The Smalltalk Report

The International Newsletter for Smalltalk Programmers

January 1993 Volume 2 Number 4

Taking Exception to Smalltalk, Part 2

By Bob Hinkle & Ralph E. Johnson

Contents:

In the last issue (Nov./Dec. 1992 Smalltalk Report), we described the system-independent parts of our implementation of an exception-handling system. This month's article describes those parts of our implementation that rely on Smalltalk V/286 specifics, with special emphasis on contexts. We will describe the architecture of process stacks, how contexts fit onto that stack, and how temporaries are laid out in contexts. The ability to examine and manipulate contexts is both powerful and useful and has been exploited in two other programming efforts that we know of. The first and foremost is the system debugger, which uses all of contexts' capabilities, including the modification of local variables and the resumption of execution at any point in the stack. The other example is a backtracking system for Smalltalk developed by Wilf LaLonde and Mark Van Gulik,[1] which, like our exception handler, uses contexts to implement non-standard control flow.

The last few pieces to our implementation are system-specific methods in the class Exception, extensions to the fundamental classes Process, Context, and HomeContext, and the addition of three new context-related classes. The changes in these classes are extensions to Digitalk's base that make processes and particularly contexts easier to work with. The same changes might not be necessary in another implementation of Smalltalk; in particular, ParcPlace's Smalltalk-80 provides all the functionality we need and more.

THE MACHINE-DEPENDENT IMPLEMENTATION

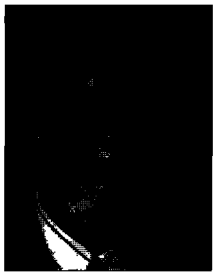
We still need three Exception methods to describe: fetchHandlerBlock:, restart, and return. Each of these methods depends on some specific aspects of V/286.

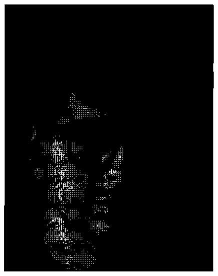
We begin with fetchHandlerBlock:, the method used by propagatePrivateFrom:, to find the correct handler for the receiving exception. FetchHandlerBlock: is implemented as:

```
fetchHandlerBlock: startContext
    startContext sendersDo: [ :ctxt |
        (ctxt selector == #handle:do:
            and: [ctxt receiver accepts: signal])
            ifTrue: [handlerContext := ctxt.
                ^ctxt at: 4]].
    ^nil
```

In general, startContext will be the value of the exception's signalContext instance variable, which is the context of the raise message. The message sendersDo: is used to iterate down the context stack from startContext, applying the block to each context in turn. The block checks each context looking for a handler for the exception; the correct handler context will be the first one reached where handle:do: was sent to the exception's signal or one of its parents (which is what the accepts: method checks for). When such a context is found, it's remembered as the handlerContext, and the object in its fourth slot is returned. This object will be

continued on page 4...


John Pugh


Paul White

EDITORS' CORNER

Smalltalk has many advantages, but what about all these stories of performance problems? This is a common refrain among companies considering Smalltalk as a mainstream development vehicle and certainly a common battle song for those who feel compelled to enter the language wars debate on the side of C++. We'll stick to the high ground and steer clear of language comparisons. Experienced Smalltalk programmers know that the time taken to produce a working application is of paramount importance and can often be cut by an order of magnitude by using a productive programming environment. We know that the performance of most systems can be improved enormously by examining the small fraction of the code where the application spends most of its time. Performance problems are most often due to the implementors selecting a bad design or choosing an inefficient algorithm rather than any deficiency of the programming language.

What's the reason for this diatribe? Well, most companies considering Smalltalk don't have the benefit of our experience. Indeed, many of our readers would also welcome being able to share in the experiences of others. We would like to solicit "experience reports" for publication in The Smalltalk Report that focus on practical issues. Performance is but one topic of interest. Here's a list of others to get you started:

- The use of Smalltalk in domains as diverse as banking and computer integrated manufacturing.
- Reuse. What has been achieved? How quickly has it been achieved? What mechanisms were used to promote use?
- Whan performance problems have arisen, what has been their cause and what steps were taken to identify and rectify the problems?
- Experiences in linking Smalltalk with existing legacy systems.
- Experiences in linking Smalltalk with components developed in languages such as assembler, C, or COBOL.
- Experiences in proting Smalltalk applications from one platform to another.
- Experiences in using team programming tools.
- Experiences in managing large Smalltalk projects.
- What metrics are useful? Have useful metrics emerged from projects?
- What OOA and OOD methodologies are being used in Smalltalk projects?

Last month, Bob Hinkle and Ralph Johnson described the system-independent components of their exception handling system. In the second and final part of "Taking Exception to Smalltalk," Bob ans Ralph complete their description by describing the components of the system that are specific to their target environment, Smalltalk/V 286. These articles have again highlighted one of Smalltalk's often overlooked advantages; the fact that "system" components such as processes and contexts are Smalltalk objects. This means that they can't be manipulated like any other Object in Smalltalk and permits Smalltalk programmers to augment Smalltalk with new "control-flow" facilities such as backtracking and exception handling. Look for more articles from Bob and Ralph on the reflective nature of Smalltalk in future issues.

In our second feature article this month, Mary Beth Rosson from the IBM T.J. Watson

continued on page 18

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by SIGS Publications Group, 588 Broadway, New York, NY 10012 (212)274-0640. © Copyright 1993 by SIGS Publications, Inc. 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 publishers. Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada.

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
Chuck Duff, The Whitewater Group
Adele Goldberg, ParcPlace Systems
Tom Love, OrgWare
Bertrand Meyer, ISE
Meilir Page-Jones, Wayland Systems
Sesha Pratap, CenterLine Software
P. Michael Seashols, Versant
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, Carleton University
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
Karen Tongish, Production Editor
Jennifer Englander, Art/Prod. Coordinator

