The Smalltalk Report

The International Newsletter for Smalltalk Programmers

January 1994 Volume 3 Number 4

CROSS-PURPOSE EXCEPTION HANDLING

(PART 1)

by Ken Auer & Barry Oglesby

Contents:

There has been much discussion about exception handling for object-oriented languages in recent years. Unfortunately, the discussion has focused primarily on exception handling for a single thread of control. Such exception handling can often cause compromises in an object-oriented system architecture that has legitimate uses for both multiple processes and exception handling.

Often, multiple concurrently executing processes are necessary to provide desired functionality. These processes usually operate independently and are unaware of each other. Only when a problem occurs must they cooperate to clean it up. Often, a problem occurs in one process, but it is another which must respond to it. For example, in an equipment-supervisor application, a material transfer process that has a problem may cause an exception that should be handled by any process expecting that material to show up at a certain place.

Most commercially available exception handling frameworks[1-3] and language features[4] provide exception handling for only a single thread of control. While extremely useful, such a limitation can discourage or disallow the exploitation of multiple processes to accomplish tasks for which they would otherwise provide excellent solutions. If multiple processes are used, exceptions are either not handled, or they are handled by adding excessive baggage such as instance variables and extra code to provide explicit communication between processes.

Recently, after running head-on into these problems, we decided to extend Smalltalk's process and exception handling mechanisms to make the creation and coordination of processes much simpler, effective, and reliable. Through iterative design and development, we have implemented some fairly simple yet powerful framework extensions. We have found that these extensions sufficiently handle cross-process exception handling in a generic, non-intrusive manner. And, although this is work in progress, our extensions have been successfully applied to a real-world project.

These extensions are not meant to be used as default exception handling techniques. They were made exclusively for use in cross-process exception handling from within the context of the problems presented below. They may be useful in other contexts, but they certainly should not be used in contexts where single thread of control exception handling is desired. Great care has been taken to preserve the standard approach to single thread exception handling. These extensions have no effect on existing software.

This article, divided into two parts, describes how we have extended Smalltalk's process and exception handling frameworks to allow a powerful, flexible, and elegant solution to cross-process exception handling. Part 1 will first provide some background by describing the general frameworks provided by the vendor. We will then present one problem for which no adequate solution exists. Finally, we will discuss our solution to this problem, along with simple examples of its use. Part II of this article will first describe an additional problem with no adequate solution. We will present our solution, including some simple examples,

continued on page 4...

John Pugh

Paul White

EDITORS' CORNER

We are writing this as we head into the Christmas holidays, which started us thinking about a"Smalltalk Christmas Wish List." We originally thought about going to the mall to sit on Santa's knee with this one, but instead decided to write it here (since everybody knows Santa reads THE SMALLTALK REPORT, anyway!).

We were careful to keep our list short. First, it's time to improve the development environment itself. Both Digitalk and ParcPlace haven't made significant changes to the way people browse in years. While it was the best tool available at one time, numerous changes could be made to improve productivity immensely and at a reasonable cost. Second on our list is to make it easier to break into the Smalltalk community. Though the language itself is simple to explain, Smalltalk is a difficult thing to learn and many changes could be made to allow novices to understand how to use it. Third, could we have truly private methods? Fourth, could we rethink the debugging environment? We must make it possible to debug complex interactions with tools other than the simple step/send and hop/skip currently used.

The topic of exception handling has received, proportionally, a great deal of space within this publication. We believe this to be appropriate for a number of reasons. One, it is an interesting topic. Two, it is a difficult topic, and any advice that can be shared by "experts" will help us all. Third, it illustrates the power of Smalltalk, and even if the ideas presented don't directly assist you, we're sure you will be able to learn from them. To this end, Ken Auer and Barry Oglesby of Knowledge Systems begin a discussion this month on introducing exception handling mechanisms for dealing with "multiple concurrently executing processes." Their proposal takes exception handling beyond the level provided currently by Smalltalk vendors.

Ever have a desire to return to "Smalltalk School?" The description provided by Rob Vens of last summer's week-long immersion session sounds very enticing. Most of us are so involved in our current projects that we don't get to take time out to explore Smalltalk in a manner described by him. It seems like an idea that would make sense for most of us, and one our employers would see the benefits of. Plus, it would be a lot of fun—which remains Smalltalk's biggest drawing card!

Kent Beck this month takes aim at two issues he has discussed in the past, namely the shortcomings of automatically using accessor methods for accessing instance variables and of case statements in the Smalltalk language. By presenting examples of each, he makes a strong case against both. Also in this issue, Juanita Ewing is joined by Steve Messick in a discussion of how to achieve better platform independence in our aplications. They discuss ways in which we can factor the pieces of our applications that are dependent on the environment and make them more portable.

Happy New Year!

John Pugh

THE SMALLTALK REPORT (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by SIGS Publications Inc., 588 Broadway, New York, NY 10012 212.274.0640. © Copyright 1994 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Mailed First Class. Subscription rates 1 year (9 issues): domestic, $79; Foreign and Canada, $94; Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: THE SMALLTALK REPORT, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. For service on current subscriptions call 800.783.4903.
To submit articles, please send electronic files on disk to the Editors at 509–885 Meadowlands Drive, Ottawa, Ontario K2C 3N2, Canada, or via Internet to pugh@scs.carleton.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).
Printed in the United States.

The Smalltalk Report

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

SIGS PUBLICATIONS
Advisory Board
Tom Atwood, Object Design
Grady Booch, Rational
George Bosworth, Digitalk
Brad Cox, Information Age Consulting
Adele Goldberg, ParcPlace Systems
Tom Love, IBM
Bertrand Meyer, ISE
Meilir Page-Jones, Wayland Systems
Sesha Pratap, CenterLine Software
Cliff Reeves, IBM
Bjarne Stroustrup, AT&T Bell Labs
Dave Thomas, Object Technology International

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

Columnists
Kent Beck, First Class Software
Juanita Ewing, Digitalk
Greg Hendley, Knowledge Systems Corp.
Ed Klimas, Linea Engineering Inc.
Alan Knight, The Object People
Eric Smith, Knowledge Systems Corp.
Rebecca Wirfs-Brock, Digitalk

SIGS Publications Group, Inc.
Richard P. Friedman
Founder & Group Publisher

Art/Production
Kristina Joukhadar, Managing Editor
Susan Culligan, Pilgrim Road, Ltd., Creative Direction
Gwen Sanchirico, Production Coordinator
Andrea Cammarata, Production Systems Coordinator

Circulation
Bruce Shriver, Circulation Manager
K.S. Hawkins, Fulfillment Manager

Marketing/Advertising
Thomas Tyre, Advertising Mgr—East Coast/Canada
Gabrielle James, Advertising Mgr—West Coast/Europe
Helen Newling, Recruitment Sales Manager
Wendy Plumb, Advertising Assistant
Sarah Hamilton, Promotions Manager—Publications
Caren Polner, Promotions Graphic Artist

Administration
William Ryan, COO
Margherita R. Monck, General Manager
David Chatterpaul, Accounting Manager
James Amenuvor, Bookkeeper
Margot Patrick, Assistant to the Publisher

SIGS PUBLICATIONS

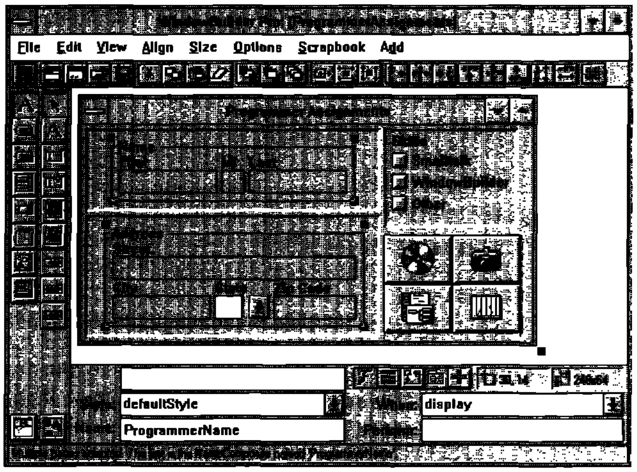
Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, THE SMALLTALK REPORT, and THE X JOURNAL.

SHARE
OBJECT SYSTEMS
INC.

WINDOWBUILDER PRO!

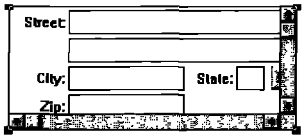
The New Power in Smalltalk/V Interface Development

Smalltalk/V developers have come to rely on WindowBuilder as an essential tool for developing sophisticated user interfaces. Tedious hand coding of interfaces is replaced by interactive visual composition. Since its initial release, WindowBuilder has become the industry standard GUI development tool for the Smalltalk/V environment. Now Objectshare brings you a whole new level of capability with WindowBuilder Pro! New functionality and power abound in this next generation of WindowBuilder.

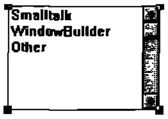


WindowBuilder Pro/V is available on Windows for $295 and OS/2 for $495. Our standard WindowBuilder/V is still available on Windows for $149.95 and OS/2 for $295. We offer full value trade-in for our WindowBuilder customers wanting to move up to Pro. These products are also available in ENVY®/*Developer* and Team/V™ compatible formats. As with all of our products, WindowBuilder Pro comes with a 30 day money back guarantee, full source code and no Run-Time fees.

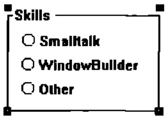
Some of the exciting new features...



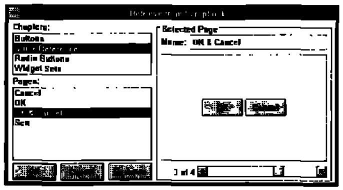
- CompositePanes: Create custom controls as composites of other controls, treated as a single object, allowing the developer higher leverage of reusable widgets. CompositePanes can be used repeatedly and because they are Class based, they can be easily subclassed; changes in a CompositePane are reflected anywhere they are used.



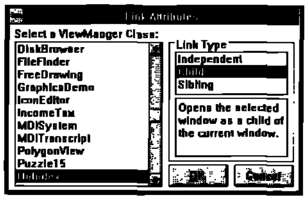
- Morphing: Allows the developer to quickly change from one type of control to another, allowing for powerful "what-if" style visual development. The flexibility allowed by morphing will greatly enhance productivity.



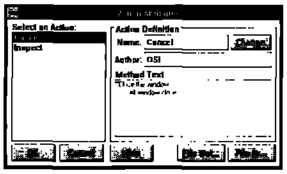
- ScrapBook: Another new feature to leverage visual component reuse, ScrapBooks provide a mechanism for developers to quickly store and retrieve predefined sets of components. The ScrapBook is a catalog of one's favorite interface components, organized into chapters and pages.



- Rapid Prototyping capabilities: With the new linking capabilities, a developer can rapidly prototype a functional interface without writing a single line of code. LinkButtons and LinkMenus provide a powerful mechanism for linking windows together and specifying flow of control. ActionButtons and ActionMenues provide a mechanism for developers to attach, create, and reuse actions without having to write code. These features greatly enhance productivity during prototyping.



Opens the selected window as a child of the current window.



- ToolBar: Developers can Create sophisticated toolbars

just like the ones in the WindowBuilder Pro tool itself.

- Other new features include: enhanced duplication and cut/paste functions, size and position indicators, enhanced framing specification, Parent-Child window relationship specification, enhanced EntryField with character and field level validation, and much more...

- Add-in Manager: Allows developers to easily integrate extensions into WindowBuilder Pro's open architecture.

