsafe access to these shared resources. If we don't make access to shared resources threadsafe, we could end up in the situation illustrated by the following example.

```
array := #(1 2 3 4 5 6 7) copy.
process1 := [array do: [:element | Transcript show: element printString, ' '.
    (Delay forMilliseconds: 500) wait ] ] fork.
(Delay forMilliseconds: 1000) wait.
process2 := [array at: 6 put: nil.
    Transcript show: '<Setting 6=nil> '] fork.

Results: 1 2 <setting 6=nil> 3 4 5 nil 7
```

We want to protect the array so that only one process can access it at a time. We do this with a mutual exclusion semaphore, which we create by sending the forMutualExclusion message to Semaphore. We ask the semaphore to run the code by sending it the critical: message with the block of code to run, and the semaphore is smart enough to only run one block of code at a time.

```
array := #(1 2 3 4 5 6 7 ) copy.
sem := Semaphore forMutualExclusion.
process1 := [sem critical:
        [array do: [:element | Transcript show: element printString, ' '.
        (Delay forMilliseconds: 500) wait]]] fork.
(Delay forMilliseconds: 1000) wait.
process2 := [sem critical:
        [array at: 6 put: nil.
        Transcript show: '<setting 6=nil> ']] fork.

Results: 1 2 3 4 5 6 7 <setting 6=nil>
```

As a brief aside, Semaphores work by having processes wait until a signal is sent to the semaphore. The mutual exclusion semaphore sends itself a signal when it's created, so that the first block of code to be run by the semaphore already has a signal waiting. That is, it doesn't have to wait. Once the code has been executed, the semaphore sends itself another signal, priming itself in advance for the next code block. It does so by:

```
^mutuallyExcludedBlock valueNowOrOnUnwindDo: [self signal]
```

How do the priorities of the different processes affect mutual exclusion? Fortunately, mutual exclusion works as you'd want it to work, regardless of priority. If we change the previous example so that process1 is forked with forkAt: Processor userBackgroundPriority and process2 is forked with forkAt: Processor userSchedulingPriority, we get the same results. The critical block is still run to completion before the higher priority process can get access to the shared resource.

The next question is can another process get access to a shared resource if it's not cooperating by sending the critical: message? As the following example shows, the answer is yes:

```
array := #(1 2 3 4 5 6 7 ) copy.
sem := Semaphore forMutualExclusion.
process1 := [sem critical: [array do: [:element |
        Transcript show: element printString, ' '.
        (Delay forMilliseconds: 500) wait]]] fork.
(Delay forMilliseconds: 1000) wait.
process2 := [array at: 6 put: nil.
        Transcript show: '<setting 6=nil> '] fork.

Results: 1 2 <setting 6=nil> 3 4 5 nil 7
```

So, to protect shared resources, the processes must cooperate. Both processes have to agree to use the same semaphore to protect the shared resource. Let's go ahead and implement access to a shared resource, a Dictionary, as we might do in a real application. We will create and initialize the object, then provide read, write, and delete access to the resource. Our first decision is whether to subclass off Dictionary or create a new class that has a Dictionary as an instance variable. Since we want to restrict access to just a few messages, it's easier to create a new class than worry about all the possible ways someone might try to access a subclass of Dictionary. So, we'll create a new class with two instance variables, collection and accessProtect:

```
new
    ^super new initialize

initialize
    collection := Dictionary new.
    accessProtect := Semaphore forMutualExclusion.

at: aKey put: anItem
    ^accessProtect critical: [collection at: aKey put: anItem]

at: aKey
```

```
    ^accessProtect critical: [collection at: aKey ifAbsent: [nil]

remove: aKey
    ^accessProtect critical: [collection removeKey: aKey ifAbsent: [nil]]
```

Having got this far, we now need to say that the Transcript is not threadsafe. It so happens that all our examples work in VisualWorks 2.0, but writing to the Transcript from multiple processes is not guaranteed to work correctly. In fact we have an innocuous looking Transcript example that in VisualWorks 1.0 hangs until you press ctrl-C. So, while we use the Transcript in our examples, we don't recommend writing to it from multiple processes in production code. Much of the time, code that is not threadsafe will work because the Smalltalk scheduler is non-preemptive and so many code segments will run to completion. However, if you ever add code that causes the process to give up control, you may find that your code no longer works correctly.

INTERRUPTING ANOTHER PROCESS

Now, suppose you want to ask a particular process about its state. Perhaps you want to know if it's waiting for a particular input, or whether it's finished some part of its processing. In our product, where we have separate processes handling different robot tape libraries, we sometimes want to know the status of the library; for example, if it's on-line or off-line. There are several ways to handle this desire for information.

One solution might be to restructure your application so you don't need access to this information, but we'll ignore this one because it's not very interesting to this article! Another solution would be to have the process post the needed information in a shared resource, protected by a mutual exclusion semaphore. This has the potential disadvantage that the process may be updating the shared resource with a lot of information, but perhaps no one is reading it very often.

Another approach would be to send an object to the process using a shared queue and have the object figure out the information then send it back on another shared queue. We'll talk more about shared queues later, but a disadvantage of the shared queue approach is that the process needing the information will usually have to wait until the process can get to the shared queue, pull the object off it and process it. It's not an approach to use if you are in a hurry.