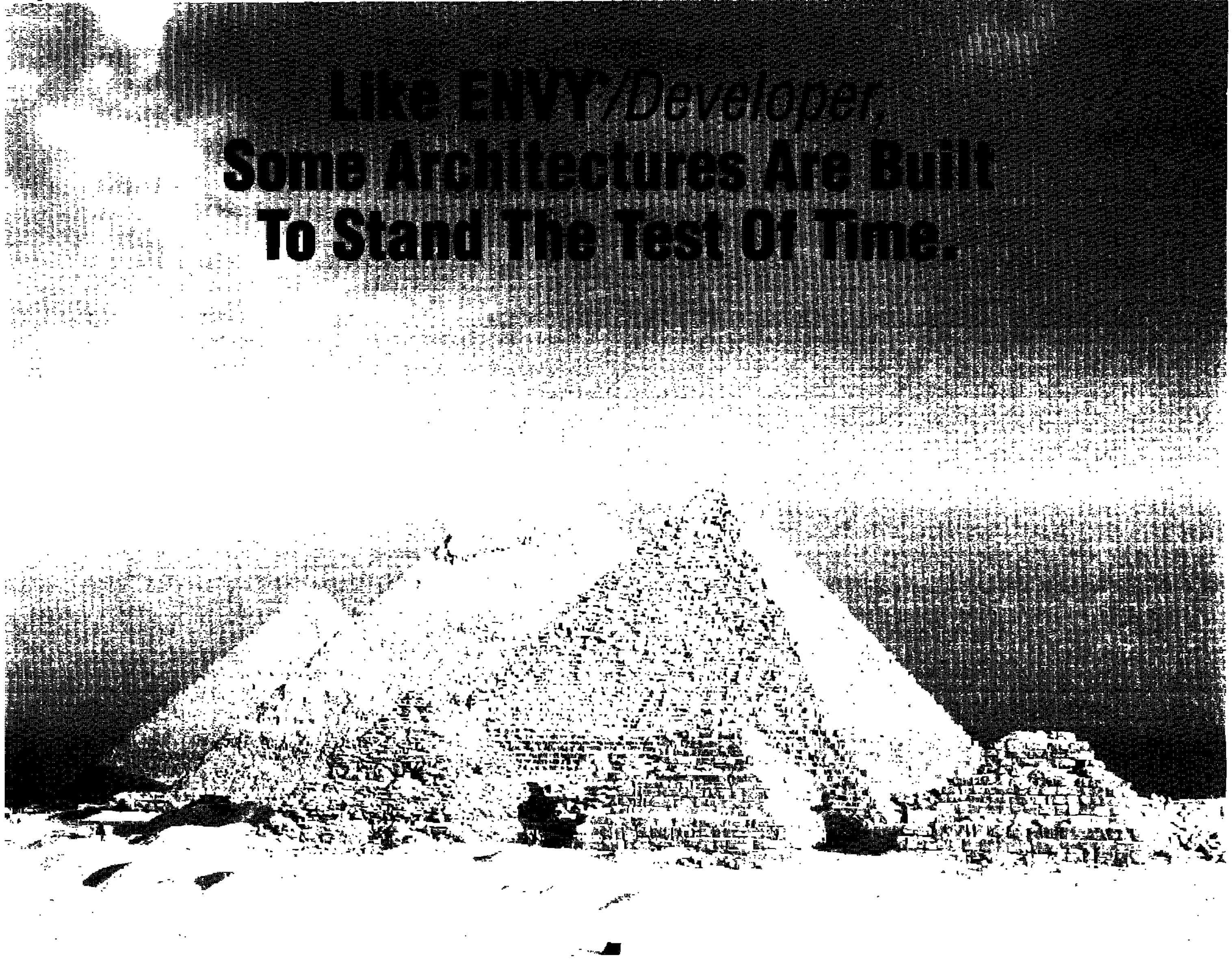
Circulation
Ken Mercado, Fulfillment Manager
John Schreiber, Circulation Assistant
Vicki Monck, Circulation Assistant

Marketing/Advertising
Diane Morancie, Advertising Mgr—East Coast/Canada
Holly Meintzer, Advertising Mgr—West Coast/Europe
Helen Newling, Exhibit/Recruitment Sales Manager
Sarah Hamilton, Promotions Manager—Publications
Lorna Lyle, Promotions Manager—Conferences
Caren Polner, Promotions Graphic Artist

Administration
Ossama Tomoum, Business Manager
David Chatterpaul, Accounting
Claire Johnston, Conference Manager
Cindy Baird, Technical Program Manager
Amy Friedman, Projects Manager

Margherita R. Monck
General Manager

SIGS PUBLICATIONS



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

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

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

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

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

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

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

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

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

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

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

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

ENVY is a registered trademark of Object Technology International Inc. All other brand and product names are registered trademarks of their respective companies.

...continued from page 1

the block that was passed as the first parameter of the handle:do: message. It is fourth because of the order Digitalk stores local variables in contexts, with first slots for block arguments in reverse order of their appearance, followed by temporaries in reverse order, then parameters in reverse order. Figuring this out requires some knowledge of the context layout in V/286; we'll describe that in more detail when we discuss contexts below.

The return method is implemented in terms of returnDoing:, which itself is implemented as follows:

```
returnDoing: aBlock
    "The stack is unwound to the context of the handle:do:
    message that caught this Exception, at which point
    aBlock is evaluated and its value returned as the value
    of the handle:do: message."
    | answer |
    answer := aBlock value.
    self handlerContext unwindLaterContexts.
    (self handlerContext at: 2) value: answer
```

This is analogous to the implementation for proceedDoing:. The only difference is in accessing the block that will (when evaluated) return into the right context. In this case returnBlock is stored in the handler's context (recall our definition of handle:do: from Part 1 of this article) and is accessible in the second slot of the context's array of temporaries. So evaluating the returnBlock returns from the handle:do: context as desired. As with proceedDoing:, though, the method must call unwindLaterContexts first to make sure unwind blocks are evaluated.

Implementing restart relies on restartAt:, a Digitalk-provided method for class Process, as seen in the following code:

```
restart
    "Restart the #handle:do: context."
    | index process |
    handlerContext unwindLaterContexts.
    process := handlerContext process.
    index := process frameIndexOf: handlerContext.
    process restartAt: index
```

This method makes a process restart execution at an arbitrary context in its context stack. First, as before, the exception unwinds all contexts above its handlerContext. The exception then finds the handlerContext's index in its process, and tells its process to restart execution there.

To motivate the changes to the Process and Context classes, we need first to describe how these classes relate in the base system. Process in V/286 is a subclass of OrderedCollection; its indexed instance variables are used to store information about the stack of unresolved message sends. Conceptually, we think of each message send as being represented by a Context object. However, for optimization purposes, V/286 only creates HomeContexts for certain method invocations. (In particular, they create a HomeContext only if the method that's evaluated contains a block.) This dual representation is potentially troublesome, so we hide it behind two new context-related classes. Before looking at these classes, though, we need to understand the layout of Process' stack. Each message send receives a stack frame of five or more slots on the stack, with the layout shown in Figure 1.

Each stack frame begins with the frame address of the previous message. This frame address is unique for each message send and persists as long as the message is on the stack. In addition to this address, message sends can be referenced by their frame index, which is the message send's position on the stack. The topmost (i.e., most recent) send is at frame index 0, the previous send is at index 1, and so on.

After the frame address comes the byte array (Smalltalk's compiled representation of the method), the compiled method, the instruction counter, and the message's receiver. If a HomeContext exists for the frame, it will be stored in the sixth slot. If there is no HomeContext for the frame then there will be slots for each of the parameters and temporary variables (in the same order referred to above: block arguments in reverse order followed by temporaries in reverse order followed by parameters in reverse order).

The Process class provided by Digitalk comes with a method, contextFor:, which returns the context for a given frame index. This has two problems for our purposes: First, the context may not exist, in which case nil is returned; second, the context returned for blocks (instances of class Context in V/286) is their HomeContext, which is not the same as their frame on the context stack; and third, since the frame index changes as execution proceeds, we really need to use the frame address to identify our contexts. So we added three methods to Process: indexOfAddress:, frameContextFor:, and groundedContextFor:. The method indexOfAddress:, which converts a frame address into a frame index, first checks if the input frame address matches the address of its top frame. If it does, the receiving process returns 0 as the associated frame index. Otherwise the process returns the index of the first frame whose frame address matches the input.