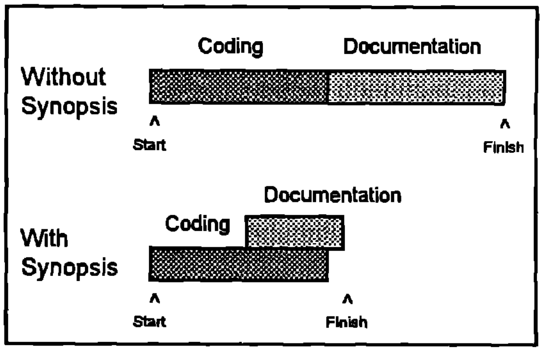
Catch the excitement, go Pro!
Call Objectshare for more information.

(408) 727-3742

Objectshare Systems, Inc
Fax: (408) 727-6324
CompuServe 76436,1063

5 Town & Country Village
Suite 735
San Jose, CA 95128-2026

WindowBuilder and WindowBuilder Pro are trademarks of Objectshare Systems, Inc. All other brand and product names are registered trademarks of their respective companies



...continued from page 1

will provide an extensive example that incorporates both types of extensions.

BACKGROUND

The frameworks described here are based on Objectworks\Smalltalk and the concepts or details described may or may not be applicable to other versions of Smalltalk. Several implementation details are purposely left out since they have no impact on the concepts that are the focus of this article.

Processes

A Smalltalk process is a non-preemptive, lightweight process. A non-preemptive process will not be interrupted:

- By another, same-priority process against its will
- Mid-instruction in any case (an instruction being defined here as anything that causes a new context to be added to the stack)

Any block of code can be the basis of a process. Once a new process is created, the processor must be asked to schedule the process. A process can explicitly be asked to suspend, resume, or terminate. A process can also be implicitly suspended by waiting on a semaphore. A non-terminated process that has not explicitly been suspended is either in a waiting or executable state.

The processor is constantly responsible for deciding which instruction to execute next. It does this by giving control to one of its scheduled, executable processes. The processor will choose the process with the highest priority first. If there are multiple executable processes at the same priority level, it chooses them on a round-robin basis.

A process will continue to execute until interrupted in one of two ways:

- If a semaphore is signaled on which a higher priority process is waiting, that process will be given control before the active process's next instruction is executed. This semaphore can be signaled by the active process or by the Smalltalk virtual machine (if it is a timing- or delay-related semaphore).
- A process can also voluntarily give up control. This can be accomplished by explicitly waiting on a semaphore, by asking the processor to yield to another process at the same priority, or by explicitly suspending itself.

Processes are typically created by sending the newProcess or fork messages to a BlockClosure. Either message returns the newly created process. However, the active process that created the new process has no handle on it. Also, the new process has no handle on its parent process. Unless it is stored explicitly once created, the new process cannot be controlled by its parent process other than via manipulation of commonly accessible objects. The parent and child processes are separated at birth, and all records of the family tree are destroyed.

Since each process is a single independent thread of control, the life, death, or injury of one has no implicit effect on another. A process will execute its instructions as they are defined at cre-

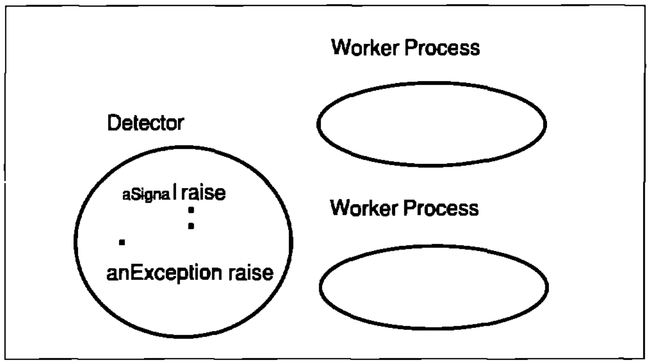


Figure 1. Parallel processes (current).

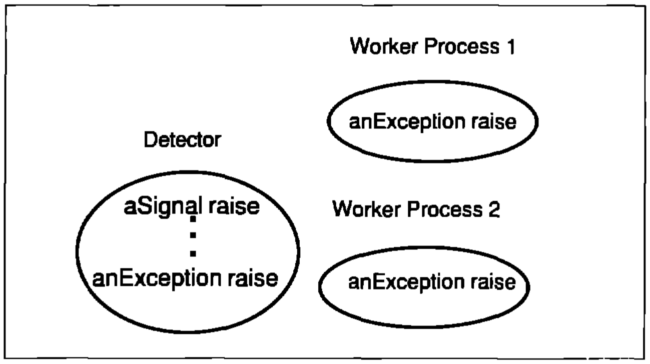


Figure 2. Parallel processes (desired).

ation time. The processor has control over when a process is allowed to continue, but does not affect the order of its instruction execution. However, new instructions can be inserted before a process's next instruction explicitly via manipulation of its context. The most common way for this to occur (outside of debugging) is to send the interruptWith: message to a process. This message takes as an argument a zero argument block. This block will be executed before the process's next intended instruction.

Exception Handling

Once the execution of a process has begun, various types of problems can occur. The method that detects a problem may not know what to do about it. The proper way to handle the specific problem may vary based upon some higher-level context. The generic term for this class of problem is exception. Smalltalk provides a fairly flexible exception handling framework.

This framework is based on signals that, when raised, cause an exception to occur. The context stack of the process in which the signal is raised is searched until one that can handle the exception is found. The handler then acts on the exception. If no handler is found in the context stack, several procedures are tried, with the final one being to run the emergency handler.

Signals have a generalization/specialization hierarchy. Exception handlers will handle exceptions raised by the named signal, or any more specific signal in its lineage. For example, an exception handler for object errorSignal will handle exceptions created by raising object errorSignal or arithmeticvalue divisionByZeroSignal. An exception handler for ArithmeticValue divisionByZeroSignal will handle integer divisionByZeroSignal, but not object errorSignal. A partial hierarchy of predefined signals is as follows:

```
Object errorSignal
        Object notFoundSignal
                Object subscriptOutOfBoundsSignal
                Object nonIntegerIndexSignal
                Dictionary keyNotFoundSignal
        Object messageNotUnderstoodSignal
        ArithmeticValue errorSignal
                ArithmeticValue divisionByZeroSignal
        Stream positionOutOfBoundsSignal
```

An exception handler is simply a one-argument block that takes an instance of exception as its argument. A handler is created by sending the handle:do: message to a signal as follows:

```
aSignal
        handle: [:exception | "handling code"]
        do: ["code for which the above handler applies"]
```

If aSignal or a more specific version of aSignal is raised during the execution of the do: block, the execution of the do: block is interrupted and an exception that is sent as an argument to the handle: block is created.

A handler block can redirect the flow of control in one of four ways:

- Refuse to handle (reject) the exception
- Exit from the handler block and the method in which it is located with some value
- Proceed from the point at which the error occurred (only for certain signals raised in a certain way)
- Restart the do: block.

This article will only expand on the first option, reject. When an exception is rejected, the search for another handler will continue up the context stack. Note that the handler block can do anything before it rejects the exception. For example, it can actually provide some of its own handling prior to passing it on to some other object.

Exceptions contain certain information that may be useful to the handler. For example, the exception can be asked which signal was raised in order to create the exception. It is also the exception that directs the search up the context stack for an appropriate handler. Therefore, the exception must know at which context the search should start. By default, it is the context in which the signal was raised.

Since processes are independent once created, exceptions search only a single context stack for a handler. This causes problems in several realistic scenarios when multiple, concurrently executing processes are operating within a single system. When mutliple, concurrently executing processes exist in one system, they fall into two categories:

- Processes that are parallel to each other or siblings. For example, one process may be looking for problems detected by sensors in a robot, another may be operating the robot.
- One process is a subordinate, or child, of another. For example, a process to operate a robot is created by, and reports to, a factory supervisor process.

These categories often are be combined to create a working system. For instance, the robot-operating process is parallel to the problem-detection process and subordinate to the factory supervisor process. Processes have no handles on their parents or siblings. If problems occur in one, no obvious path of communication exists to another.

In Part 1 of this article, we will discuss the problem of exception handling between parallel, concurrent processes and our solution. In Part 2, we will discuss the problem of exception handling between subordinate processes and our solution.

PARALLEL PROCESSES

It is very possible, especially in the context of machine-control software, that the process that detects a problem is not the process that needs to respond to the problem. For example, a process (known as a *detector*) may exist whose sole function is to monitor an actual physical device. Several additional processes (known as *workers*) that depend on this device to work as expected may also exist. Currently, if the detector process discovers a problem and raises a signal, only it can respond to that signal. An exception will be raised only within the context of the detector process. The worker processes are unaware that a problem has occurred. This scenario is illustrated in Figure 1.

What is necessary is to also have the worker processes respond to that raised signal. If the detector process detects a problem in its device and raises a signal, an exception should be raised not only within itself, but within any worker processes that depend on that device. Each process can then respond appropriately. This scenario is illustrated in Figure 2.

Without modifications to the current framework, a scenario like this would require the detector processes to know all processes that depend on their devices. In keeping with the principles of encapsulation and proper distribution of behavior, it is not wise to have either the detector processes explicitly know enough about worker processes or the worker processes explicitly know the detector processes. In either case, changing the implementation of the detector or worker processes may affect the other. For example, if an additional worker process were necessary, the detector process would have to be modified to know about it.

PROBLEM SOLUTION

To provide the mechanism for many device-dependent processes to be interested in the raising of one signal from an independent detector process, a new handler method, as well as a new instance variable, was added in signal. The new method is called onException:do:, and it employs this new instance variable (called dependentProcesses) to keep track of all interested processes. The invocation of this method is similar to that of Signal>>handle:do: and is described as follows:

```
aSignal
        onException: [:exception | "handling code"]
        do: ["code for which the above handler applies"]
```

The implementation of onException:do: differs from handle:do: in that onException:do: simply adds the active process as a dependentProcess of the receiver signal, then invokes handle:do: to execute the do: block. Upon returning from this execution, the active process is removed from the signal's dependentProcesses. The implementation of Signal>>onException:do: is described below:

```
onException: exceptionBlock do: doBlock
        "Execute the code in the doBlock. If the receiver
        is raised in any process, it will also be raised in
        the dependent processes."
        |result currentProcess |
        [self addDependentProcess:
        (currentProcess := Processor activeProcess). result :=
              self handle: exceptionBlock do: doBlock]
valueNowOrOnUnwindDo:
                  [self removeDependentProcess: currentProcess]. ^result
```

To make use of the processes captured by Signal>>onException:do: as dependents, a new kind of exception class was created called MultiprocessException. In addition to providing normal exception behavior, this new exception subclass caches a collection of processes in an instance variable called interruptProcesses. These processes are all those on which the raising of the signal depends.

Now, if the specific signal is raised during the execution of the above do: block, the execution of the do: block is interrupted. Depending upon the existence of dependent processes, either an exception or a MultiprocessException is created by the signal. If handle:do: was originally employed as the handler, then a normal exception will be created upon raising the signal. If onException:do: was employed, then a MultiprocessException will be created. In either case, the new exception is then raised.

Raising a MultiprocessException causes each of its interruptProcesses to handle a new exception. This is accomplished simply by setting the interruptProcesses of the new MultiprocessException to the signal's dependent processes at creation time. When the MultiprocessException is raised, it interrupts all of its interruptProcesses with a block raising a new exception (based on the parameters set before the raise). As soon as each process is interrupted with this exception-raising block, it stops normal execution and begins executing its own exception handling block.