The approach we are going to look at is one where you can actually interrupt a process and ask it to do something for you. The mechanism is to send an interruptWith: [aBlock] message to the process, passing as a parameter the block of code you want executed. The process saves its context, executes the passed-in block, restores its context, then resumes its business. Here's an example. Process1 is simply waiting for time to pass before doing anything. We interrupt it and ask it to print something.

```
process1 := [(Delay forSeconds: 4) wait.
        Transcript cr; show: 'process1 done waiting'] fork.
process2 := [(Delay forMilliseconds: 100) wait.
        process1 interruptWith:
            [Transcript cr; show: 'process2 interrupt']] fork.
```



```
Results:    process2 interrupt
        process1 done waiting
```

That's all well and good, but what happens if the process is doing something that it really doesn't want interrupted? Fortunately, there's a way to prevent interrupts, which is to protect the special block of code with a valueUninterruptably message. The valueUninterruptably method sends the active process an uninterruptablyDo:[aBlock] message.

uninterruptablyDo: takes the parameter block and asks a semaphore named interruptProtect to run the block in critical mode. interruptWith: also asks interruptProtect to run its block in critical mode. Since valueUninterruptably and interruptWith: both ask the same semaphore to run their blocks critically, only one of the code blocks executes at a time.

Here's the previous example with process1 protecting its work against interruption:

```
process1 := [[(Delay forSeconds: 4) wait.
    Transcript cr; show: 'process1 done waiting']
        valueUninterruptably] fork.
process2 := [(Delay forMilliseconds: 100) wait.
    process1 interruptWith:
        [Transcript cr; show: 'process2 interrupt']] fork.


Results:    process1 done waiting
            process2 interrupt
```

Are the interruptWith: and valueUninterruptably messages ones that you should use? Our view is to use them if you have to, but use them sparingly. ParcPlace recommends against their use.

The method comments for valueUninterruptably and uninterruptablyDo: both say "Use this facility VERY sparingly." One problem with running a process uninterruptably is that you can't even use ctrl-C to interrupt it should things go wrong. Another is that if a process running uninterruptedly does something time consuming, such as reading a file, no one else can get the processor during that time. The only classes that send valueUninterruptably are Profiler and SharedQueue. ControlManager and Process are the only classes that send interruptWith:.

SHARED QUEUES

Our main objective in talking about interruptWith: and valueUninterruptably is to illustrate some interesting capabilities, then let this lead to a discussion of SharedQueues. So here we are. SharedQueues are the general mechanism for communicating between processes. They contain an OrderedCollection so that all objects that go onto a shared queue are taken off in chronological order. To set up communication between processes, you create an instance of SharedQueue and tell both processes about it. One process will put objects on the shared queue using nextPut: and the other process will use next to get objects from the queue. When a process sends the next message, it blocks until there is something on the queue. If the process doesn't want to block it can send isEmpty or peek.

Because shared queues are so important for communicating between processes, they need to be as safe as possible. For this reason, all access to shared queues is protected by a mutual exclusion semaphore using the critical: message, and this block of code is protected by a valueUninterruptably message. For example, here's how ParcPlace implements the size message to a shared queue.

```
size
    ^[accessProtect critical: [contents size]] valueUninterruptably
```

Again, the critical: message makes sure that only one operation happens at a time, so for example, it makes sure that one process is not getting an object from the queue while another process is adding an object. The valueUninterruptably makes sure that the shared queue operations can't be interrupted by a process sending an interruptWith: message.

Here's an example of shared queues in use. Process2 prints the number and puts it on the shared queue, and process1 reads the queue and prints the number:

```
sharedQueue := SharedQueue new.
process1 := [[number := sharedQueue next.
    Transcript show: ` R', number printString] repeat] fork.
process2 := [1 to: 5 do: [:index |
        Transcript show: ` W', index printString.
        sharedQueue nextPut: index.
        (Delay forMilliseconds: 500) wait]] fork.
Results: W1 R1 W2 R2 W3 R3 W4 R4 W5 R5
```

Try this again after removing the Delay in process2. Because process2 now always has something to do, it does not give up control and so process1 waits for the processor until process2 is completely finished. The Transcript output now looks like:

```
Results: W1 W2 W3 W4 W5 R1 R2 R3 R4 R5
```

OUR PRODUCT

In our product we make heavy use of processes and therefore of shared queues (See Fig. 1). We have one process that does nothing more than block on a socket waiting for input. It puts the input on a shared queue and another process takes it off. This second process sends each object a queueYourself message, telling the object to put itself on the appropriate shared queue for the robot tape library that the request is going to. Each library controller blocks on its own shared queue, waiting for a request to process. Finally, after the request has done what it needs to do, a response is created and put on an output shared queue. The output process gets response objects from this queue and sends them out over a socket to the appropriate UNIX process. Because Smalltalk gives a process control while it has things to do, we put a Processor yield after each shared queue nextPut:. This gives each process the opportunity to run, even when other processes have more they could be doing.

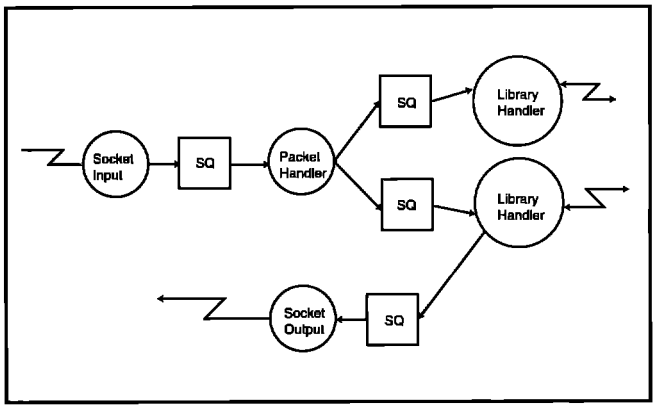


Figure 1.

There is actually a lot more going on than this, and to solve our specific problems we created a subclass of SharedQueue, which we call a PrioritySharedQueue. Rather than keeping objects in chronological order in the shared queue, it orders them by priority then time. It also has methods to search for specific types of object and to delete objects. However, that's a story for another day. This just about wraps the article up, but before we leave, we'd like to mention briefly a new class that appeared in VisualWorks 2.0 that makes use of processes.