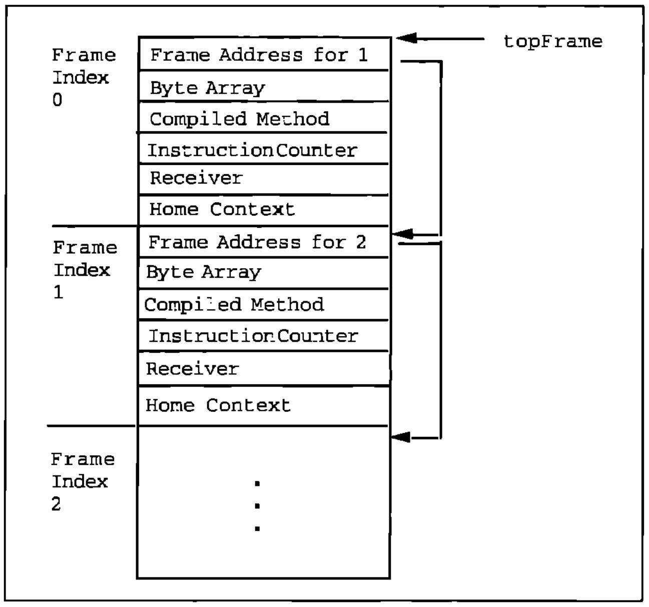


Figure 1. Process stack layout.

```
indexOfAddress: frameAddress
    | index address |
    frameAddress = topFrame
        ifTrue: [^0].
    index := 1.
    [address := self frameAt: index - 1 offset: 0.
    frameAddress = address]
        whileFalse: [
            address = 0
                ifTrue: [^nil].
            index := index + 1.
        ].
    ^index
```

We discovered the need for the frameContextFor: method by trial and error. We had problems in an early version of the system with the fetchHandlerBlock: method because our search down the context stack would sometimes skip over one or more contexts and occassionally fail to find the right exception handler. We found that contextFor: was the culprit because when called on the index of a block's stack frame, it always returns the block's HomeContext, while we wanted the frame for the block's own activation. The frameContextFor: method overcomes this problem by only returning the HomeContext if it exists and if the stack frame in question is its own home frame, as follows:

```
frameContextFor: frameIndex
    "Answer the context object from the stack frame at
    frameIndex, but only if that stack frame is its own
    home frame. In particular, don't return the HomeContext
    for a block's frame."
    ^(self methodAt: frameIndex) hasBlock
        ifTrue: [
            (self homeFrameOf: frameIndex) = frameIndex
                ifTrue: [self frameAt: frameIndex offset: 5]
                ifFalse: [nil]]
        ifFalse: [nil]
```

The other addition to Process is a method called groundedContextFor:—an extended version of contextFor:., from which groundedContextFor: differs only in that it always returns a context-like object, whether or not a real HomeContext exists for the frame requested:

```
groundedContextFor: frameIndex
    | context frameAddress |
    context := self frameContextFor: frameIndex.
    ^context isNil
        ifTrue: [
            frameAddress :=
                (frameIndex = 0)
                    ifTrue: [topFrame]
                    ifFalse: [self nextFrameAt:
                        frameIndex - 1].
            frameAddress = 0
                ifTrue: [nil]
                ifFalse: [PseudoContext
                    forFrame: frameAddress
                    forProcess: self]]
        ifFalse: [GroundedContext
            forContext: context
            forProcess: self]
```

This method makes use of the two new classes we added. A GroundedContext object is returned when a real HomeContext is available for the frame. (We return a GroundedContext rather than the HomeContext itself because GroundedContexts have additional behavior—they particularly have an instance variable for the process they belong to.) When a real HomeContext is not available for the frame, frameContextFor: returns nil, and in that case the method creates a PseudoContext object. This object knows only the frame address of the frame it represents, but using the frame address it can behave exactly like a normal context. Thus, PseudoContexts and GroundedContexts seem identical externally, hiding the difference between those frames with and without HomeContexts.

PseudoContext and GroundedContext are designed to fulfill the same interface, so we also created an abstract class called AbstractContext, which is a common superclass for the two. AbstractContext defines the interface and implements a number of methods by depending on a few methods from its subclasses. In particular, it defines the method unwindLaterContexts as:

```
unwindLaterContexts
    "Search down the stack, starting with the current
    context, evaluating the unwindBlock in every
    Context>>valueOnUnwindDo: or
    Context>>valueNowOrOnUnwindDo: context. Stop at
    the receiver."
    | s |
    self thisContext
        sendersDo:
            [:ctxt |
            ctxt == self ifTrue: [^self].
            (ctxt receiver isKindOf: Context)
                ifTrue: [(s := ctxt selector) ==
                    #valueOnUnwindDo:
                    ifTrue: [(ctxt at: 1) value]
                    ifFalse: [s == #valueNowOrOnUnwindDo:
                        ifTrue: [(ctxt at: 2) value]]]]
```

This is similar to the fetchHandlerBlock: method. It looks down the message stack starting at the current context, looking for any context for the valueOnUnwindDo: or valueNowOrOnUnwindDo: messages. If it finds one, it evaluates the unwind block, which is available in the first slot of the valueOnUnwindDo: context or the second slot of the valueNowOrUnwindDo: context.

In addition, AbstractContext defines the sender method as follows:

```
sender
    ^process groundedContextFor:
        (process indexOfAddress: self address) + 1
```

AbstractContext defines the at: and at:put: methods to provide access to its underlying context's array of temporary variables. The at: method is implemented as:

```
at: anInteger
    | index |
    index := process indexOfAddress: self address.
    ^process tempAt: index number: anInteger
```

at:put: works the same way except that it stores into the slot

continued on page 9

Minimalist Instruction for Smalltalk

Mary Beth Rosson

As the object-oriented paradigm's potential begins to be realized in commercial development (see Bran Selic's article in the September 1992 SMALLTALK REPORT), companies are exploring the use of Smalltalk or other object-oriented languages in their own development activities. A major roadblock, however, is still the training of developers: Learning Smalltalk requires much more than simply learning the syntax or even a rich class hierarchy—it involves internalization of an entirely new approach to software design. For the last several years, our project at the IBM T. J. Watson Research Center (involving John Carroll, Sherman Alpert, and Kevin Singley) has been developing an instructional approach to Smalltalk for building an understanding of object-oriented design from the very start.

THE LEARNING CONTEXT

Our project began by considering a critical group of Smalltalk learners—professional programmers already experienced in the use of procedural languages. These potential users represent the bulk of the commercial software development population and are likely to have significant technical credibility in their organizations; they would thus have a major impact on decisions concerning adoption of this new technology.

Experienced professional programmers are experts confident of their abilities to design software solutions to just about any problem. They typically have learned many languages during their education and careers and have high expectations about their abilities to acquire new languages. They are attracted to new tools, especially ones that promise to speed up or otherwise facilitate the process of designing and implementing software.

Most experienced programmers initially approach Smalltalk with enthusiasm. Indeed, a first encounter may seem promising: Smalltalk syntax is not especially exotic and our observations suggest that an experienced programmer can make some sense of Smalltalk code with little specific instruction (e.g., recognizing that a temporary variable is being manipulated and then evaluated via a conditional expression). Problems arise when programmers attempt to trace through a bit of code involving one or more instances of other classes. At this point, learners are confronted by the extensive class hierarchy and realize how little they understand the objects that might participate in a typical Smalltalk application. Worse, the skills needed for tracking down this information (e.g., heuristics for identifying collaborating objects) are completely lacking. The net result is that new users often "disappear into the hierarchy," spending weeks or months in aimless browsing of the methods in one class and then another.

An inability to make progress quickly in learning Smalltalk is especially frustrating and ironic given Smalltalk's reputation as a rapid prototyping environment. New users expect Smalltalk to help them quickly build and experiment with prototype applications, but they soon discover this "quickness" comes at considerable cost. Indeed, the classes and frameworks used for implementing user interfaces, a key aspect of any application prototype, are among the most complex in the system; new users are typically advised to delay exploration of the user interface code until they are comfortable with simpler, more conventional classes like Collection.

The problem here is not simply that user interface functionality is inherently complex (this is true, but most expert programmers have encountered complex user interface code before). It is that analysis of an interactive application requires a pre-understanding of object-oriented design; how responsibility for complex operations is typically shared by a large number of cooperating objects, normally instances of classes distributed over disparate pieces of the class hierarchy; and how communication patterns among these objects are established and exercised.