To accomplish this process-interrupting behavior, several exception methods had to be overridden in MultiprocessException, most notably raise. Its implementation is described below:

```
raise
        "Raise the receiver by raising a new Exception based upon
        the receiver in each of its interrupt processes."
        self interruptProcesses do:
                [:eachProcess | | interruptBlock |
                "Create the block to be sent to eachProcess."
interruptBlock := Processor activeProcess == eachProcess ifTrue:
            [[self newException raise]]
                    ifFalse:
                                    [[self newException
searchFrom: eachProcess suspendedContext; raise]].
                "Interrupt eachProcess with the interruptBlock."
        [eachProcess interruptWith: interruptBlock] fork]. super raise
```

The explanation of this method is divided into three parts:

- Creation of the interrupt block for each interruptProcess. A block of code must be created with which to interrupt eachProcess. The code in this block must first create a new exception, then it must verify that the exception starts its search from the correct initial context, and, finally, it must raise the exception. The only difficulty was to decide what was the correct initial context. If eachProcess is the active process, then the exception created by newException is automatically initialized with the correct initialContext. If eachProcess is not the active process, then its suspendedContext is the correct initialContext and must be set (via searchFrom:).

- Interruption of each interruptProcess. Once the correct interruptBlock is created, then it is a simple matter to interrupt eachProcess (via interruptWith:) with the block.
- Invocation of super raise. The implementation of raise in exception must be invoked in case the active process is in normal exception handling mode (via handle:do:). If this is the case, then the normal method of finding the handler and raising the exception must be preserved.

A simple example illustrating this enhancement follows:

```
[Object errorSignal onException:
                [:ex | Transcript cr; show: 'Handling exception']
        do:
            [Transcript cr; show: 'Delay started'.
            (Delay forSeconds: 5) wait.
        Transcript cr; show: 'Delay ended']] fork
```

Execution of this block of code will result in a process being created. This process will show 'Delay started' in the transcript window, followed by 'Delay ended' after five seconds. If object errorSignal is raised (via 'Object errorSignal raise') in any running process (including this one) during the five second delay, then normal processing would be interrupted in this process and its exception handling block would be invoked, thus producing 'Delay started' in the transcript window, followed by 'Handling exception' after a delay of under five seconds.

At this point, we have discussed some general background on the process and exception handling frameworks as provided by the vendor, as well as a problem that arises when multiple concurrently executing processes are necessary and our solution. In part 2 of this article, we will describe an additional problem and our solution, including some simple examples. Also, we will provide an extensive example that incorporates both types of extensions. ■

References

1. ParcPlace Systems, Inc. Objectworks\Smalltalk" User's Guide, Chapter 8, 1992.
2. Hinkle, B., and R.E. Johnson, Taking exception to Smalltalk, part 1. THE SMALLTALK REPORT, 2(3),1992.
3. Hinkle, B., and R.E. Johnson, Taking exception to Smalltalk, part 2. THE SMALLTALK REPORT, 2(4),1993.
4. Pyle, I.C. THE ADA PROGRAMMING LANGUAGE. Prentice Hall, Englewood Cliffs, NJ, 1981.

Ken Auer is the Director of Development Services at Knowledge Systems Corporation, Cary, NC. He can be reached at 919.481.4000 or kauer@ksccary.com. Barry Oglesby is a member of the technical staff of Knowledge Systems Corporation. He can be reached at 919.481.4000 or boglesby@ksccary.com.

Kent Beck

Death to case statements, part 2

Hold on a sec. . . .

Before I finish bashing case statements, I'd like to return to the scene of an earlier crime, my perfidious assault on that bastion of Smalltalk orthodoxy, the ubiquitous accessor method. (Whew! That's a ten-buck sentence if I ever seen one.) I argued that the violation of encapsulation provided by accessor methods more than offset any benefit of inheritance reuse. I talked to several readers at OOPSLA who were offended by that column, although no one wrote me directly. Well folks, none of those beer-soaked conversations convinced me differently in Washington, and a couple of recent events leave me even more sure that insisting all variable access go through a message is a bad idea.

Here's the basic problem: Beginners don't get the message that accessor methods should be private by default. They hear the rule, *access variables only through a message,* and they think, "Great, here's one thing I can do to make sure I'm not messing up." They're using their new object and they say, "Hey, if I just had that variable over there I could solve my problem." Next thing you know, representation decisions have leaked all over, none of the objects have grown the behavior they need, and progress slows to a crawl.

I was at a client recently where they had misused accessor methods all over the place. The biggest problem was in changing collections behind the owning object's back. They wrote code like this:

```
Schedule>>initialize
    tasks := OrderedCollection new

Schedule>>tasks
    ^tasks
Then in user interface code they would write:
ScheduleView>>addTaskButton
    ...
    model tasks add: newTask
```

The problem with this code is that it assumes that tasks returns an object that responds to add:. If they changed the representation in Schedule to store tasks as a Dictionary instead of an OrderedCollection, the ScheduleView code breaks. The implementation of Schedule has leaked out, and that's exactly the kind of problem objects are supposed to help us avoid.

Later on in this same assignment, the horror that accessing methods are there to avoid happened to me—I changed an instance variable so that it was lazily initialized. I had to change all those methods that directly accessed the variable so they sent a message instead. It took me all of three minutes and I was done.

The point here is not that accessor methods are useless. There are definitely cases where judicious use of accessors can improve code. However, teaching beginners always to use accessors before they are able to understand the need to keep some methods private avoids reuse problems far down the road at the cost of encouraging them to violate encapsulation.

Enough about accessors. If you don't agree, let me know. I'd love to see a reasoned discussion of this issue, since accessors are accepted as an article of faith by so many people, and I see lots of bad code being written while adhering to the letter of the "accessor law."

The real purpose of this article is to complete my thoughts about case statements from last issue. QKS' SmalltalkAgents has introduced a case construct. I'm making the argument that case statements in an object language are superfluous, and they prevents discovering important new objects. Rather than just complaining about case statements, though, I'll show you how to turn a situation that uses case logic into a richer use of objects. (This is typical of patterns: They don't just describe a good or bad situation, they tell you how to get from bad to good.)

PATTERN: OBJECTS FROM STATES

Problem

Parallel case statements are a maintenance nightmare. Changing one instance of the case without changing the others can lead to subtle bugs. How can you use objects to eliminate case statements?

Constraints

One of the goals of any programming activity is not to introduce any more complexity than necessary. Creating methods and classes that don't have any payoff is a common programming mistake. The solution to the case statement problem should create only new classes and methods that pay for their existence with reduced maintenance, improved readability, and increased flexibility.

The solution must eliminate the case logic that causes maintenance problems. Why are case statements a problem? Essentially, multiple case statements with the same cases introduce a multiple update problem. You can't correctly change one statement without changing all the others, and this relationship is entirely implicit. While you might be able to keep track of where all the cases are today, a year from now you (or worse, someone else) will have to know to look for them all, and know where to look.

Finally, the solution should set the stage for further growth of our objects. Some of the most valuable objects you can find are the ones that are not obvious from the user's view of the world. These are the objects that structure not the world, but our computational model of the world. (Other patterns like this are Objects from Collections, Objects from Instance Variables, and Objects from Methods). Taking advantage of the appearance of case logic should make programs more explicit and more flexible.

Solution

Make an object for each state. Make a variable in the original object to hold the current state. Move the logic in each case into the corresponding state object. Delegate to the current state instead of executing in the original object. Make the state changing methods assign a different state object to the state variable.

Example: Consider a visual object that can be in one of three states: enabled, disabled, or invisible. The state is represented by storing a Symbol in the variable state. A couple of the methods might be:

```
Visual>>display
    state = #enabled ifTrue: [...display enabled... .
    state = #disabled ifTrue: [...display disabled... .
    state = #invisible ifTrue: [...do nothing...

Visual>>extent
    state = #enabled | (state = #disabled) ifTrue: [^40@40 .
    state = #invisible ifTrue: [^0@0 .

Visual>>enable
    state := #enabled

Visual>>disable
    state := #disabled

Visual>>disappear
    state := #invisible
```

Applying Objects from States, we first make an object for each state:

```
EnabledVisual, DisabledVisual, InvisibleVisual, subclasses of Object.
```

We can use the variable state to hold an instance of one of these. Moving the logic into the state objects yields:

```
EnabledVisual>>display
    ...display enabled...

EnabledVisual>>extent
    ^40@40

DisabledVisual>>display
    ...display disabled...

DisabledVisual>>extent
    ^40@40

InvisibleVisual>>display
    "Do nothing"

InvisibleVisual>>extent
    ^0@0
```

Then Visual has to change to invoke the state:

```
Visual>>display
    state display

Visual>>extent
    ^state extent
```

Finally, the state-changing methods have to change.

```
Visual>>enable
    state := EnabledVisual new

Visual>>disable
    state := DisabledVisual new

Visual>>disappear
    state := InvisibleVisual new
```

Other patterns

After you've applied Objects from States, you may have to use Delegate or Call Back to fully move each state's logic into the state object. You may be able to use Factor a Superclass to simplify the implementation of the states and prevent multiple update problems.

CONCLUSION

This and the previous column have shown how to eliminate most uses of case-type logic. The remaining examples of case statements don't appear frequently enough to justify a new language construct. The power of Smalltalk lies primarily in its simplicity, out of which richness can grow without undue complexity. Every new feature must pay for itself by solving a problem affecting a large part of the community. On this grounds, case statements just don't cut it.

What's next? In this pattern, I referred to several others that created new objects. I think I'll spend at least a couple more months exploring this theme. See you in the next issue with the second installment of "Daddy, where do objects come from?" ■

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

Juanita Ewing & Steve Messick

Techniques for platform independence

This article discusses techniques for writing platform-independent applications and class libraries. The techniques we discuss are useful for modeling environmental changes that affect your application. For example: operating system facilities that vary from platform to platform, windowing libraries for Windows and OS/2 Presentation Manager, a database connection that varies depending on the network configuration, archiving libraries that use either PVCS or Oracle for storage, a color model that depends on the current output device and even Smalltalk platforms such as Smalltalk/V and Objectworks\Smalltalk. These techniques could also be useful as part of a system that models user-experience level.

One way or another, all the techniques in this article are based on polymorphism. They rely on client objects sending messages to platform-dependent objects. The client always sends the same messages, which is where polymorphism comes into play: Every platform-dependent object must understand those messages. Thus, during both the design and implementation phases, it is important to think about the set of public messages for objects and the requirement for polymorphism.

This article will refer to classes providing platform services as *library classes,* and the client classes that use these classes as *application classes.*

INTERCHANGEABLE CLASSES

Two library classes with the same set of public messages can be used to interface to two different platforms. Because they have the same set of public messages, they are interchangeable. Often, it is convenient to arrange these classes as subclasses of a common superclass because inheritance supports common behavior. The superclass contains common messages, and documents requirements for creating additional subclasses. Often, the superclass is abstract, meaning there are no instances of it. For a discussion on creating abstract classes that are based on similar concrete classes, (see "Abstract classes," THE SMALLTALK REPORT, Vol. 3, No. 2).

The currently appropriate platform-dependent class is known as the *current class.* This is usually a concrete class, a class that can have instances. Because the current class changes, it must not be directly referenced by clients. Let's look at several alternatives for indirectly referencing the current class.

How do application classes reference the current class? Because the class may change, depending on the environment, application classes cannotreference the current class by name (Figure 1). At compile-time the current class is not known. Instead, application classes must reference an indirection to the current class, so that the current class can be replaced.