PROMISES

VisualWorks 2.0 introduces a new class, the Promise class. An instance of Promise promises to do something for you while you go off and do other things. It does this by forking a new process to carry out the work. You create the promise by sending the promise or promiseAt: message to a BlockClosure. Once the promise has been created, you can query it for its value (if the promise has been kept), or to find out if it has a value (it may still be doing its work.) In fact, promises are a little more complex than this because if the promise fails or terminates, an exception is raised, so to be robust, you should wrap the

continued on page 29

MathPack/V

David Buck

A few months ago, a friend came to me with an engineering problem. He needed to calculate some strange formulas having to do with electromagnetic interference. He tried using calculators and spreadsheets without success. In desperation, he asked me if there's anything I could do to help. "Sure," I said. "Smalltalk is a great system for crunching through formulas like that." Well, at least it's better than a spreadsheet.

He came over and showed me the formulas. Some formulas involved matrix algebra. I was fortunate enough to have written a matrix class in Smalltalk when I was doing my master's degree, so we used that. The formulas also involved complex numbers. Well, complex numbers can't be all that hard, can they? We whipped up a complex number class. Wait a minute, we had to take *powers* of complex numbers. How do you do that? We pulled out some dusty old math textbooks and looked up the formulas. We found that you apparently have to convert the complex numbers to polar coordinates, raise them to a power, and convert them back into complex numbers. OK, we typed the formulas into Smalltalk. We then tried out the equations and got some really strange results. There was a bug in the method that converts from complex numbers to polar coordinates. We had to worry about the sign of the result, which we weren't handling properly. We fixed the bug and continued on.

We finally managed to get an array of the answers, and my friend said, "OK, can you plot these?" It sounded like an easy request, but Smalltalk has no built in plotting functions. To plot a graph, I'd have to open a GraphPane, scale the numbers to the proper range, and issue the place: and line: messages to draw the graph. This was too much work for plotting about 20 points. We gave up and sketched it on graph paper. After spending about 10 hours on it, we ended up with results that we couldn't trust because we didn't know if all the steps in between were completely bug-free, and any small bug would dramatically change the results. I started thinking that Smalltalk wasn't such a great environment for this after all.

ENTER MATHPACK/V

Since that time, I've found a mathematics package by GSoft called MathPack/V. It's a mathematics package for Smalltalk/V Win16 and Win32 (the package is available for ObjectWorks Smalltalk). I now realize that if I'd had this package at the time, the task of performing all those calculations would have been trivial. MathPath has facilities for performing Matrix and Vector calculations that are more complete and more general than the ones I had implemented myself. It also has classes for Complex numbers and Polar coordinates, and they perform the raisedTo: operation correctly. When it comes to plotting the results, MathPack has an excellent set of plotting classes that let you plot multiple values in 2D or 3D simply and easily. In fact, MathPack can handle almost any numerical operation I've ever wanted to perform and even a bunch that I've never heard of. When you combine these facilities with the Smalltalk/V programming environment, you get a truly astounding mathematical workbench for solving almost any math problem you can come up with.

SYMBOLIC MATH

To start off, MathPack performs many of its operations symbolically instead of numerically. What does this mean? Well, let's take an example. In Smalltalk, if you type in

```
5 sqrt
```

you get the answer: 2.23606798. With MathPack installed, you get the answer: V5 (meaning the square root of 5). In other words, MathPack returns you a square root object. This answer is actually more accurate than the one that Smalltalk normally provides. In fact, if you run

```
5 sqrt squared
```

The answer will be the integer 5 (precisely!). The limits of this ability actually surprised me. When I tried

```
(PI / 3) sin
```

I got sin(PI/3) as the answer. Notice that in MathPack, there's an object for pi. This isn't just a global variable that contains the number 3.1415926... but rather a symbolic value that represents pi precisely. But we can go one step further:

```
(PI / 3) sin simplify
```

This gives the answer 1/2V3 (meaning one half the square root of 3).

MathPath also allows you to include variables in your equations; not Smalltalk variables but symbolic mathematical variables. The variables *X*, *Y*, *Z*, *R*, and *T* are defined for you in MathPack, but you can create your own if you wish. For example:

```
myPoly := ((X**3) - (X**2 * 4) + (X*5) + 7).
```

creates a formula representing the polynomial x3 + 4x2 + 5x + 7. You can then evaluate this formula by sending it a value: message:

```
myPoly value: 3      ==> 13
```

More simply, I could have used a polynomial like this:

```
myPoly := #(7 5 -4 1) asPolynomial
```

or this:

```
myPoly := #(1 -4 5 7) asReversePolynomial
```

Because this polynomial is stored symbolically, you can do some more intelligent things to it. For example, let's find the roots of this polynomial:

```
myPoly solve
===>  Bag((2.34372593+1.65207333i) (2.34372593-1.65207333i) -
0.68745186 )
```

MathPack found all three roots of my polynomial. The first two roots are complex; the last root is real.

Do you want to try some calculus? Try this:

```
myPoly der
===> 3x^2-8x+5
```

This gives us the derivative of the polynomial. In fact, you can calculate symbolic derivatives of any function. For example:

```
((X**2) sin + (X**2) cos) der: X   "Derivative with respect to X"
===> (2*cos(X**2)*X-2*sin(X**2)*X)
```

Unfortunately, symbolic integration is more complex. MathPack can symbolically integrate polynomials, some trigonometric functions, and some exponentiation functions, but symbolically integrating arbitrary functions is mathematically impossible. If you need to, however, you can numerically integrate any function using the Romberg method provided by MathPack.