THE MINIMALIST APPROACH

A traditional approach to the teaching of complex skills like Smalltalk programming is to decompose the skills and knowledge needed into more manageable pieces, and then gradually recompose those pieces until (realistic) complex tasks can be attempted.[1] For Smalltalk, this translates into teaching about syntax, foundation classes, etc., prior to application structure and the design of new classes or frameworks. Such an approach certainly seems rational, but can easily backfire. Learners must delay the gratification that comes from working on realistic projects; if instruction is not carefully managed, learners may not experience their work with individual components as meaningful, and may have difficulty later on fitting the bits and pieces of instruction into the "big picture" of Smalltalk programming and design.

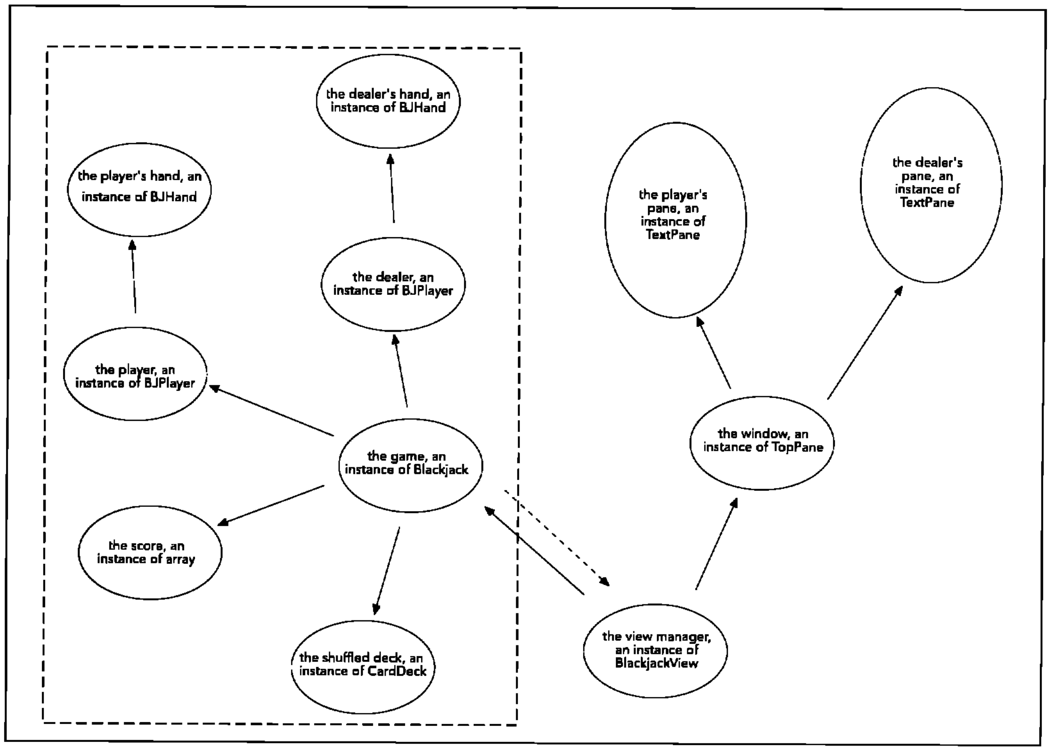
In the Minimalist model of instruction, knowledge and skills are not broken into components.[2] Instead, all instruction occurs in the context of realistic (and hence meaningful) tasks. Of course, because the goal is to let learners work on realistic tasks as quickly as possible, and because realistic tasks will be too complex for a new user, the instruction must provide considerable support. In general, it should filter and organize information about the new domain so that the learner encounters and attends to just the new information needed for the task in progress. The task itself should be chosen and managed to build on whatever prior knowledge the learner can be assumed to have, as this will make the task more familiar and simple. To make the task as meaningful as possible, the instruction should encourage inferences on the part of the learner; this will serve to connect the material being learned to other knowledge and make it more robust and relevant in future tasks. Finally, because errors certainly will occur, the instruction should anticipate them and provide recovery assistance.

MITTS: MINIMALIST TUTORIAL AND TOOLS FOR SMALLTALK

Early interactions with the blackjack application focus on understanding how blackjack works at a functional level. A simplified version of the game (having no interactive user interface) is played in a workspace by creating objects and sending messages. The learners are provided with a filtered view of the hierarchy (a Bittitalk Browser) listing just the most important classes used in the blackjack game. They are given an example of how to create a blackjack object and send it messages, then are expected to use the browser to find additional messages needed to complete the game. By asking the learners to identify and generate the message expressions themselves, the instruction encourages inference as well as the development of browsing skills. The browser hides the actual code for methods, preventing the programmers from becoming distracted by efforts to trace the implementations of messages. The instruction manual is also populated with various "hints" positioned at likely trouble spots.

Subsequently, learners move to analysis of the complete interactive version of blackjack. Their interactions with the simplified version are now part of their prior experience and serve as a foundation from which to understand the more complex interactive application. They are again supported by the environment, in this case the View Matcher tool, which displays and manages several views of the blackjack application as the game is played. (In Figure 2 the upper right subpane holds the blackjack game; the upper left holds a message execution stack; the lower right holds a filtered browser synchronized with selections in the stack; and the lower left holds blackjack-specific commentary.) One of these views is a Bittitalk Browser (this time including the user interface classes); using this browser, learners can extend their model of the blackjack game to include user-interface classes.

Analysis of message-passing patterns among participating objects in the blackjack game is organized by a message execution stack, a deliberate attempt to build on programmers' general understanding of execution stacks. Preset breakpoints cause the game to halt and an execution stack to be displayed, so that learners can see "into" the application to understand how a variety of objects are passing messages to one another to provide blackjack functionality. Selection of messages in the stack is synchronized with se-


the dealer's hand, an instance of BJHand
the player's hand, an instance of BJHand
the player's pane, an instance of TextPane
the dealer's pane, an instance of TextPane
the dealer, an instance of BJPlayer
the player, an instance of BJPlayer
the game, an instance of Blackjack
the window, an instance of TopPane
the score, an instance of array
the shuffled deck, an instance of CardDeck
the view manager, an instance of BlackjackView
Figure 1. The main objects in the blackjack game.

SIXGRAPH™ **Smalltalk/V users: the tool for maximum productivity**

- Put related classes and methods into a single **task-oriented object** called application.
- **Browse** what the application sees, yet easily move code between it and external environment.
- Automatically **document** code via modifiable templates.
- Keep a **history** of previous versions; restore them with a few keystrokes.
- **View** class hierarchy as graph or list.
- **Print** applications, classes, and methods in a formatted report, paginated and commented.
- **File** code into applications and **merge** them together.
- Applications are unaffected by **compress log change** and many other features..

Imager — **Browsers..** — Class; **Application** — **Deleted classes**, **Deleted methods**; Yarn; History — **Code recovery**
Imager — **Utilities..** — Application printing **and more..**

CodeIMAGER™ V286, VMac $129.95
VWindow & VPM $249.95
Shipping & handling: $13 mail, $20 UPS, per copy
Diskette: ☐ 3^1/2 ☐ 5^3/4

SixGraph™ Computing Ltd.
formerly ZUNIQ DATA Corp.
2035 Côte de Liesse, suite 201
Montreal, Que. Canada H4N 2M5
Tel: (514) 332-1331, Fax: (514) 956-1032
CodeIMAGER is a reg. trademark of SixGraph Computing Ltd.
Smalltalk/V is a reg. trademark of Digitalk, Inc.

lection of classes in the browser, helping to direct the learner's attention to relevant information. Blackjack-specific commentary is provided to help make connections between the visible state of the game, the message execution stack in place, and the classes and methods present in the browser.