Some class libraries use a global variable to refer to the current class (Figure 2). Application code indirectly references the current class with expressions like CurrentDatabaseInterface cancelConnection, which cancels the connection to the current database. In most cases, a variable that is global in scope is not necessary.

An alternate solution is to ask the abstract class for the current subclass (Figure 3). A class instance variable or class variable can be used to hold the actual reference. This solution produces expressions like DatabaseInterface current cancelConnection. A message to the abstract class to retrieve the current class is better than a global variable because:

- The functionality is clearly related to the class.
- The abstract class is already in the global namescope.
- The abstract class is usually named so that its purpose is obvious to clients.
- The use of a class message keeps all related functionality in a nice neat bundle that is easier to share and maintain.

Using interchangeable classes, we discuss three approaches to portability, each appropriate for a different set of assumptions. The approach you select will be based on the specifics of your situation.

MICRO-LAYERING

In the micro-layer approach to portability, we recognize that the developer must make a portable version of a library class, without changing its public interface. One of our goals is to make the library class itself as portable as possible. The requirement is to extend an existing class to accommodate multiple platforms. We cannot make a new class to replace the existing class without affecting clients of the existing class.

Assumptions for the micro-layer approach are:

- The library class must be made portable.
- Clients must not be affected.

We can, however, introduce a new, non-portable class that isolates the platform dependencies of our library class. Then we rewrite the library class methods to use the non-portable class to perform host-dependent operations. By moving platform-dependent code into a non-portable class we've done two things:

- Eliminated the need to change any public protocol understood by the library class
- Provided ourselves an easy way to port the non-portable code

The new class is completely under our control and can be ported by using interchangeable classes, as described above.

As an example, let's assume we're creating a portable Point class that can work with the host's coordinate system, and that it must work on a variety of platforms including Macintosh and OS/2. We immediately see that coordinate system is platform dependent: the *x* coordinate increases as we move from left to right on both platforms, but the *y* coordinate increases downward on Macintosh and upward on OS/2. To test whether one point is above and to the left of another one, we can write for Macintosh:

```
Point methods
isLeftAndAbove: aPoint
    "Return true if the receiver is left and above <aPoint>"
    ^self x < aPoint x and: [self y <aPoint y]
```

And for OS/2 it becomes:

```
Point methods
isLeftAndAbove: aPoint
"Return true if the receiver is left and above <aPoint>."
^self x < aPoint x and: [self y > aPoint y]
```

The different interpretation of *y* coordinates occurs throughout the class and also affects other classes such as Number and Rectangle. These two methods satisfy our second assumption (clients must not be affected), but not the first, that separate implementations are required for different platforms. Note that most of the methods are identical; only the test of *y* coordinates differs. Isolating the platform-dependent behavior in a new class will make Point (and Number and Rectangle) portable.

Let's introduce a class for coordinate system dependencies, called CoordinateSystem. Using interchangeable classes, we can design a Coordinate System micro layer for Macintosh and OS/2. We'll call the classes MacCoordinateSystem and OS2CoordinateSystem. Both are subclasses of CoordinateSystem (Figure 4).

Since the interpretation of *x* coordinates is the same, we will define the *x* axis protocol in CoordinateSystem. MacCoordinateSystem will interpret increasing *y* coordinates downward and OS2CoordinateSystem upward. Rewriting the Point method, we have:

```
Point methods
isLeftAndAbove: aPoint
    "Return true if the receiver is left and above <aPoint>."
    ^(self coordinateSystem is: self x leftOf: aPoint x)
        and: [self coordinateSystem is: self y above: aPoint y]
```

This method is portable, assuming the method coordinateSystem answers an instance of the correct subclass of CoordinateSystem. Additionally, if Point needed to be ported to a platform that interpreted *x* coordinates increasing from left to right, it is still portable providing a new subclass of CoordinateSystem is created.

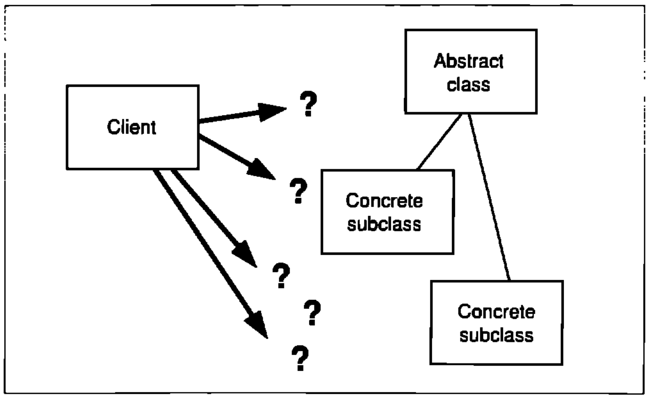


Figure 1. How are library classes referenced?

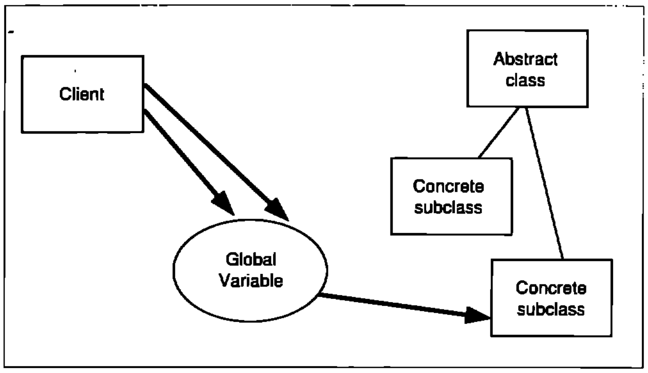


Figure 2. Application code sends messages to a global variable.

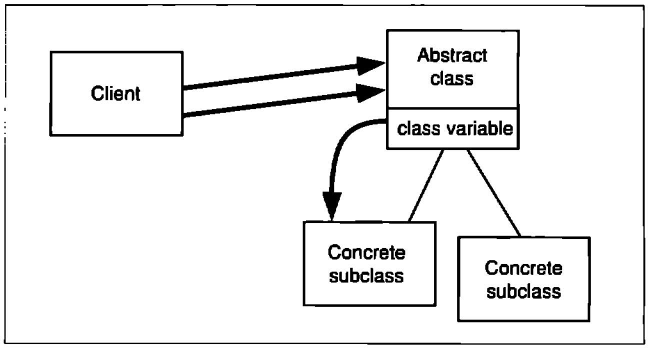


Figure 3. Application code sends messages to the abstract class.

Now let's look at CoordinateSystem and its subclasses. We need to define is:leftOf: and is:above:.

```
CoordinateSystem methods
is: firstX leftOf: secondX
    "Return true if <firstX> is to the left of <secondX>."
    ^firstX > secondX

is: firstY above: secondY
    "Return true if <firstY> is above <secondY>>."
    self implementedBySubclass

MacCoordinateSystem methods
is: firstY above: secondY
    "Return true if <firstY> is above <secondY>."
    ^firstY <secondY

OS2CoordinateSystem methods
```

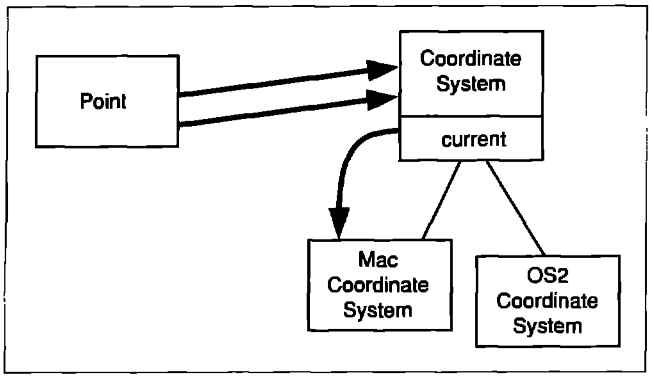


Figure 4. The major objects required to represent one standard window and one floating window on a Macintosh.

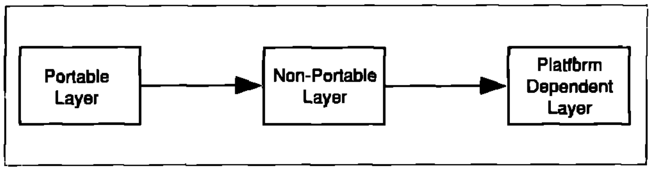


Figure 5. A portable layer that depends upon a non-portable layer for communication with the host platform.

```
is: firstY above: secondY
    "Return true if <firstY> is above <SecondY>."
    ^firstY > secondY
```

MACRO-LAYERING

The micro-layer approach illustrated the use of interchangeable classes in a microcosm. The next variation applies the same principle on a bigger scale, as the architectural basis of entire systems.

In the macro-layer approach, we must make an entire subsystem portable. The assumptions, as in the micro-layer approach, are:

- The library must be made portable.
- Clients must not be affected.

This is a good approach to use when developing a portable user-interface framework. Digitalk's Smalltalk/V version 2.0 for Macintosh uses it. A different form of it also shows up in ParcPlace's Objectworks\Smalltalk.

The general idea is simple: Develop a portable layer that depends upon a non-portable layer for communication with the host platform (Figure 5).

The platform-dependent layer is the service provided by the platform that we need access to in Smalltalk. It may be a user interface like Windows 3.1, a communications toolbox such as Apple's AOCE or even a third-party product. The only requirement is that it have a well-defined API that can be used by Smalltalk.

The portable layer implements the classes used by client applications. This is the layer most commonly used by Smalltalk programmers. A good example is a user interface framework. The portable classes that implement the framework can be used by application-specific classes to define windows. The application code is protected from platform dependencies, as long as it uses only the portable layer, and is therefore portable.

Interfacing between the portable layer and the platform-dependent layer is the responsibility of the non-portable layer. This layer must do whatever is necessary to transform portable requests, such as creating new windows, into the platform specific requests that actually create the window. This often requires transformation of data from a portable representation into the representation used by the platform, and calling the correct subroutines defined by the platform's API.

To make the system run on another platform with that platform's implementation of the service, the middle non-portable layer is ported to the new platform. If the portable layer was implemented without relying on any non-portable assumptions then it will work as is. Practically speaking, there may be some code in the portable layer that will not work on the new platform without modification. To ensure portability of client applications the public protocol defined by the portable layer may not change. But applications will work just fine if the public protocol preserves its semantics across platforms, no matter how it is implemented.

This brings us to the issue of specifying the portable layer. This is actually the most difficult part of creating a portable library. Since the underlying service we want to use is itself not portable, we cannot simply look at its API and define our portable protocol in terms of it. We have to create a framework that can be implemented on all potential platforms. The syntax and semantics of the framework has to be specified so that client applications can be defined. The specification must also define the protocol that future extensions to the framework may and or not modify. Also, any methods that have a non-portable implementation must be indicated. To learn more about current research issues in object specification, see the OOPSLA papers by Kiczales and Lamping.[1,2]

Let's consider an example. Suppose we are creating the user-interface framework for a family of applications for OS/2 and Macintosh that need to use the host's windowing system. These applications need some "non-standard" windows that always display on top of "standard" windows. These are sometimes called *floating windows* or *palettes*. Looking through the OS/2 manuals, we see that this won't be very difficult. OS/2 provides the capability we need. However, an extremely careful reading of INSIDE MACINTOSH reveals that we may be able to get one window of this sort, but if we need more than one—and we do—we're out of luck. It turns out that we have to reimplement a portion of the Macintosh window manager class to solve this problem. By applying the principle of interchangeable classes to the problem description, we can design and specify a WindowManager class that has implementations for OS/2 and Macintosh. In our portable user-interface library we define the classes StandardWindow and FloatingWindow to implement the two varieties of window we need. These classes use WindowManager to create and destroy windows and to make windows visible or invisible. We'll also have non-portable classes, OS2Window and MacWindow, to implement the platform-specific window functions like setting window title and size. The result is a user-interface framework for building portable applications, and a framework that is itself largely portable. The design includes no inherent performance penalty for either platform.