PLOTTING FUNCTIONS

Plotting graphs is a breeze with MathPack. Suppose we want to plot the values of the polynomial -x3 + 3x2-5x+7 from -10 to +10. Just type the following code into a workspace and run it.

```
| poly results |
poly := #(-1 3 -5 7) asReversePolynomial.
results := OrderedCollection new.
-10 to: 10 do: [:i |
results add: i@(poly value: i)].
```

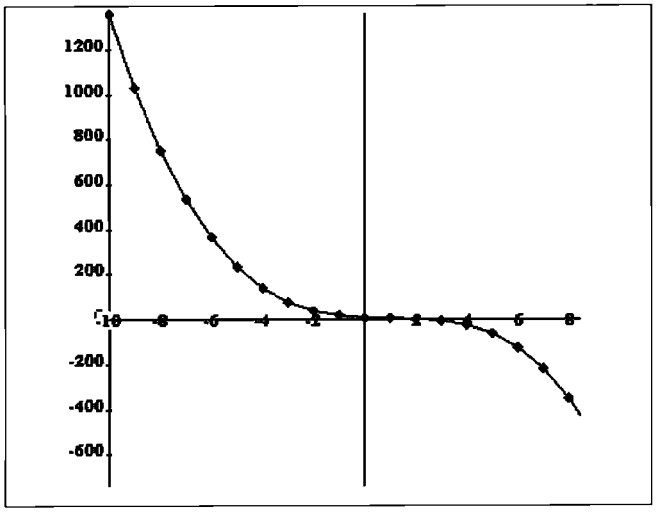


Figure 1. 2D plot of a cubic function.

```
Plot2D new
    axes: true;
    vectors: results;
    points: results symbol: #filledDiamond color: ClrRed;
    tickX: 2;
    tickY: 200;
    display
```

This produces the plot shown in Figure 1.

The first few lines of this code simply evaluate the polynomial at the desired points and collect the results into an OrderedCollection. You can use any technique you want to collect the values to plot. To plot the values, simply create a Plot2D object and set it up. The axis: message indicates that I want the *X* and *Y* axes shown. The vectors: message indicates that I want the points connected by lines. The points:symbol:color: message indicates that I want individual points plotted with the given graphical symbol (a filled diamond in this case) with the given color. You could use circles, crosses, squares, or triangles instead of diamonds. Finally, I indicate that I want tick marks on the X and Y axes and then display the plot. If you don't want ticks or axes shown, you can leave out the corresponding lines. If you want, you can add legends, change

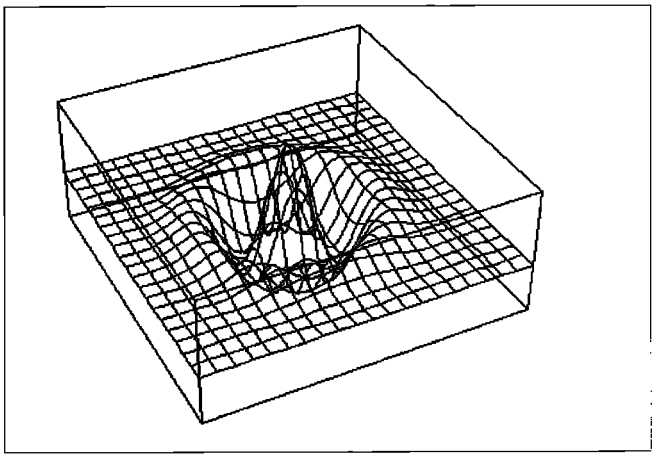


Figure 2. A 3D surface plot.

line styles, and use splines instead of straight lines to plot the graphs.

In addition to simple 2D plots, you can make 3D surface plots. The code below produced the image shown in Figure 2.

```
| aPlot |
aPlot := Plot3D new.
(((X*X)+(Y*Y)) sqrt cos * ((X*X) + (Y*Y) * -0.05) exp * 5)
    xyzPlot: (-10@-10 corner: 10@10)
    viewPoint: 30@60@50
    sectors: 20
    on: aPlot.
aPlot display
```

In addition to functions of two variables, MathPack has a number of full 3D geometric objects that can be rotated, translated, and plotted on the screen. The following piece of Smalltalk code creates a cone with an elliptical base,

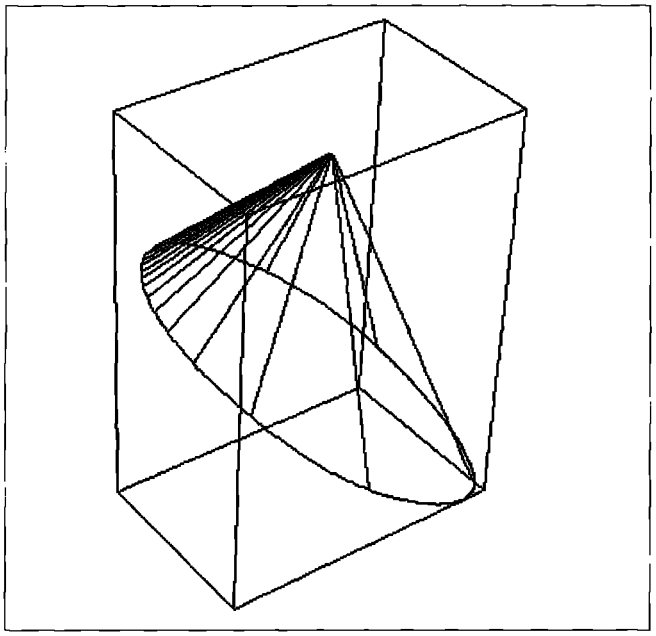


Figure 3. A 3D cone rotated and plotted.