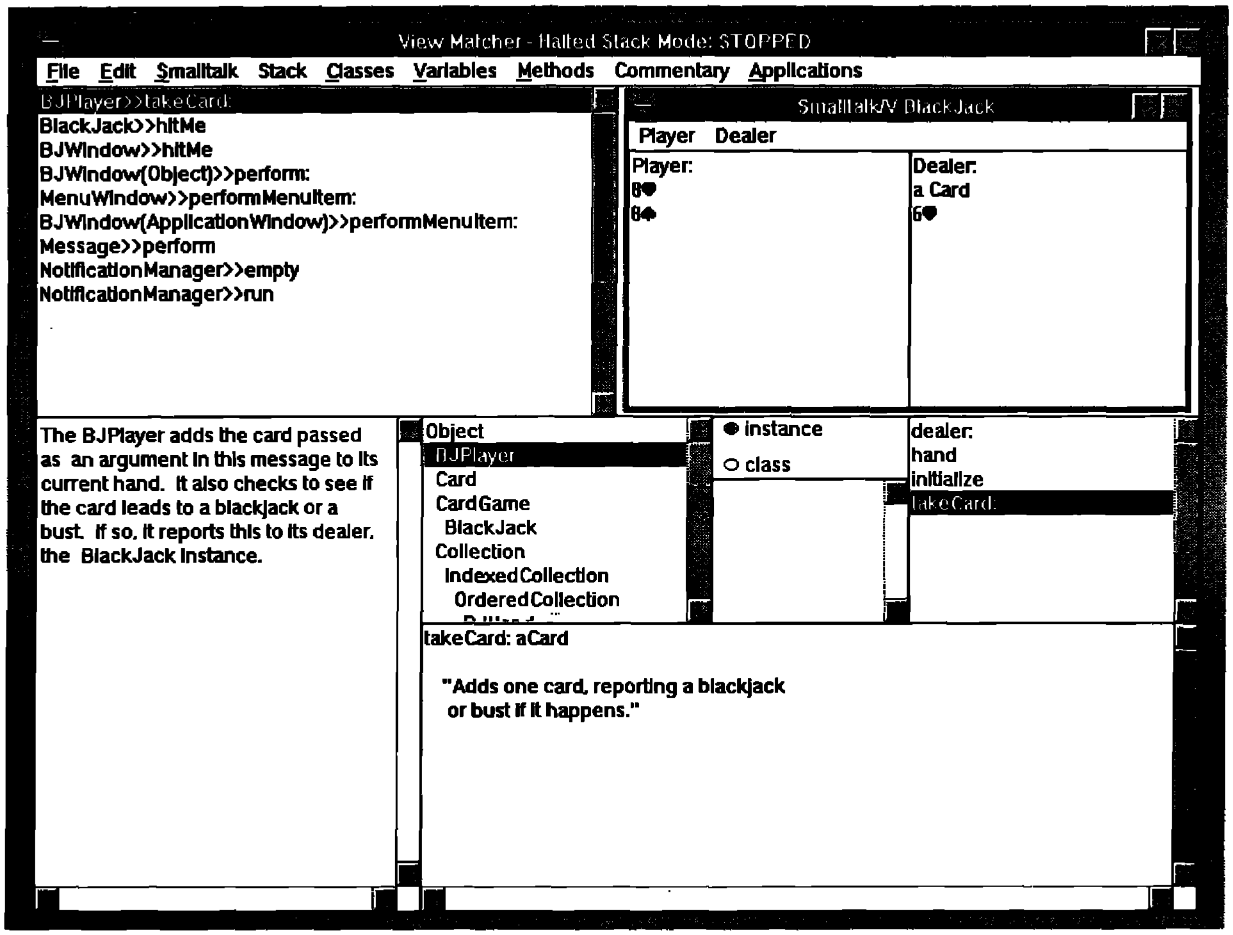
Only after considerable analysis of the structure of the blackjack application (and through this analysis, a contextualized introduction to major concepts like class instantiation, message passing, inheritance, and the Smalltalk/V PM user interface framework) are learners introduced to the actual Smalltalk "code" implementing blackjack. Consistent with the minimalist focus on realistic tasks, syntax and code comprehension are learned in the context of actual programming activities, in which first basic blackjack functionality and then features of its user interface are enhanced.

After four to six hours of blackjack analysis and enhancement, learners have acquired the basics of how to understand and modify an existing application. Because the Bittitalk Browser and the View Matcher are built from standard system tools (the class hierarchy browser and the debugger, respectively), learners are well-prepared to carry out similar tasks in the standard environment. They are supported in making this transition via several open-ended projects that point out the similarities between learning tools and the standard environment, as well as introduce new portions of the class hierarchy through analysis, modifications, and extensions to additional example applications.

EVALUATING MITTS

We have observed dozens of programmers working through MiTTS and for the most part have been pleased at its effectiveness. Well-motivated, experienced programmers can spend a day or so with MiTTS materials and end up with a good understanding of the blackjack application, as well as a general appreciation of application structure and skills for analyzing and modifying applications. As one would expect, the level of accomplishment varies considerably: The most successful individuals are those willing to experiment with new concepts under conditions of uncertainty. Many users find even our "minimal" instruction too extensive, so we are considering preparing a version that simply offers example applications, learning tools, a few orienting words, and some suggested projects.

We have noticed that MiTTS materials support programming by analogy—learners can (and do) use the example applications as models when new problems are encountered. This suggests that a simple extension to the work might be to


View Matcher - Halted Stack Mode: STOPPED
File Edit Smalltalk Stack Classes Variables Methods Commentary Applications
Smalltalk/V BlackJack
Player Dealer
● instance
○ class

BJPlayer>>takeCard:
BlackJack>>hitMe
BJWindow>>hitMe
BJWindow(Object)>>perform:
MenuWindow>>performMenuItem:
BJWindow(ApplicationWindow)>>performMenuItem:
Message>>perform
NotificationManager>>empty
NotificationManager>>run

Player:
8♥
8♣

Dealer:
a Card
6♥

The BJPlayer adds the card passed as an argument in this message to its current hand. It also checks to see if the card leads to a blackjack or a bust. If so, it reports this to its dealer, the BlackJack instance.

Object
BJPlayer
Card
CardGame
BlackJack
Collection
IndexedCollection
OrderedCollection

dealer:
hand
initialize
takeCard:

```
takeCard: aCard

    "Adds one card, reporting a blackjack
    or bust if it happens."
```

Figure 2. The View Matcher.

offer guided exploration of a wider range of examples deliberately crafted to expose other parts of the hierarchy.

MiTTS is only an introduction to Smalltalk, and one of our current concerns is how we might apply the minimalist model to more advanced instruction. The materials are clearly design-oriented, but MiTTS "teaches" design only through analysis, not through generation. Students finishing MiTTS have little sense of how to decompose and solve a programming problem of their own (unless it is conveniently similar to an example they have already analyzed). MiTTS makes no pretense at teaching learners the foundations of the class hierarchy; while learners do encounter many basic Smalltalk classes (OrderedCollection, Dictionary, String, Array, etc.), they learn only those aspects that are needed by the example applications. Our hope is that by emphasizing the design of Smalltalk applications from the start, MiTTS will have provided a structure within which to organize and integrate the mountain of Smalltalk learning that lies ahead. ■

References

1. Gagne, R. M. and L. J. Briggs, PRINCIPLES OF INSTRUCTIONAL DESIGN. Holt, Rinehart and Winston, New York, 1979.
2. Carroll, J. M. THE NURNBERG FUNNEL: DESIGNING MINIMALIST INSTRUCTION FOR PRACTICAL COMPUTER SKILL. MIT Press, Cambridge, MA, 1990.