If, on the other hand, we had tried to design the framework based only on the OS/2 API, we probably would have arrived at a much less portable version of the framework. It is quite likely that our design would not have included either WindowManager or FloatingWindow. After all, why should it? OS/2 takes care of all the bookkeeping required. We would, of course, have StandardWindow and OS2Window because we're using the layering method to isolate platform dependencies. But that alone is not enough to ensure portability. If missing functionality must be implemented for some platform, then the design must allow for that. If the functionality is not part of the design, client applications will be based on a sub-optimal design, and we will be faced with enormous backward-compatibility problems. Rather than designing a system based on the functionality available on a platform, we design the system to meet our requirements.

An interesting variation on the layering theme is found in Objectworks\Smalltalk. The non-portable layer is implemented in the virtual machine. The portable layer is implemented in Smalltalk; it is entirely portable because all platform dependencies are hidden in the virtual machine. Using this approach, Objectworks ensures portability of the applications defined in Objectworks\Smalltalk and also of their image file.

PLATFORM SERVER

The last variation is a pragmatic approach that is often used to extend the set of platforms an existing application can support. In this approach, a platform server class is used to contain all platform specific code that the application relies on. There is one platform server class for each platform, providing a consistent interface to platform functionality.

This can be used when an application relies on two platform libraries that do not have an identical public interface. Our advice is to use this technique only if you do *not* have control of library classes, or as a stopgap measure if you can rewrite library classes. If possible, you should refactor and expand the set of library classes, resulting in many interchangeable classes.

Assumptions for the platform server approach are:

- Many small variations in library classes
- The developer cannot rewrite library classes

For this technique, let's discuss an example involving the platforms SmalltalkAgents for Macintosh, and Objectworks\Smalltalk. Suppose we have an application that must run on both. This application requires streams and a collection that holds its elements in order. We also want the ability to do some rudimentary performance analysis, and therefore need an operation that can be used to time the execution of a block.

The way we access the required functionality is different with each Smalltalk platform. To isolate the bulk of our application from platform dependencies, we compartmentalize the variations for each platform into a platform-server class. The class ObjectworksServer is a mapping to functionality on the Objectworks\Smalltalk platform, and the class SmalltalkAgentsServer is a mapping to functionality on the SmalltalkAgents platform.

Let's examine a sample of methods from the server classes. Methods that identify an appropriate class, such as the method orderedListClass, are useful when two platforms have similar

ObjectworksServer methods

```
ObjectworksServer methods
orderedListClass
    "Return a class that holds its elements in order."

    ^OrderedCollection

readFrom: aStream through: anObject
    "Return a collection of elements read from <aStream>, starting
from the current stream position up to and including <anObject>."

    ^aStream through: anObject

timeToExecute: aBlock
    "Return the number of milliseconds to execute <aBlock>."

    ^Time millisecondsToRun: aBlock
```

SmalltalkAgentsServer methods

```
SmalltalkAgentsServer methods
orderedListClass
    "Return a class that holds its elements in order."

    ^List

readFrom: aStream through: anObject
    "Return a collection of elements read from <aStream>, starting
from the current stream position up to and including <anObject>."

    | throughCollection |
    throughCollection := aStream upTo: anObject.
    throughCollection add: aStream next.
    ^throughCollection

timeToExecute: aBlock
    "Return the number of milliseconds required to evaluate <aBlock>,
rounded to the nearest ms. The computation is at best approximate
because the basic unit provided by Apple is a tick (1/60 s)."

    | timer startTime |
    timer := ClockDevice new.
    startTime := timer ticks.
    aBlock value.
    ^timer ticks - startTime * 100 + 3 // 6
```

classes with different names. It can also be useful to help identify dependencies and collaborations.

Ordinarily, when operating on an object, we send messages directly to the object. When we provide a mapping to that functionality in a platform server class, we must send a message to the server class. The original object becomes an argument. The message readFrom:through: provides an operation on a stream, but it is a message sent to the server class with the stream as an argument.

> “One way or another, all the techniques in this article are based on polymorphism.”

Sometimes the server class will end up implementing functionality that is simply not present on one of the platforms. The message timeToExecute: is a mapping to existing functionality for Objectworks\Smalltalk, but it is new functionality for SmalltalkAgents.

DYNAMIC VS. STATIC

There is another issue, orthogonal to variations on interchangeable classes, that deserves discussion. This is the issue of how applications can be configured.

If an application runs on one platform at a time, developers can use configuration management tools to build applications with the appropriate platform-dependent classes. The result is several versions of an application, one for each platform. We call this situation a *static configuration,* and do not discuss it in detail. There are several commercially available tools for configuration management of Smalltalk applications such as Team/V and ENVY Developer.

If the application must run on multiple platforms, developers can design their application to dynamically support the appropriate one. In this situation, called a *dynamic configuration,* the result is one version of the application that includes all platform-dependent classes.

SETTING THE CURRENT CLASS

There are several different ways of installing the current platform-dependent class. The exact mechanism depends on how often the environment changes. Does the current class potentially change every time it is accessed, or does the default change less frequently?

Some classes are installed when Smalltalk is started. In Smalltalk/V for Macintosh, any object can register for notification when Smalltalk starts with an expression like this:

```
SessionModel current
    when: #startup
    send: #setCurrent
    to: PlatformInterface
```

The setCurrent method includes an expression to set the current class. We use the class ServiceRegistry to identify the current platform. The method setCurrent is implemented by the class PlatformInterface.

```
setCurrent
    "Set the current platform interface class based on thecurrent platform."

    | platformName |
    platformName := ServiceRegistry globalRegistry
                serviceNamed: #PlatformName
                ifNone: [^self installStub].
    platformName = 'Macintosh'
        ifTrue: [^self current: MacPlatformInterface new].
    platformName = 'OS/2'
        ifTrue: [^self current: OS2PlatformInterface new].
    platformName = 'Windows'
        ifTrue: [^self current: WindowsPlatformInterface new]
```

Another strategy is to wait until a current class is requested and then determine the current class if necessary:

```
current
    "Answer the interface used by the current platform."

    Current == nil
        ifTrue: [self setCurrent].
    ^Current
```

Even with this strategy, the old current class must be flushed at some appropriate time, such as image start-up, so that the current class will be installed when the class is accessed. This expression should be executed sometime during application start-up:

```
PlatformInterface flushCurrent
```

CONCLUSION

Developing platform independent applications means more than writing code for different hardware platforms such as Macintosh and IBM PCs. Different software platforms can also be addressed with the same techniques.

Carefully consider all possibilities for extension of your application while choosing design and implementation techniques. If you follow the approaches laid out in the article, it will be much easier to move your application from one platform to another. There are, no doubt, other interesting techniques that can be used to ease the task of porting between platforms. We'd like to hear about them. Send descriptions of your own practices to juanita@digitalk.com. ■

Juanita Ewing and Steve Messick are senior staff members at Digitalk Professional Services, 921 SW Washington, Suite 312, Portland, OR 97205, 503.242.0725.

References

1. G. Kiczales and J. Lamping. Issues in the Design and Specification of Class Libraries, OOPSLA '92, Vancouver. pg.435–451.
2. J. Lamping. Typing the Specialization Interface, OOPSLA '93, Washington, pg. 201.

Jan Steinman & Barbara Yates

Shoot-out at the Mac corral, part 2

About ten years ago, a Smalltalk product called Methods was introduced for IBM PCs and compatibles. This product grew into the Smalltalk/V product family that now largely dominates the IBM-compatible Smalltalk market.

Along the way, a specific need for a Macintosh product arose, and the first Smalltalk/V-Mac, built largely from acquired technology, shipped in November 1988.

As corporate MIS departments discovered Smalltalk, and IBM itself endorsed it, more and more effort went into enhancing and refining Digitalk's IBM-compatible line, while the Mac version went through only two minor "bug-fix" update cycles, culminating in June 1991 with version 1.2, which sold for $199.95.

In 1992, Digitalk acquired Instantiations, a Smalltalk consulting company largely derived from the remains of the disbanded Tektronix Smalltalk product group. This Portland, OR–based group brought extensive knowledge of both 680x0 Smalltalk virtual machine implementation and Smaltalk-80 virtual image internals.

It is not surprising that this flow of history should lead up to August 1993 and the first major upgrade of Smalltalk/V-Mac since its introduction. Version 2.0, while overdue, is impressive: it boasts about three times as many classes and methods as were in 1.2. Although the price has increased by a similar factor to $495, it is probably still the least expensive commercial Smalltalk available today—especially if it is still available at the upgrade price of $195 that was in effect at the time this was written.

Part 1 of this series (THE SMALLTALK REPORT, Vol. 3, No. 3) described a new and original Smalltalk implementation, SmalltalkAgents, from Quasar Knowledge Systems. (A brief update on their progress appears in this article.) Also available for the Mac is Visualworks\Smalltalk-80 from ParcPlaceSystems.

(For convenience, the three dialects will be referred to as STA, ST-80, and ST/V, respectively. Unless otherwise noted, use of "ST/V" refers only to the Mac 2.0 version, not the other products in the Digitalk stable.)

HIGHLIGHTS AND CHANGES

Unfortunately, we must begin the feature list with an omission. There are no release notes and no concise description of the differences between version 1.2 and 2.0. Those already familiar with ST/V will quickly spot major additions, but many small improvements—and incompatibilities with version 1.2—await discovery. This makes it difficult to plan a port; it would be nice to know concisely what had changed, so that one could scope the porting effort.

Among a long list of long-awaited features claimed by Digitalk for version 2.0 are: DAL/DAM SQL, 32-bit color QuickDraw, QuickTime, Balloon Help, and AppleEvents. We'll look at just a few new things in detail.

WINDOW INSPECTOR

A new specialized Inspector for windows is available on the Window menu. This is a useful tool that many people end up building themselves—we'd like to see in all dialects of Smalltalk. It eliminates the need to open many levels of Inspector to figure out what is going on in the application window you are developing (Figure 1).

CLASS HIERARCHY BROWSER

The Class Hierarchy Browser has a different arrangement of panes and buttons, but otherwise it appears to be similar to the one in 1.2. One pane lists the local and inherited instance and class variables of the class. Selecting in this pane filters the methods list to accessors and setters of the variable. Clicking again on a list selection does not deselect it, so it is not obvious how to unfilter the methods list.

Find Class in the Classes pane now works with wildcards, but it is fussy. When searching for window classes, *wind*, *windo*, *window*, *window, and *Window were all rejected with the same message: Please enter a more specific pattern. This was frustrating, and not very useful!

The access to the senders or implementors of a message in a selected method requires more menu actions in 2.0 than in 1.2. We found the extra steps annoying, and we disliked having to have an extra window open to get to the information we wanted. Developers with small screens will really notice this difference, although ST/V is still more screen-efficient than STA.