rotates it about three axes, and plots it on the screen. The result is shown in Figure 3.

```
|aPlot |
aPlot := Plot3D new.
Cone new
    baseCurve: ((PolarConic e: 0.9 k: 2.0) loLim: 0.0; hiLim: 6.28);
    apex: 0.0@0.0@10;
    rotateWithRoll: 0.3 pitch: 0.7 yaw: 0.5;
    plotFromViewPoint: 40@60@50 sectors: 20 center: 0@0@0 on:
aPlot.
aPlot display.
```

Three dimensional figures supported by MathPack include 3D points, lines, and curves as well as boxes, cones, pyramids, cylinders, ellipsoids, spheres, planes, revolving curves, and toruses. You can then combine these basic figures together into composite objects to model more complex 3D objects. MathPack, however, isn't intended to be a 3D modeling program. The plots of these shapes are wireframe only, without hidden surface removal. If you need to perform sophisticated 3D modeling and rendering, you should look into a package that is better tailored to it or be prepared to implement your own in Smalltalk.

MATRIX ALGEBRA

I've spent quite a lot of time writing 3D graphics software, so I can really appreciate the matrix and vector facilities of MathPack. The Matrix classes provide virtually all the functionality I've ever needed from matrices and more. Creating matrices couldn't be easier. You can simply say the following:

```
#((2 4 3) (5 -3 2) (7 1 9)) asMatrix
===>  |2  5 7 |
      |4 -3 1 |
      |3  2 9 |
```

The asMatrix message interprets each subarray as a column of the matrix. If you want to treat them as rows, you can use asRowMajor instead. Now that you have a matrix, you can perform normal operations such as addition, subtraction, multiplication, and division. As you would expect, there are also methods to access individual elements in the matrix and to return row vectors and column vectors from the matrix.

Using matrices, you can solve a system of simultaneous linear equations. There are two ways of doing this. The most efficient way is to send the solve: message to the matrix passing in the vector to solve for. For example, suppose you know that

```
2x + 5y - 3z = -29
x -  2y + 4z = 32
-x + 3y - 2z = -23
```

What values of x, y, and z satisfy these conditions? Well, just type this into MathPack:

```
#((2 5 -3) (1 -2 4) (-1 3 -2)) asRowMajor solve: #(-29 32 -23)
===>  (2 -3 6)
```

So, $x = 2$, $y = -3$, and $z = 6$ satisfy all three of the above equations.

The other way to solve these equations is to multiply the result vector by the inverse of the matrix:

```
#((2 5 -3) (1 -2 4) (-1 3 -2)) asRowMajor invert * #(-29 32 -23)
asVector
===>  | 2 |
      |-3 |
      | 6 |
```

With square matrices you can calculate eigenvalues and eigenvectors using MathPack. Other useful operations include matrix transposition, LU decomposition, and pseudoinverses.

MathPack/V

The matrix approach to solving systems of equations is fine if the equations are linear. If you want to solve nonlinear equations or systems of inequations, you can use the SimultaneousEquations or the SystemOfInequalities classes. To use these classes, you must provide the equations symbolically. MathPack can then use the Newton–Raphson technique to solve the equations.

DIFFERENTIAL EQUATIONS

Do you need to calculate the path of a space ship navigating through a trinary star system? Well, maybe we'll just worry about hitting a target 75 meters away with your bow and arrow. Both of these problems require a technique called "numerical differential equation solving."

MathPack can numerically solve differential equations using a technique known as fourth order Runge–Kutta. In a nut shell, it means that this is a stable and accurate technique for doing this sort of work. Many simple programs use a technique called Euler's method, which is to add a bit of the acceleration to the velocity and to add a bit of the velocity to the position on each step. Euler's method works well in simple situations (like the arrow example above), but more involved calculations require a better system like Runge–Kutta.

There are faster and more sophisticated techniques for solving ODEs that MathPack doesn't provide, but these techniques don't handle the tough parts as well as Runge–Kutta. My only regret is that the implementation of Runge–Kutta provided by MathPath isn't adaptive. This feature would allow the algorithm to take small steps over the rough terrain while taking long strides over the easy parts.

STATISTICS AND OTHER STUFF

MathPack includes a huge array of statistical facilities. Well, at least it's huge from my perspective, because I don't often need to use statistics. I find that a simple mean supports most of my statistical needs. But for those who need more, MathPack has it. You get Chi-square tests, one-way and two-way analysis of variance, linear and polynomial regression, generalized least squares fit, and nonlinear least squares fit. There's also a random number generator that can generate uniform and Gaussian random numbers.

In the "other stuff" category, there's a class for performing digital signal processing functions. The most commonly used function is the Fast Fourier Transform, but also included are Cos and Sin transforms, convolutions and deconvolutions, correlation of data sets, and spectral analysis.

In some unrelated other stuff, there's an interesting new kind of Number in MathPack. It's a decimal fraction. It can represent decimal numbers with any desired degree of accuracy. For example, if you want to calculate pi to 20 digits, you can type

```
PI asDecimalFraction: 20
```

Any Integer, Float, or Fraction can be converted into a decimal fraction.

QUIRKS AND QUIBBLES

MathPack has never given me a wrong answer. It has, however, given me answers that need to be interpreted carefully. For example, in the following equation, I'm trying to calculate the derivative of pi*x2. Here's MathPack's answer:

```
PI * (X ** 2) der: X
===> (2*X*PI+X**2*PI')
```

The real answer should be 2*X*PI. Is MathPack wrong? Not really. In the second part, X**2*PI' is zero because PI' is 0. The problem here seems to be that PI isn't really known as a numeric constant, so MathPack doesn't know how to differentiate it. It blindly uses the chain rule and puts an apostrophe to indicate that the PI needs to be differentiated but MathPack doesn't know how to do it. (Actually, I was quite impressed that it worked this well.)