Mary Beth Rosson is a research staff member at IBM's T.J. Watson Research Center in Yorktown Heights, New York, where she has been since 1982. Her current research centers on the problems of learning and applying the concepts of object-oriented design. She has a B.A. in psychology from Trinity University and a Ph.D. in experimental psychology from the University of Texas. She is on the editorial board of Interacting with Computers, and is a member of ACM SIGCHI, the Human Factors Society, and the Society for Computers in Psychology.

■ **TAKING EXCEPTION TO SMALLTALK, PART 2** *coninued from page 5*

(using tempAt:number:put:) rather than reading from it. The definition of the address method used in sender and at: differs for PseudoContext and GroundedContext, with the former returning the value of its frameAddress instance variable and the latter returning the frameOffset of its underlying HomeContext.

Finally, we added the method thisContext to Object. Like sender, this feature is built into Smalltalk-80. Unlike the sender method, though, thisContext is supported as a pseudovariable (like self) in Smalltalk-80, but we implement it as a method for V/286:

```
thisContext
    ^[] homeContext sender
```

This completes the implementation of our exception-handling system. After adding this package to your V/286 system, you can introduce the use of signals to identify common or important errors, and so support dynamic responses to errors in future work. Besides this practical benefit, our addition of signal handling to V/286 is important as an illustration. Because processes and contexts are objects programmers can manipulate, we were able to extend the functionality of the low-level system to support our needs as application programmers; in particular, methods for Exception needed to reflect on the system's operation by knowing about and accessing V/286's representation of contexts and processes. Without that ability, we'd have been unable to make the necessary changes—only language implementers could make them by changing the language and the compiler themselves.

It's also interesting to compare our implementation with ParcPlace's system. While we've provided much of the same functionality, Smalltalk-80's exception handling has two advantages over ours. First, their system is more efficient because it is supported by the virtual machine instead of being implemented entirely in Smalltalk. They have several important optimizations to speed up expensive operations such as traversing the context stack looking for the next exception handler or unwind block. In addition, the reflective nature of our implementation is slower because we rely on more layers of message sends and abstractions—this problem will exist until reflective programming can be recognized and optimized by the compiler. Smalltalk-80 also behaves better because, as we noted last month, normal method returns are treated just like returns from exceptions, so unwind blocks will be executed if skipped over either in exception handling or in normal computation. We couldn't provide the same function because the semantics of method returns are hardwired into the V/286 virtual machine; we would have benefited from a more reflective implementation in which method returns could be modified by the programmer. In the future we hope to write other articles that highlight reflective aspects of Smalltalk and other practical benefits of objectifying system internals. ■

References

1. LaLonde, W., and M. Van Gulik. Building a backtracking facility in Smalltalk without kernel support. PROCEEDINGS OF OOPSLA '88, OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, pp. 105–122, November 1988. Printed as SIGPLAN Notices, Volume 23, Number 11.

Bob Hinkle and Ralph E. Johnson are affiliated with the University of Illinois at Urbana-Champaign. Mr. Hinkle's work is supported by a fellowship from the Fannie and John Hertz Foundation.

Kent Beck

An Objectworks\Smalltalk 4.1 Wrapper Idiom

One of the most significant changes to Smalltalk in recent years is the refactoring of display functionality into the VisualComponent hierarchy of Objectworks\Smalltalk release 4. I am only beginning to realize the full implications of factoring borders, composition, and layout into their own objects.

Some recurring problems arise in using the new architecture. This column addresses the problem of addressing the right object in an environment of changing compositions of objects. The solution was invented by Jay O'Connor, a bright new Smalltalk programmer I've been working with for the last year. My columns have been getting longer and longer, so I've have tried a different approach this time. I will address a specific problem in a specific context, rather than try to tackle something as general as "Collections: The Big Picture." Let me know what you think. My numbers are listed at the end of the article.

ARCHITECTURE

In the beginning there was View. (Well, not entirely the beginning, but back there someplace). And View was responsible for rendering a model's data on the screen. To make pretty pictures on the screen, though, View picked up a few more responsibilities along the way—drawing borders, composing subviews, transforming coordinates, and clipping display operations. All this responsibility and the state to support it made View difficult to subclass, expensive to instantiate, and hard to teach to new programmers.

Two things were needed: to split each of the responsibilities into its own object and to compose the objects so they could work together to achieve the same results as before. Objectworks\Smalltalk release 4 introduced the new architecture. It has three main families of objects: VisualComponents for displaying models, GraphicsContexts to translate and clip display operations, and Wrappers to modify the GraphicsContext on the way down to the VisualComponent.

VisualComponent

VisualComponent is an abstract superclass. It requires its subclasses to implement displayOn: aGraphicsContext, which renders the component. All displaying is done relative to 0@0, so the component need not know where it eventually ends up on the screen. Subclasses also have to implement preferredBounds, which returns a Rectangle describing the size the component thinks it should be. Layout may make its screen appearance smaller.

VisualPart

VisualPart adds an instance variable, container, to VisualComponent. A VisualPart is linked to its containing component so it can send itself invalidate when it wants to be redrawn. In the original display model, a View that got an update and wanted to display itself would do it right then. If several Views redisplayed this could result in an annoying flicker. With the invalidation model, all VisualParts wanting to display register their interest, and the next time a Controller goes through an idle loop it notices that redisplay needs to happen and it all occurs at once.

CompositePart

A CompositePart puts several VisualComponents together. Their placement and displayed size are subject to a layout object that can flexibly position components in either relative or absolute positions relative to the CompositePart. It is easy to specify locations like "the top third of the composite with a margin of 20 pixels at the top."

Wrapper

Wrappers pass most messages through to their component. Some messages get intercepted or modified on the way. A translating wrapper, for instance, would change where its component displayed. A color wrapper would change the colors used by its component. Wrappers are supposed to be transparently composable. That is, you should be able to insert new wrappers anywhere between the containing object and the component without interfering with the operation of the component or any of the other wrappers.

BoundedWrapper

One of the most common wrappers, BoundedWrapper translates and clips graphics operations. You'll almost never explicitly create a BoundedWrapper. Adding a component to a composite automatically inserts a BoundedWrapper whose size and location are set according to the specified layout.

BorderedWrapper

You might think that bordering and bounding would be handled in separate wrappers, in the spirit of purity and composability. Instead, apparently for implementation reasons, BorderedWrapper is a subclass of BoundedWrapper. BorderedWrappers compute their preferredBounds by increasing the size of their components' preferredBounds by the size of their border. They implement displayOn: by displaying the border, insetting the clipping bounds, and asking their component to display.

GraphicsContext

All graphics operations go through a GraphicsContext. It has symbolic protocol to display images, lines, rectangles, strings, and so on. It also carries along a clipping bounds and translation. Wrappers like BoundedWrapper work by modifying the GraphicsContext passed along to their component. It also carries along a foreground color and background color, so components that operate in two colors need have no knowledge of what colors to use, leaving it up to the GraphicsContext to have the right color set (perhaps by enclosing the component in a wrapper that sets the color).

The heavy reliance of this model on wrappers leads to a problem. You would like to treat the chain of wrappers and the component they enclose as a single unit. You would also like to insulate the wrappers and their component from changes in the wrapper chain.