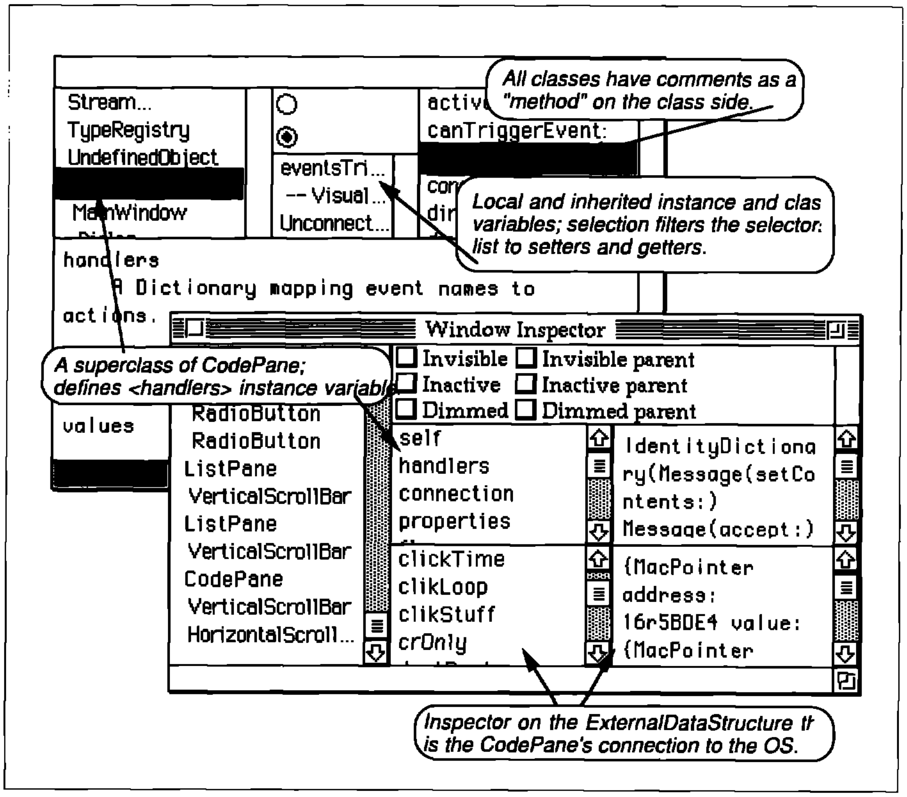
Text automatically word-wraps in code panes. This feature is replaced by horizontal scroll bars for any window in which the user turns auto-wrapping off. ST/V's non-word-wrapped heritage is irritatingly apparent—method comments are formatted wider than the default window width, which makes for difficult reading.

WORKSPACES

Temporary variables are automatically generated for code written and executed in workspaces. (This is obviously part of the new Tektronix heritage—wherever two or more former Tek Smalltalkers gather, they bemoan the lack of workspace variables!) Another improvement is that workspaces saved to a file and later reopened retain their text fonts and emphasis. Unfortunately, along with the improvements come a large increase in the amount of time it takes to open a workspace from a file. This exceeds a 10:1 performance decrease in certain situations we encoundered.

EVENTS—A ROADMAP

The classic Smalltalk ìdependents changed/updateî mechanism of user interaction is showing its age, and Smalltalk developers are clamoring for event-driven operation. ST/V-Mac 2.0 uses a strategy for change initiation and propagation that may be simpler to program than the classic dependency mechanism.


All classes have comments as a "method" on the class side.

Local and inherited instance and clas variables; selection filters the selector list to setters and getters.

A superclass of CodePane; defines <handlers> instance variabl

Inspector on the ExternalDataStructure tr is the CodePane's connection to the OS.

Figure 1.

The advantage of true event handling is that polling of user interface components is hidden, ideally by direct use of operating system services. Rather than polling the state of input devices and locations in Smalltalk code, user actions and other system happenings spontaneously result in the evaluation of Smalltalk expressions.

Apparently, this is not quite the way ST/V operates—it still has a single bottleneck Smalltalk Process that gathers and dispatches operating system events, so the differences tend to be syntactic rather than semantic.

Perhaps the worst way to try to attempt to understand the ST/V event dispatching is the way we initially tried. Merely browsing the code of Application and related classes does not quickly reveal how to use ST/V event dispatching effectively. This is the approach of those who are unused to having in-depth documentation!

Those who are used to ìchanged/updateî event dispatching may want to start with the tutorial in Chapter 9, even if they normally do not do tutorials. The tutorial in Chapter 10 also looks at building your application window and setting up events. Next, an examination of the Application class in the ENCYCLOPEDIA OF CLASSES, combined with a reading of the programming reference chapters (6 and 7), will rapidly add to your understanding of setting up, triggering, and responding to events.

At this point, browsing the code of existing Application subclasses is fruitful. Eventually, the serious Smalltalker will want to memorize Appendix C, or at least make up a ìcheat sheetî from it.

To maintain one paradigm between user-initiated object change and other object changes, ST/V 'events' replace the generic update sent to all dependents of the object receiving changed. This somehow feels less object oriented—previously, an object's dependents decided whether they were interested in that object's happenings at runtime; now the object must register for specific happenings, thus weakening the traditional model/UI division of responsibilities. ST/V makes this difference a bit less onerous for pre-2.0 applications by configuring an event and action when an application receives the addDependent: message, providing some compatibility for many cases.

Are events, as implemented in ST/V, a good thing? They are undoubtedly valuable for capturing user actions, but they can certainly be abused. We can't say that the result is easier to understand, debug, or maintain than the classic "changed/update" mechanism, and it may prove less flexible in complicated situations, but it is probably simpler to program in most cases.

Either way, developers porting to 2.0 will not be able to ignore events, since they are firmly embedded in the application architecture. Perhaps some third-party user interface builder will come to the rescue and provide more support for putting events in place.

BETTER DOCUMENTATION

The package includes three manuals: TUTORIAL, PROGRAMMING REFERENCE, and ENCYCLOPEDIA OF CLASSES,—about 2-1/2

Should I Use Smalltalk/V–Mac 2.0?

platform compatability is immediate concern YES — *ST/V is not the best choice. ST–80 does this extremely well. Consider using ST–80.*

NO

Digitalk is becoming increasingly attentive to portability issues. Expect dialects of ST/V to grow closer, but it will take time.

portability with other Smalltalks required YES — *ST/V base classes are fairly portable with ST–80, but anything non–trivial will require some effort. OS dependent classes may take considerable effort and UI classes may require a re–write, even among ST/V dialects.*

NO

Consider using ST/V.

performance is major concern YES — *ST/V finishes at the bottom on low–level benchmarks. However, Digitalk now has acquired 680x0 Smalltalk VM experience; future releases should improve, and low–level tests may not reflect your use.*

NO

Consider using ST/V.

working with a team NO — *Consider STV. Its single–user facilities are generally good.*

YES

There is no credible groupware available for ST/V–Mac at this time. Although there will undoubtedly be a Team/V port, the present lack of tools means rigorous procedures and conventions will be necessary to use ST/V–Mac in a team environment.

robust environment is immediate factor NO — *Consider STV. It will undoubtedly improve with subsequent versions.*

YES

We and others we communicated with experienced numerous bugs and crashes. Although it is called "2.0," this release is a major re–write, and we feel it is of similar stability to SmalltalkAgents, a brand–new product.

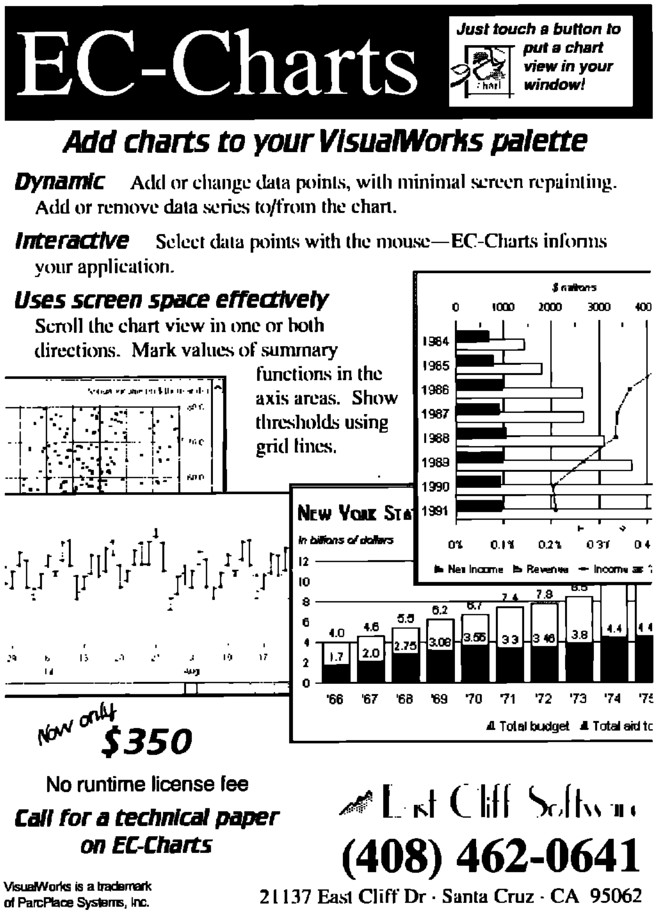
rich toolset required NO — *Consider using STIV. Its toolset is much better than ST/V–Mac 1.2.*

YES

ST/V cannot yet compete with other Smalltalk's tools. It lacks a GUI builder, programming tools lack features found in other Smalltalks, and there are no third–party tools available yet. This will change, but it will take time.

times more material tan found in the 1.2 manual. We found the encycopedia's methods index useful as a reference for standardizing selector names. The reference book is well-written, and its appendix on events is a valuable condensation of events information. Another nice section, especially for beginners, is the glossary, which defines most of the terms used throughout the documentation.

Finally (and perhaps most importantly), all of the methods we looked at had comments. A method on the class side called COMMENT is used to provide class comments. Although using a special method for class commentary is somewhat of a hack, the fact that there *is* commentary is an area in which ST/V beats both STA and more recent additions to ST-80.



EXCEPTION HANDLING

One reason some choose ST-80 is its flexible exception handling mechanism, since ST/V had none. Digitalk now has something similar to the PPS context-unwind/marked-method exception mechanism, but they made it somewhat easier to use, at the expense of being somewhat more difficult to create special cases. For example, both exception systems have a special-purpose do-this-thing,-and-then-whatever-happens,-make-sure-to-do-this-other-thing mechanism, since that is a useful way of releasing external resources such as files. In ST-80, the mechanism involved leaks through into the method selector; you need to use the rather unwieldy method:

```
[statement with possible error] valueNowOrOnUnwindDo:
    [statement that always executes]
```

ST/V uses the simpler, more intuitive:

```
[statement with possible error] ensure:
    [statement that always executes]
```

Likewise, a block that is only executed when some other statement has a problem is invoked using valueOnUnwindDo: in ST-80, and ifCurtailed: in ST/V.

Like ST-80, ST/V also has a more general exception mechanism, using a block-centered, class-based approach, rather than the Signal-centered, instance-based approach pioneered in ST-80. For example, in ST-80:

```
Number divideByZeroSignal
    handle: [:exception | exception return: 0]
    do: [x/y]
```

returns zero if y is zero, whereas in ST/V:

```
[x/y]
    on: ZeroDivide
    do: [:exception | exception exit: 0]
```

does the same thing, using analagous components. ST/V is simpler to use in simple cases, but requires more work to set up new exceptions, since they must be coded as complete classes. In ST-80, new exceptions are very simply coded in the class protocol of objects that will be generating them, but are therefore not global, making the user of them work a bit more to protect a block of code.

STA uses a completely different approach, which requires exceptions to be coded as compiler directives. This may be the easiest of all to use, but the STA exception mechanism was undocumented at the time of this review; this opinion is based on browsing examples in the STA image.