A more serious problem is the way that MathPack hooks itself into the existing Smalltalk system. It's certainly convenient for 5 sqrt to give you back a square-root object, but if you're going to alter existing methods like this, you have to be extremely careful. There are some messages that Numbers understand that Root objects don't. For example, if you try running "5 sqrt rounded" in MathPath, you'll get a walkback window because Root objects don't understand rounded. To fix the problem, you have to use "5 sqrt asFloat rounded".

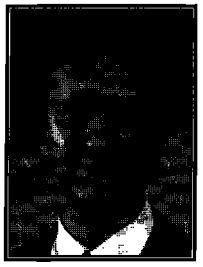
The problem here is that if you have existing code that used to work without MathPack, it may not run after MathPack is installed because MathPack changes the way some system methods work. If you are using MathPack as a mathematical workbench, this issue isn't very serious. If you get a walkback window, you can easily fix the problem and continue. In this way, Smalltalk/V with MathPack becomes a very powerful scientific calculator. This is the ideal environment for MathPack. Using MathPack routines in a delivered application, however, may be risky because you can never accurately predict when the results of a calculation will be numeric or symbolic, and the difference may be critical.

There are some inconsistencies in the system that are, really, more annoying than troublesome. For example, if you have a function and you want to plot it, you can send the function a plotFrom:to:points:type:on: message. For example:

```
(X*X) plotFrom: -10
to: 10
points: 20
type: #vectors
on: aPlot
```

Great. But try replacing "(X*X)" with "#(1 0 0) asReversePolynomial" and it doesn't work. Polynomials don't understand the same messages as functions. To differentiate a polynomial, you send it a der unary message. To differentiate a function, you send it a der: keyword message with

continued on page 29



MARK LORENZ

Architecting large OO projects

MANAGING THE complexity of most commercial OO projects requires planning for and controlling an architecture for your business object model. This involves dividing up your system into subsystems, assigning contracts between the subsystems, and establishing your architects' ownership of the contracts.

Figure 1 shows a partial project architecture along with ownership assignments. Development teams own particular subsystems and are responsible to build these subsystems so that they support the subsystem contracts. These contracts, such as *Maintain inventory levels*, provide a set of public services to the other subsystems. The client subsystem teams treat the server subsystem as a black box, ignoring the complexities inside.

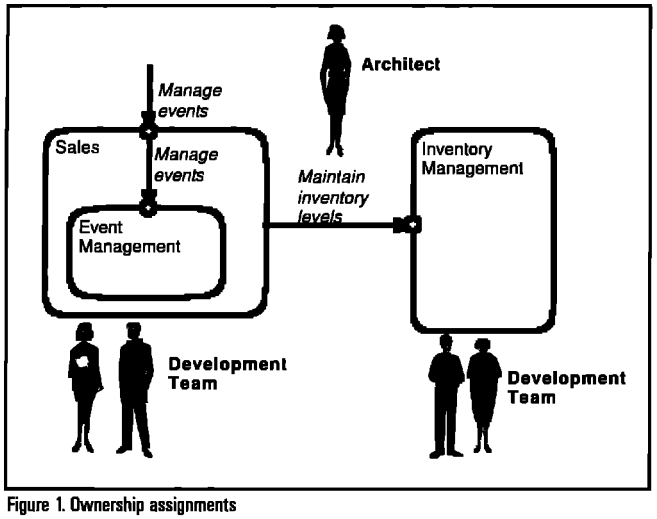


Figure 1. Ownership assignments

This organization allows the different development teams to proceed relatively independently of each other—an essential requirement for large projects.

Note: This discussion is extracted from the author's forthcoming book RAPID SOFTWARE DEVELOPMENT.[3]

Mark Lorenz is founder and president of Hatteras Software, Inc., a company which offers numerous services and products to help other companies use object technology effectively. He welcomes questions and comments via e-mail at 71214.3120@compuserve.com or phonemail at 919.319.3816.

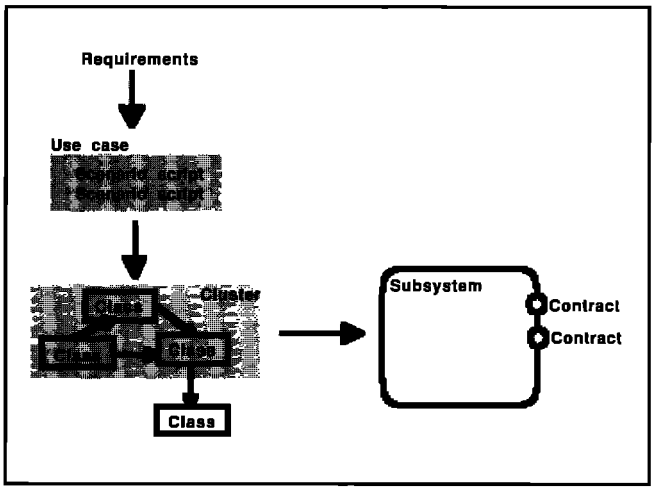


Figure 2. Exploring an architecture for one subsystem.

AN ARCHITECTING PROCESS

So, how does this architecture come about? Many times, projects do not have a good idea of what subsystems exist ahead of time. The subsystems, much like other abstractions such as frameworks and abstract classes, become apparent as the system exploratory process proceeds.