You could write "container container remove: container" to remove a component from its composite. This assumes that the container is some form of wrapper and its container is a CompositePart. When I started using the new framework I wrote code like this often. I also found it breaking often, because I inevitably wanted to insert new wrappers in the chain. Every time I inserted a new wrapper I'd have to change the length of "container container ..." expressions to match the new setup.

A slightly more modular way to fix this problem is to implement a pass through message in Wrapper. The component could say "container remove: self" and Wrapper>>remove: aVisualComponent could pass the message on to its container. The problem with this solution is that it introduces many messages in Wrapper.

IMPLEMENTATION

What we needed was an abstract way to address a message somewhere in the wrapper chain. We wanted to treat the chain as a single object for most operations. First we needed to get the top wrapper in the chain. VisualComponents don't have a container, so they assume there are no wrappers.:

```
VisualComponent>>topWrapper
    "By default"
    ^self
```

VisualParts ask their container for the topWrapper. Note the nil check. Many methods in VisualPart would be simpler if VisualParts maintained the invariant that their container could not be nil. I have never found a case where it wasn't nil, except after a window has been released. The lack of an invariant leaves you with one more thing to remember—always check the container before sending it a message.

```
VisualPart>>topWrapper
    container isNil ifTrue: [^self].
    ^container topWrapperFor: self
```

The argument to topWrapperFor: will always be the component immediately below the receiver. When you get to an object that isn't a wrapper, you return the object below, which is guaranteed to be either a wrapper or the original component itself:

```
VisualComponent>>topWrapperFor: anObject
    ^anObject

Wrapper>>topWrapperFor: anObject
    container isNil ifTrue: [^self].
    ^container topWrapperFor: self
```

TopWrapper is useful all by itself. I often keep a collection of VisualComponents that I'm interested in managing. When I want to remove one from a CompositePart, I need—you guessed it—the topWrapper. I can say component topWrapper container remove: component topWrapper. Simple. It no longer matters what part of the chain I hold onto. I can get to the top and bottom easily.

Back to our effort to send messages "somewhere in the chain." We want to send a message to the first wrapper that implements it. The search proceeds from the top wrapper down, so we can insert new wrappers to intercept the message later. This creates the potential for other problems later on, like accidentally shadowing a method in a lower Wrapper, but I haven't found it to be a problem in practice.

```
VisualComponent>>wrapperSend: aSymbol
    ^self topWrapper wrapperDelegate: (Message selector: aSymbol)
```

Using this protocol, a component can say things like self wrapperSend: #disable, assuming that some wrapper in the chain implements disable, but not assuming where in the chain it resides.

Now we need to implement wrapperDelegate:. The default implementation is to perform the message:

```
VisualComponent>>wrapperDelegate: aMessage
    ^self perform: aMessage selector withArguments: aMessage
arguments
```

Wrappers need to be a bit smarter. They perform the message only if they can understand it; otherwise they pass it on to their component (remember, search for the appropriate wrapper proceeds top down).

```
Wrapper>>wrapperDelegate: aMessage
    ^(self respondsTo: aMessage selector)
        ifTrue: [super wrapperDelegate: aMessage]
        ifFalse: [component wrapperDelegate: aMessage]
```

Here is an interesting question: Should the VisualComponent send the message regardless, or ignore it if it isn't understood? I can see both sides. Ignoring it is better for modularity because if you don't have a wrapper that responds, the message disappears without a trace. On the other hand, if someone is

continued on page 17

Juanita Ewing

How to Create Smalltalk Scripts

In my last column I discussed a critical concept in Smalltalk application development: Never view your image as a permanent entity. I concentrated on extracting application source from an image and rebuilding the image by recreating classes and methods from source. However, with complex applications, classes, pools, and global variables also must be defined and initialized, usually with a script.

This column describes how to construct Smalltalk scripts and includes a discussion of some useful expressions and structuring mechanisms for scripts.

FORMAT

A Smalltalk script is really just Smalltalk code stored in a file, usually in file-out format.* This is the format with the !'s, in which most people are used to seeing class and method definitions. The most common way to create a file in this format is to use a file-out menu item in a browser.

A class definition consists of the definition string followed by a single exclamation point:

```
Point subclass: Object
    instanceVariableNames: 'x y'
    classVariablesNames: ''
    pools: ''!
```

Method definitions are more complicated. A header portion indicates the class to which methods belong. It begins and ends with exclamation points and is followed by one or more method definitions, each ending with a single exclamation point. After all method definitions, another exclamation point is appended, indicating that the series of method definitions is over:

```
! Point methods !
   x
      "Return the x component of the receiver"
      ^x !
   = aPoint
      "Return true if both the x and y coordinates areequal."
      ^self x = aPoint x and: [self y = aPoint y]! !
```

Most implementations require a separating character between the last two exclamation points. This is because exclamation points in the definition source are doubled when written to a file. If there is no separator between them, the system will interpret it as a single exclamation in the method source.

The header portion of method definitions varies from implementation to implementation. The example above uses a Smalltalk/V style header. In Smalltalk-80–derived implementations, the header also identifies a protocol:

```
!Point methodsFor: 'accessing'!
x
    "Return the x component of the receiver"
    ^x! !
```

Other interesting pieces of code that people want to put in scripts are really just do-its. It turns out that class definitions are also do-its, so we already know the proper format: code followed by a single exclamation point.

USEFUL SCRIPT EXPRESSIONS

I usually start my file-in scripts with a comment describing the contents of the script and any relevant assumptions. Filing in a comment has no effect on your image, but is a handy way to document the contents of a file. Just like any other do-it, an exclamation must follow the expression:

```
"This file contains the script to load the drawing application. This load
procedure has been tested with version1.4" !
```

Another common expression in a file is a class initialization. In this example the message initialize is sent to the DrawingApplication class. The appropriate initialization method will vary from class to class:

```
DrawingApplication initialize !
```

You may query the user for the location of relevant files before proceeding. Note the use of a temporary variable here:

```
| directory |
directory := Prompter prompt: 'Where is the archivedirectory?'.
directory isEmpty
    ifFalse: [(Disk file: directory, '\archive') fileIn]!
```

A dialog with the user might be appropriate during the file-in process, particularly if the expression is destructive. In this example, a global name is going to be removed from the system. Place interaction with the user at the beginning of the script to allow automated builds:

```
| confirm |
confirm := Prompter confirm: 'The next step is irreversible.Continue?'.
confirm ifTrue: [Smalltalk removeKey: #Vector] !
```

Some applications make use of global variables, which can be declared and initialized in scripts. Current ways to declare

* Described in Chapter 3 , TheSmalltalk-80 Code File Format, of Smalltalk-80 : Bits of History, Words ofAdvice, edited by Glen Krasner (Addison-Wesley, 1983).

global variables reveal some implementation details of the global name space in Smalltalk implementations. Global variables are stored as symbols in Smalltalk, which is a dictionary. Don't forget the # mark, which creates a symbol literal in the expression. In our example we create two global variables, the first with an initial value of nil. Nil is used as the initial value of variables in other places in the Smalltalk system:

```
Smalltalk at: #DrawingMode put: nil.
Smalltalk at: #DefaultColor put: ClrBlack !
```

It is also possible to test if a particular global name has been defined. This can be useful when combining segments of an application in a mix and match style. In the first expression the existence of the global name Vector is tested for; if it is not defined, then the file containing its definition is loaded. This type of expression is really ad hoc configuration management.