SUPPORT

Digitalk provides well-regarded suport through their CompuServe forum. They announce all patches and bug fixes, and allow downloading to registered licensees. This again shows Digitalk's strong PC/clone bias—Mac users are more likely to be on America Online, AppleLink, or Usenet than CompuServe. We've seen grumblings in the Usenet Smalltalk newsgroup about CompuServe users getting more attention than Internet users.

For example, until we asked about the availability of patches, we had no way to know they existed. We were fortunate to have personal contacts among the team responsible for ST/V (special thanks to Pat Caudill, the ST/V-Mac project leader), so we got good response when we inquired about bug fixes. However, we can't write from firsthand experience about general responsiveness of technical support for ordinary customers. We also do not know how the typical customer finds out about the availability of bug fixes, other than via CompuServe—we have been licensed for ST/V-Mac since 1990, and were only notified of major upgrades available for an additional fee.

PROBLEMS

After installation, we carefully write-protected the image, so that we would not inadvertently change something that would affect a measurement. We then experienced repeated application crashes upon launching. This was extremely frustrating, until we discovered that you cannot run a write-protected image.

In ST/V version 1.2, a notifier tells you this and lets you exit; in 2.0, it simply crashes into Macsbug. ST-80 happily runs a write-protected image, allowing you to save to an alternative file name upon save; STA cryptically opens a standard file dialog, and when you choose the write-protected image, it gracefully exits without ever telling you why. We much prefer the ST-80 tactic, and wish other vendors would adopt it.

Other than the write-protect incident, we experienced only one unexplained application crash. On attempting to resize a window on a 28K file, ST/V died with error #25 (dsMemFullErr, out of memory). On the other hand, we experienced numerous walk-backs (with version 2.0) when trying things as reasonable as installing Digitalk-supplied bugfix patches.

PATCHES, PATCHES

At the time this was written, there were two files of bug fixes and a new application and image, together known as version 2.0.2, available from Digitalk. Together, they correct over 70 bugs, classifying about 25% as serious problems (e.g., walk-backs, methods silently misperformed, crashed application).

However, we could not get the patch installer to work by following the supplied README file. Bypassing the installer and manually filing in the patch files worked somewhat, but we also had to update the image's idea of which patches were installed, so that other files (such as the newly patched Compatibility.st) would file in. It took over an hour to install the patches, which seems unreasonable—upgraders should get the new image and application, and not bother with the patches.

Later, we looked more closely at the installer problem. Directory current, used by the installer, answered the top-level directory, not the actual directory from which ST/V started. We moved our directory to the top-level folder, and the patch installer worked. Incidentally, one of the patches corrected this problem—Digitalk's QA department must be using their prescient version of the patch installer!

Fortunately, version 2.0.2 is much more stable, and all the problems we recorded in 2.0 (except for the write-protect crashes) appear to be solved.

COPY PROTECTION

Unlike STA, ST/V is not copy protected, and it can be moved between machines as allowed under the U.S. copyright fair use doctrine. (Our review copy did not come with a license, so we cannot say that this would fall within a strict reading of the Digitalk license, but at least it is possible.)

However, like STA (and other ST/V dialects), certain classes (such as the compiler) are protected, and they cannot be viewed, modified, or debugged. This is lamentable—changing the compiler isn't something one should do every day, but having the option to do so may make some problems simpler. Those who are attracted by Smalltalk's reputation as an open system have ST-80 as their only alternative at this time.

PORTING FROM 1.2 TO 2.0

Digitalk describes 2.0 as a top to bottom rewrite of Smalltalk/V for Macintosh. Like all things in life, this is both good news and bad news. We've tried to give an impression of the depth and breadth of the good news.

The news is not entirely happy for users who have to port their applications from 1.2 to 2.0. The lack of release notes means that the user has no first line of approach to discovering puzzles like removed or changed classes. (In comparison, Object Technology International lists each changed method in their release notes, greatly easing the porting task.)

Several developers we surveyed said the port of their 1.2 code was not simple. "I thought I knew ST/V-Mac really well," said a developer who had been with ST/V-Mac for three years, "but now I'm having to relearn a lot." The many new features are strongly desired in the developer community, but many were bewildered by the magnitude of the changes. A section on porting in Digitalk's othewise fine documentation would go a long way to redress the lack of release notes.

Also, 1.2 contained a popular goodie called Application Browser that aided users in creating file-out packages to rebuild their applications in another image. Not only is this goodie missing in 2.0, but the global name Application is taken for an entirely different purpose, so all files made with the 1.2 Application Browser require editing before installing them in 2.0. Digitalk is experiencing customer pressure, and will probably replace the Application Browser in some form.

MEASUREMENTS

Table 1, which originated in last issue's STA article, compares available Macintosh Smalltalk implementations. Readers should refer to that article for additional information on how we got our numbers. Changes since that article are underlined.

Since last issue's article, we've added two measurements. In an attempt to include some indication of high-level performance, we added a simple timing of how long it takes to open a window on a 28K styled-text file. (Only STA and ST/V 2.0 displayed the text appropriately styled.) While this is not a particularly rigorous test, it tends to parallel the slopstone and smopstone benchmarks against STA and ST/V 1.2, although it counters to some extent the results against ST-80, which has sluggish linear-read file performance due to multiple buffering.

We also added a line for the application (vs. image only) disk file size.

As we mentioned last month, low-level benchmarks do not necessarily predict the performance you'll get. To paraphrase Mark Twain, there are lies, damn lies, and benchmarks. In particular, ST/V (and STA) is much closer to the Mac than is ST-80, so windows open faster and menus drop more quickly than they do in ST-80's Mac emulation.

Most of the measurements of 2.0.2 are within 6% of those of 2.0. There are 503 new methods, which add 118K to the image size. This seems like quite a bit for two minor revision cycles and indicates the youthfulness of the product.

The number of classes can be misleading and does not necessarily correlate with quantity of functionality. Recall that ST/V has a class per exception type, while ST-80 has a method per exception type. Also, 181 classes—nearly a third of the to-

tal—have names that begin with *Mac*, indicating functionality may be distributed among more classes in ST/V than in other Smalltalk dialects.

The first column for each implementation (the percentages) lists that implementation's measurement relative to the greatest in the group. In general, lower is better, although items such as ìnumber of classesî are better the higher the number.

SmalltalkAgents UPDATE

As the deadline for this article arrived, we had just received STA version 1.1b15. Although it is a beta version, QKS assured us that short of crucial bug fixes, what we received is identical to what is shipped in late November as version 1.1. We've updated the measurements table with data from the new version.

The new measurements show that QKS has not been on vacation these two months. Essentially every measurement has improved in that time. The smopstones doubled in performance, while the preferred memory size was reduced 17% and the image plus application size went down 5%. Keep in mind that the removal of internal debugging code probably accounts for much of the size reduction, and possibly some of the speed improvement as well.

More important than the measurements is that STA 1.1b15 feels much more stable than the version 1.0.1 previously reviewed. We experienced no application crashes or debuggers while evaluating version 1.1b15.

Only speed and quality appear to have changed since we last described STA. The biggest omissions of version 1.0.1 still exist: no manual, no GUI builder, no crash recovery strategy, partially implemented inspector and debugger, nonstandard command key map, etc. STA remains a product with exciting potential, suitable for early adopters and pilot projects, but don't bet your business on it just yet.

SUMMARY

Digitalk's Smalltalk/V-Mac version 2.0 is an ambitious upgrade to version 1.2, and is a bargain, even at the full list price. Those with ongoing ST/V-Mac projects should not hesitate to port to it, although such a port may not be trivial. Three times the quantity of code and eased access to the toolbox alone make it worth switching for ongoing development, and the addition of exception handling alone may be enough to justify a port of ongoing work.

However, three times the code potentially means three times the resource consumption. Those with completed ST/V applications may not want to port to 2.0 if RAM usage, disk size, and performance are utmost issues. Those who are supporting an existing application that must work nicely on an old

Table 1. Comparison of available Macintosh Smalltalk implementations.

	STA 1.1b15		ST/V 1.2		ST/V 2.0.2		ENVY/ VisualWorks	
Start-up time[a]	33%	16	15%	7.3	81%	39	100%	48
Image save time[a]	94%	16	19%	3.3	100%	17	94%	16
Slopstone (no FPU)[b]	100%	0.14	38%	0.054	32%	0.045	26%	0.037[d]
Slopstone (FPU)[c]	87%	0.20	30%	0.070	26%	0.059	100%	0.23
Smopstone (no FPU)[b]	100%	0.19	24%	0.046	18%	0.036	21%	0.039[d]
Smopstone (FPU)[c]	100%	0.27	22%	0.060	20%	0.053	81%	0.22
28k file window[a]	35%	1.8	35%	1.8	84%	4.3	100%	5.1[e]
Required memory (K)	85%	3,500	35%	1,465	65%	2,654	100%	4,096
Preferred memory (K)	50%	5,000	20%	1,953	36%	3,584	100%	10,003
Image size (K)	56%	2,316[f]	16%	669	53%	2,191	100%	4,121
Application size (K)	96%	492[f]	17%	88	43%	222	100%	512
Number of classes[i]	42%	340[h]	20%	165	75%	606	100%	811[g]
Number of methods[j]	39%	6,685	20%	3,490	62%	10,693	100%	17,268[g]

[a] Time in seconds, measured with a hand-held stop watch, using a fast 11 ms. disk.

[b] Mac PowerBook Duo 210: 25 MHz 68030, 12 MB RAM, 80 MB disk, 1-bit LCD, System 7.1, no system extensions or power saving.

[c] Mac 11ci: 25 MHz 68030, 20 MB RAM, 32 KB cache, 80 MB and 1.2 GB disks, 1-bit internal video, System 7.1, no system extensions.

[d] The poor result is possibly due to memory starvation. We could not obtain "preferred" memory and had to settle for a mere 9,560K. This result was disturbing, so we repeated it—and gave up after two hours!

[e] Strongly correlated with file sizes. ST-80 was actually as fast as ST/V 1.2 for very small files.

[f] STA uses one file, image size is data fork size, application size is resource fork size.

[g] Includes ENVY, which adds 126 classes and 3,213 methods.

[h] Fewer classes than previous version. A number of demo classes appear to be "unbundled" in files, rather than be delivered in the image, although the method count is actually higher in this version.

[i] Obtained via MetaClass allInstances size, which may include classes with hidden source code.

[j] Obtained via CompiledMethod allInstances size, which may include methods with hidden source code.

Plus or Classic might consider staying with 1.2.

Those considering new development need to ask a number of questions before settling on ST/V. Are you willing to watch CompuServe for the inevitable patches and bug fixes that come with a young product? Do you need the utmost performance? Are you already comfortable with another vendor's product? Will you be working with a team? Do you need to work on multiple platforms? None of these questions alone may rule out ST/V, but if many of them are issues, a different vendor's Smalltalk may be appropriate for you. ■

Jan Steinman is a partner in Bytesmiths, a consulting company that specializes in helping organizations start new Smalltalk projects. Jan has over 11 years of object experience in embedded systems, instrumentation, scientific visualization, finance, and telecommunications. Prior to forming Bytesmiths, he was project leader for Tektronix's monochrome Smalltalk virtual image. He can be reached at jan.-bytesmiths@acm.org.

Rob Vens

European Smalltalk Summer School

One of the most satisfying things to do as a Smalltalk programmer is to be immersed in Smalltalk for five days and nights. And the perfect venue for such an activity was the first Smalltalk Summer School, organized by the European Smalltalk User Group (ESUG). At a beautiful location in Brest, on the campus of Telecom Bretagne on the French coast of the Atlantic Ocean, 23 students from five European countries attended five days of intermediate-to-advanced-level tutelage and pure fun. About half came from universities and half from industry.