Figure 2 gives an overview of the process for one subsystem:

- Use cases are written for the system requirements.
- Scenario scripts are used as a technique to fill in details of the object model.
- Key classes are clustered into more closely coupled groups, called subsystems.
- Subsystems are assigned public contracts from groupings of key responsibilities of the classes.
- Development teams are assigned ownership of the subsystems. Their focus is on building a subsystem that supports its contracts.
- Architects are assigned ownership of the subsystem contracts. Their focus is on controlling any changes to the subsystem contracts.

Figure 3 shows how the architecture team moves across all subsystems for the system problem domain, working with each

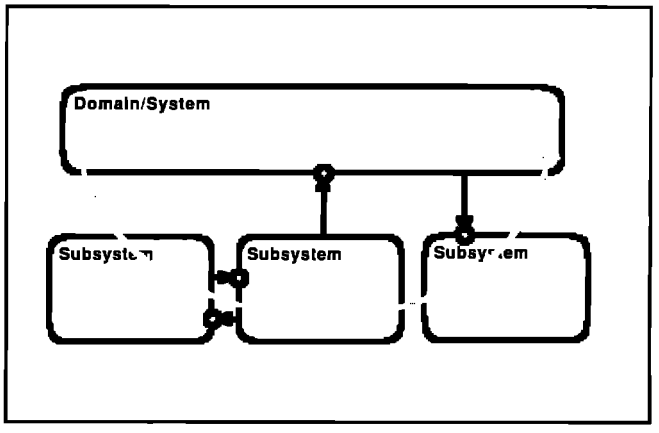


Figure 3. Traversing the system.

Project Practicalities

of the subsystem teams to model their portion of the system at a high level.

The contracts between each of the subsystems that make up the system are discovered during rapid modeling sessions of two to ten days each, depending on the subsystem size. Some questions that help identify subsystem contracts are:

- Why do we have this subsystem?
- What basic services should it provide?
- Does it make sense for this subsystem to provide this service?
- What services does this subsystem need from other subsystems?

Once the rapid modeling session has been completed for one subsystem, its team is free to start iterative development in parallel with other efforts. The development team must negotiate subsystem-level contract changes with the architects, who have the broad, system-wide perspective. The architects will involve affected subsystem owners in the change negotiations.

SUMMARY

We have discussed a proven process for architecting large OO projects. This process is essential for large projects to be able to manage the complexity and communications across the teams. It is also very effective for geographically distributed projects.

TERMINOLOGY

Architect: A person with a broad view of the system's interrelationships that owns the subsystem contracts.

Black box: Viewing something from the outside only, ignoring the internal workings.

Contract: A grouping of public responsibilities that provide services to a subsystem' and/or class' clients.

Key class: A class that is essential to model a particular problem domain.

Script: A time-ordered sequence of message sends through the model to support a functional thread for a use case.

Subsystem: A grouping of more tightly coupled classes and contained subsystems that support one or more contracts.

Use case: A particular usage of the system to support its requirements.

References

1. Jacobson, Ivar, *et al.* OBJECT-ORIENTED SOFTWARE ENGINEERING: A USE CASE DRIVEN APPROACH, Addison-Wesley,Reading, MA, 1992.
2. Lorenz, M. OBJECT-ORIENTED SOFTWARE DEVELOPMENT: A PRACTICAL GUIDE, Prentice Hall, Englewood Cliffs, NJ, 1993.
3. Lorenz, M. RAPID SOFTWARE DEVELOPMENT, SIGS Books, New York, NY, 1995, forthcoming.
4. Wirfs-Brock, R., *et al.* DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice Hall, Englewood Cliffs, NJ, 1990.

Processes

continued from page 22

promise in a handle:do: message. But since we are only mentioning promises in passing, we'll just show a simple example of a promise in action.

```
count := 0.
Transcript cr.
promise := [DialogView confirm: 'Is it true?'] promise.
[promise hasValue]
    whileFalse:
        [Transcript show: count printString, ' '.
        count := count + 1.
        (Delay forSeconds: 1) wait].

Transcript show: promise value printString.
```

Alec Sharp is an Advisory Software Engineer at StorageTek. He is the author of Software Quality and Productivity, published by Van Nostrand Reinhold. He can be reached at alec_sharp@stortek.com.

Dave Farmer is a Senior Software Engineer at StorageTek. He can be reached at david_farmer@stortek.com.

They both work on the UNIX Storage Server software, which manages connections to networked hosts and drives the StorageTek family of robotic tape libraries.

MathPack/V

continued from page 26

the variable to differentiate as a parameter. I can understand the difference (polynomials don't have an explicit variable), but the difference becomes confusing.

Finally, I found that the manual was good when it came to listing the classes and methods but poor in terms of concrete examples. There should be more examples of plotting in both 2D and 3D, differential equations, the statistical functions, and the DSP functions. It's rather tricky trying to figure these out from only the explanations of the methods. There are, however, a number of examples stored as class methods in the MathTest hierarchy that you can refer to for some additional examples.

CONCLUSION

All in all, MathPack is an excellent package for solving serious math problems or just for exploring the mathematical world. The combination of MathPack and Smalltalk makes the symbolic operations very easy to use. It's like having a mathematical workbench at your disposal with a wide variety of power tools ready for you to use. Now, the next time my friend asks me to do some mathematical calculations for him, I'll be ready. ♀

Smalltalk Idioms

continued from page 12

this number by sending Processor the message bytesTenured. I will describe all the available messages in a later article. Here is the script, which formats the number of bytes tenured for display.

```
UpdateBytes
    self setValue: Processor bytesTenured printString , ' bytes'
```

Launch the resulting window. Then go operate your favorite interface. You can watch as objects get promoted. If you are doing an operation which you don't think should create any long lived objects, but lots of bytes are shown as being tenured, you may have a candidate for some tuning. I found drag and drop to be a good example.

This has been a quick introduction to your garbage collector. I will cover what it means in practical terms for the various Smalltalks in future issues. As always, if you have comments or questions, please let me know. I love to hear from you.

The best of comp.lang.smalltalk

continued from page 32

you want, add it to the appropriate class initialization method, and reinitialize the class. Then, you need to create a TextAttributes object based on those CharacterAttributes, figure out any additional parameters you need, add it to the appropriate class initialization method, reinitialize, and call resetViews. This is not appealing to a user accustomed to operating systems with hundreds of fonts to choose from and nice font selection dialogs to do the choosing.

Lots of people have developed their own font selection windows to deal with this problem. Wayne Parrot (parrott@bcm.tmc.edu) not only did it but has also made the code available in the general Smalltalk ftp archives (st.cs.uiuc.edu or mushroom.cs.man.ac.uk).

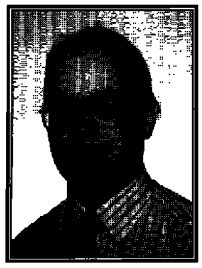
The code is indexed under fontmgr, and it is a simple (about 16 K) file-in for a font editing window. The window allows you to:

- view sample text in an existing text style
- view sample text by incrementally editing a FontDescription
- install a FontDescription as a system text style
- remove system text styles
- reset all views to a specified text style

This is a convenient add-on, and in my limited testing it seemed to work well. The code is for VisualWorks 1.0, but I do not think it would be at all difficult to port to version 2.0, as there have not been many changes in font handling between these versions.

Reference

1. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading, MA, 1994.



ALAN KNIGHT

What's new on the net

IN THE PAST, everything you could do over the network was pretty much limited to a terminal interface. Even though my machine might be running a sophisticated graphical user interface, my network communications used an emulated VT100 terminal. This was particularly true if accessing the network through a modem.

No more. With increasing modem speeds, it is now possible to get reasonable performance with graphically based network applications. Some of the best known of these applications are World Wide Web browsers.

WORLD WIDE WEB

The World Wide Web is a simple concept with remarkable results. It lets people establish *pages* that can contain styled text, pictures, and links to other pages. These other pages are not limited to the same site, but can be anywhere accessible on the network. The difference from old-style applications is amazing. Instead of a VT100 emulation and the ftp program, you suddenly have a graphical HyperText browser that spans the network.

The other amazing thing is the amount of stuff that is out there. It is not like ftp sites, where there are a few large sites that have almost everything you need. Instead, there are enormous numbers of small sites, where people have set up Web pages on topics of interest to them, with links to related sites. You can easily stumble across links to completely unexpected places and spend hours exploring them (I started out looking for Smalltalk-related stuff and ended up browsing a list of vegetarian restaurants in Atlanta). The browsers can also put a prettier face on more conventional net resources like ftp sites and newsgroups.

It is hard to convey how much fun this technology is. I urge you to get hold of a SLIP or other internet connection, find a Web browser, and try it out for yourself. I guarantee you will enjoy it, and you may even find something useful.

I do not know enough about the different products to suggest anything more detailed. Mosaic is the original (and free) Web browser, but there are lots of others around, and there is a rapidly growing range of books and products available to help you get started with the Internet.

Once you are set up, here are a few Smalltalk-related Web pages to get you started on the useful stuff. This certainly is not a comprehensive list, and I expect there will be many new entries by the time this column sees print. Although the names are long and intimidating at first glance, you only need to use them as a starting point. Once you are into the Web, you can get most places just by following links.

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

Jeff McAffer, a PhD student at the University of Tokyo, has set up a page for all things Smalltalk related. It serves as a good starting point for finding other Smalltalk resources. It is accessible as:

http://web.yl.is.s.u-tokyo.ac.jp/members/jeff/smalltalk.html

A list of Smalltalk FAQS is available in HyperText form at:

http://www.cis.ohio-state.edu/hypertext/faq/usenet/smalltalk-faq/faq.html

The University of Illinois Smalltalk archive has a page under construction at:

http://st-www.cs.uiuc.edu

ParcPlace's ParcBench Bulletin Board is also accessible via Gopher (another protocol that is compatible with WWW). It can be reached at:

gopher://parcbench.parcplace.com/11/ParcBenchII

Quasar Knowledge Systems (QKS), makers of SmalltalkAgents, have their own page at:

http://www.qks.com

For more general OO information, there is a searchable data base that includes links to pages for other OO languages, research groups, and lots of other interesting stuff:

http://cui_www.unige.ch/OSG/OOinfo/index.html

PATTERNS

Design patterns are one of the current hot topics in software development. In addition to publications and conferences, there has been a lot of electronic activity on this topic.

One resource is a mailing list on the subject of software patterns. To subscribe to the list, e-mail patterns-request@cs.uiuc.edu with a message containing the single word *subscribe* in the body.

There is also an archive of pattern-related material in the directory /pub/patterns on the st.cs.uiuc.edu ftp site. It includes:

- an archive of messages from the patterns mailing list
- a bibliography of patterns-related material
- source code for the C++ examples from DESIGN PATTERNS[1]
- papers from a variety of conferences, including position papers for workshops, submitted papers, and so forth. Many of them are in PostScript form.

Finally, there is also a WWW site for patterns information. It can be accessed at:

http://st-www.cs.uiuc.edu/users/patterns/patterns.html

It includes some additional information that did not appear to be available on the ftp site, including references to some example patterns.

FONT MANAGER

Support for fonts is not one of the strong points of VisualWorks. While it is certainly not easy to handle fonts both portably and well, many users find the choice of only five different fonts restrictive. Sure, it is possible to add new font choices. All you have to do is create a new CharacterAttributes object with the characteristics

continued on page 30