ESUG was fortunate in acquiring tutors like Trevor Hopkins, Mario Wolczko (both of Manchester University, the latter well-known as maintainer of the Manchester Smalltalk archives) and Patrick Barril from University Pierre-et-Marie Curie in Paris. The courses were not committed to any Smalltalk dialect, with emphasis on the two major dialects.

I would like to share with you some of my experiences of this Summer School, and give some impression of how Smalltalk is faring in Europe. The three major parts of the program were tutorials, workshop, and demos. Three days were devoted to tutorials. On the first day, we were thoroughly informed about the Smalltalk\Objectworks imaging model and windowing interfaces by Mario Wolczko. Using a wealth of examples from the Manchester archives, we passed the evening experimenting on the excellent computer facilities that were provided.

Patrick Barril provided remarkably deep insight into the inner workings of the Smalltalk/V virtual machine. It appeared that much is possible on this normally avoided level of Smalltalk, and he in fact gave many the impression that there is no reason at all to leave the virtual machine untouched.

On the second day, Mario treated us to an assortment of advanced programming techniques, like the effective use of blocks, exception handling, metaclasses, weak references, and binary storage. He also informed us of much-needed techniques to measure and improve performance. The general attitude of the attendees, most of whom were working on industrial-level projects, was the need to produce industrial-quality applications. Mario's tutorial gave all of us better instruments to achieve this goal.

The third day was devoted to demos. As well as showing interesting applications, these demos were good opportunities for discussions with vendors. In the context of a summer school session, much more fruitful interaction with vendors is possible than at large conferences. They certainly could not get away with cheap sales pitches, but were confronted with critical and knowledgeable customers!

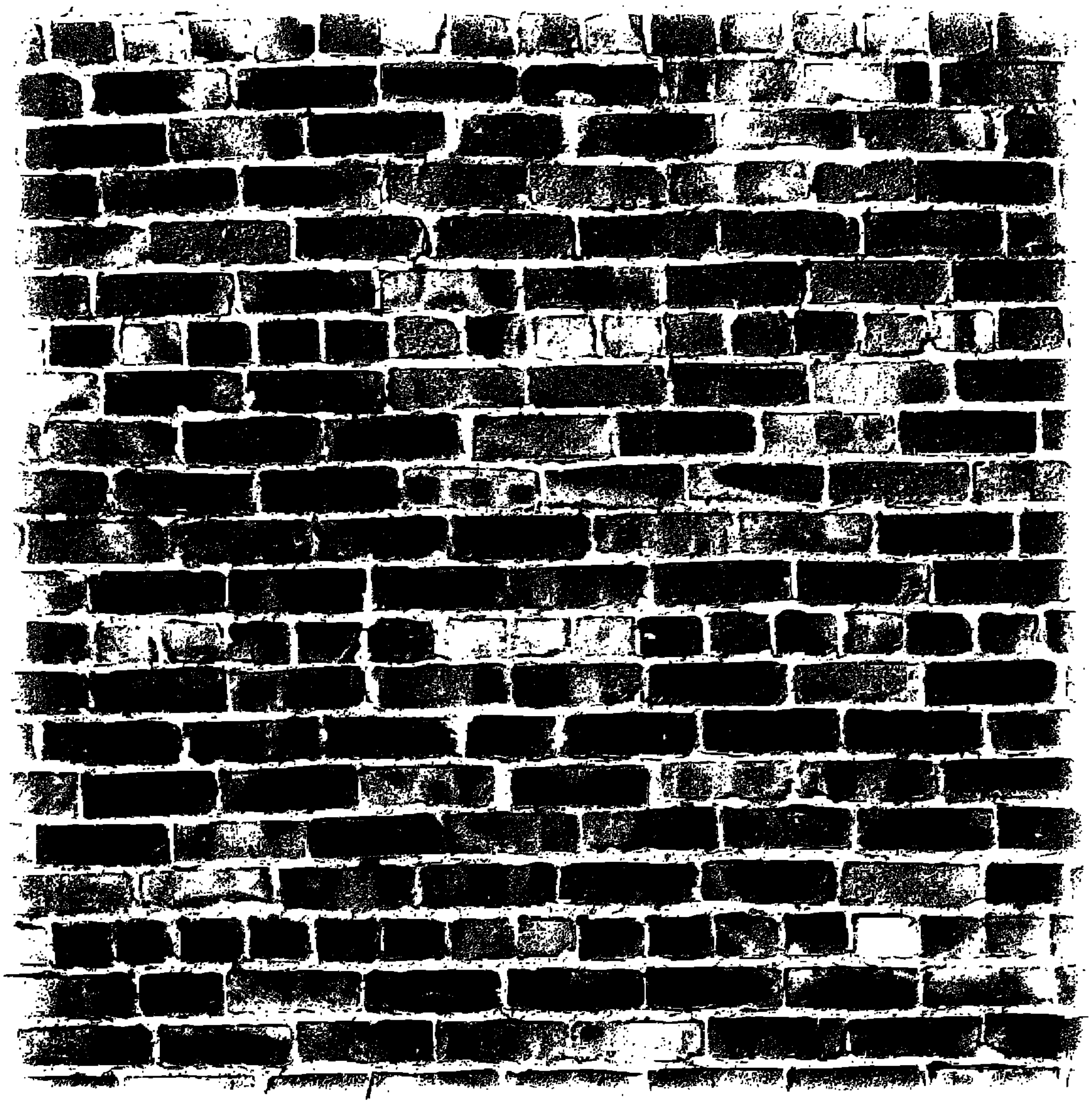
As was to be expected, there were demos of applications to link Smalltalk with the rest of the world. Patrick Barril showed Digitalk's PARTS, provided courtesy of the French distributor Tau Ceti S.A., as well as Smalltalk/V for several platforms. Clearly Digitalk is moving toward instance-based programming, which resulted in heated discussions about the difference between their approach and ParcPlace's MVC paradigm. Georg Heeg Co., a distributor and vendor from Germany, showed an implementation of distributed processing in Objectworks-Visualworks Smalltalk called Remote Objects. Servio demonstrated their database management system, GemStone. A small company from the Netherlands showed a tool for project management in Smalltalk/V, called SmallTool—something much needed by teams working on Smalltalk applications. In the same vein, Georg Heeg Co. demonstrated a new tool called Application Management.

On the fourth day, Trevor Hopkins gave a presentation on Smalltalk's contribution to the hot issue of client-server computing. Providing us with advanced techniques, we felt very inspired to apply these techniques in our own work. Those of us working in the field were better equipped to deliver higher-quality applications. Mario continued his tutorial on interface-construction and MVC in the afternoon. We certainly did not have enough time on the computers to try out all his examples!

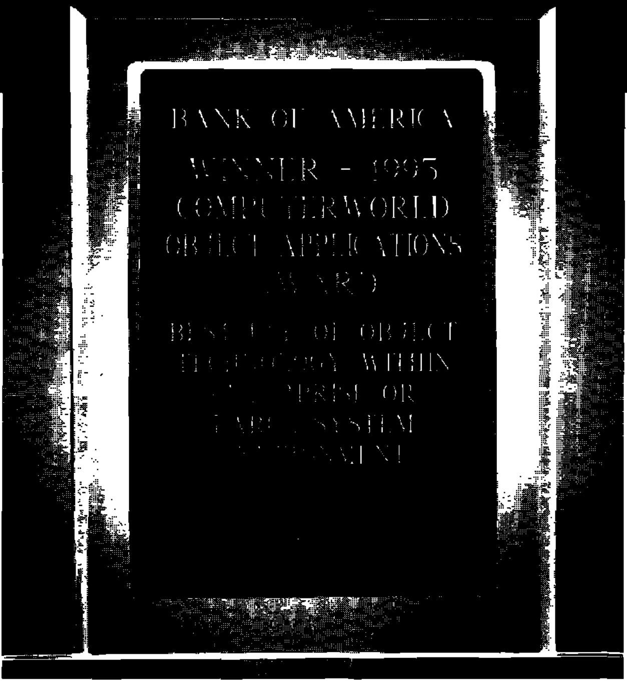
On the last day, two parallel workshops addressed the issues of industrial problems like project management, fast development under heavy time constraints, and research issues like parallel programming, constraints, and simulation techniques. In these workshops, we were able to share our own experiences in these areas. It became clear during the resulting discussions that the Smalltalk world is moving rapidly into commercial areas like banking and consulting. This creates specific problems and demands for Smalltalk vendors. For this, a Smalltalk users group is very important.

Ending a week of activity with this rate and intensity of interaction is always difficult and somewhat painful. But all attendees and tutors agreed that a new summer school next year will definitely be organized. Cork, Ireland is a likely location. I certainly hope to see some of this year's attendants there, as well as many more new ones! ■

Rob Vens is secretary of USUG. He is a researcher on the Faculty of Management of the University of Groningen in The Netherlands and can be reached by email at R.W.Vens@bdk.rug.nl. ESUG, can be reached via email at esug@ibp.fr.



LOOK WHAT HAPPENED WHEN DIGITALK BROKE INTO THE BANK.

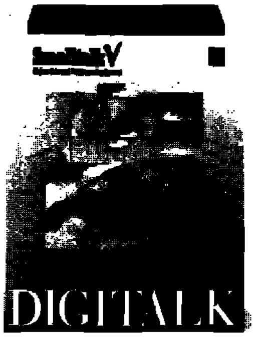


Congratulations to Bank of America on their new 11-state wide area network. A system they call "the most sophisticated distributed network in the world."

With good reason. Their network configuration tools have already won the Computerworld 1993 Award for Best Use of Object-Oriented Technology within an Enterprise or Large System Environment.

Of course, that's what happens when a company like Bank of America turns to a powerful technology like Digitalk's Smalltalk/V.

LIKE MONEY IN THE BANK.



Why are so many Fortune 500 companies like B of A switching to Smalltalk/V? Smalltalk/V lets you show prototypes of enterprise-wide systems in weeks instead of months. In fact, systems as ambitious as Bank of America's can be completed in as little as 18 months.

In addition, our Team/V Group Development Tool lets large teams of programmers use version control to easily coordinate their work. Plus you'll be surprised at how quickly your in-house staff becomes productive with Smalltalk/V.

The bottom line is Smalltalk/V helps a company get more done in less time. Which can save very large amounts of corporate cash.

RATED #1 BY USERS TOO.

On behalf of Computerworld, Steve Jobs presented the award to Bank of America. But industry luminaries and Fortune 500 managers aren't the only ones who have recognized the value of Smalltalk/V. Users have discovered that Smalltalk/V is the only object-oriented technology that's 100% pure objects. With hundreds of reusable classes of objects, thousands of methods and 80 object classes specifically designed to build GUIs fast. Which means no more time spent writing code from scratch.

BANK ON SMALLTALK/V.

So it's no wonder that so many companies are doing award-winning work with Smalltalk/V. Incidentally, Smalltalk/V applications can be easily ported between Windows, OS/2 and Macintosh. And you can distribute 100% royalty-free.

For information on how Digitalk's Smalltalk/V can save you time and money, call 1-800-531-2344 department 310 for our special White Paper. And be sure to ask about Digitalk's Consulting and Training Services.

Call right now, and see how Smalltalk/V can yield a maximum return on your investment.

SMALLTALK/V. 100% PURE OBJECTS.

DIGITALK