In the second expression the existence of Vector is tested for and a message displayed if the name is already defined. Since the user doesn't furnish any meaningful input, a better alternative is to write messages to the Transcript instead of putting dialogs in the middle of a script:

```
Smalltalk
    at: #Vector
    ifAbsent: [(Disk file: 'Vector.st') fileIn] !
(Smalltalk includesKey: #Vector)
    ifTrue: [MessageBox message: 'About to redefine Vector']!
```

Another type of global that needs to be declared is a pool. Current ways to declare pools also reveal some implementation details. Pools are dictionaries and keys are available in the methods of classes using the pool. Note that the declaration of the pool is a separate expression from the subsequent references to it. Each expression is independently compiled.The first expression is compiled and executed, which declares the pool if neccessary. We avoid redefining the pool if it already exists because that would orphan existing references to its variables. After the pool has been declared, subseqent do-its can reference it by name. The second expression defines three pool variables. This example is appropriate for Smalltalk/V systems. In Smalltalk-80-derived systems, the keys should be symbols:

```
Smalltalk
    at: #TypesettingConstants
    ifAbsent: [Smalltalk at: #TypesettingConstants put: Dictionary new]!
TypesettingConstants at: 'Bold' put: '.B'.
TypesettingConstants at: 'Italic' put: '.I'.
TypesettingConstants at: 'Underline' put:'.U' !
```

STRUCTURING SCRIPTS

Do-its in a workspace or file are executed, logged in the changes file, and never referenced again by the system. Typical Smalltalk source control mechanisms don't capture do-its; thus do-its are difficult to maintain. To overcome this problem in scripts, which typically have many do-its, developers should, whenever possible, turn do-its into methods. Methods are maintained by the Smalltalk system and can be browsed and filed-out. They don't disappear after execution. An expression to initialize a class variable, for example, can be turned into a class method.

Files are the basis of another structuring mechanism. Application source can be composed of multiple files based on functionality. Several files based on functionality are more reusable than a single large application file. It is easier to distribute and use a piece of functionality if it is separated from the rest of an application. Because extracting a unit of functionality from a large application source file is very challenging, interesting functionality will not be reused if it is not separated.

Even though application source is separated into multiple files, the application can be reconstructed quite easily. Scripts often load a series of files in a particular order. In this expression three files are loaded into an image:

```
(Disk file: 'enhancements.st') fileIn.
(Disk file: 'classes.st') fileIn.
(Disk file 'initialization.st') fileIn !
```

An alternative equivalent expression easier to extend is:

```
#(
'enhancements.st'
'classes.st'
'initialization.st')
    do:
        [:each | (Disk file: each)fileIn ] !
```

The final structuring mechanism to discuss is based on a class. In this mechanism, we devote an entire class to rebuilding an application. This class probably also has functionality to store the source for an application. Do-it expressions not related to a class should be incorporated into methods in the rebuilding class. For example, an expression that creates and initializes a global variable should become a method. Then all methods creating global variables should be called from a controlling method:

```
initializeGlobals
    "Define and initialize global variables."
    self initializeDrawingGlobal.
    self initializeLabelGlobal.
    self initializeDrawingLocationGlobal
```

Developers should create a similar set of methods for defining and initializing pools. The entire rebuilding class can now be maintained by the Smalltalk system, instead of equivalent code maintained by the developer in script files. The source for the rebuilding class also needs to be archived in the same manner as the source for the rest of the application.

CONCLUSION

The ability to declare globals and pools and to initialize classes in a noninteractive mode is important in rebuilding complex Smalltalk applications. Understanding the file-in format and having a few examples can go a long way toward creating effective scripts, but script code should be turned into methods and classes whenever possible. Be wary of complex scripts and initialization methods too difficult to debug and maintain. ■

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

Alan Knight

Designing a Data Structure Library

This month's column centers on issues involved in creating an O-O library of data structures, specifically those involving cursors. Because of the background required and the Eiffel-specific nature of much of USENET's discussion on this topic, I've avoided direct quotes and given a general introduction to the concepts.

CURSORS AND ITERATION

Cursors, in the context of data structures, are an abstraction of position in a collection. Just as a graphical cursor marks a place on the screen, these cursors mark a position in the data structure. As a trivial example, in the loop:

```
1 to: anArray size do: [:i |
   (anArray at: i) printOn: Transcript].
```

the integer i acts as a cursor; it marks a position in the array.

If the structure were a dictionary then its keys could serve the same purpose:

```
aDictionary keys do: [:eachKey |
   (aDictionary at: eachKey) printOn: Transcript].
```

The choice of cursor can be quite significant. The array example above works well, but the dictionary example wastes time doing a lookup for every key. A much more efficient mechanism (which Dictionary iteration methods use) is to operate directly on the underlying representation, which is an array:

```
1 to: aDictionary size do: [:indexIntoPrivateStorage |
   | association |
   association :=
   (aDictionary basicAt: indexIntoPrivateStorage) isNil
      ifFalse: [
         association value printOn: Transcript]].
```

This operates much more efficiently than the previous version, but has some disadvantages. It makes the code significantly more complicated, circumvents encapsulation to expose representation details, and makes the code totally dependent on those details. Overall, it is extremely bad code with the potential to ruin reputations and turn programmers doing routine maintenance into homicidal maniacs.

It is difficult to do efficient iteration with explicit cursors while minimizing bad code and the senseless loss of human life. The situation only gets worse if we consider structures like sets, which cannot be indexed at all using public methods. The solution? Make an abstraction of cursors. While this will have to be implemented differently for different collections, it should allow us to write code like:

```
| cursor |
cursor := aCollection cursorAtStart.
[cursor atEnd] whileFalse: [
 (aCollection atCursor: cursor) printOn: Transcript.
  cursor next].
```

Why would you ever want to do that?

I haven't yet explained why anyone would want explicit cursors. I'm sure many experienced Smalltalkers are shaking their heads and thinking What's wrong with do:? as they turn to the next article. They have a point. For many applications, the best method of iteration is the standard:

```
aCollection do: [:each |
   each printOn: Transcript].
```

Although do: is likely implemented using a cursor, that complexity is hidden. The code is shorter, clearer, and just as general. Why bother with explicit cursors? Unfortunately, there are circumstances in which do: and its siblings just aren't adequate. Sometimes the cursors are themselves meaningful:

```
1 to: aCollection size do: [:i |
   Transcript show: 'Item ', i printString, ' = '.
   (aCollection at: i) printOn: Transcript.
   Transcript cr].
```

Sometimes iteration does not break down naturally into processing single items. For example, on a collection of characters we might wish to operate on groups (e.g., words). We might like the decisions on how to group characters to be made by methods deep inside the processing loop. At the same time, we want to maintain a clean interface.

We can deal with this situation by making a new object that contains both the collection and a cursor. If all our methods deal with this object instead of the collection, then different methods can use or change the cursor position easily. In Smalltalk these objects are normally called *streams.*

STREAMS

Unfortunately, standard Smalltalk streams have a few deficiencies. The main problem is that they don't use a general mechanism for cursors. Instead, streams have an integer index that is used to record position. Collections (such as sets or dictionaries) that can't be indexed by integers can't be used in a stream.

I suspect there are those who would argue that this is a feature, that integer indices allow us to stream over any ordered collection, and that we weren't meant to stream over unordered collections. I can't agree with this. First of all, I see nothing wrong with streaming over unordered collections. Second, even if we restrict ourselves to collections with order, integer cursors are only really practical for array-based representations. Using an integer cursor for a large linked list or tree structure would be very inefficient.

MORE COMPLEX STRUCTURES

The limited implementation of streams is most likely a symptom of the lack of different data structures in the standard Smalltalk image. Because everything in the normal image is array-based, integer indices are fine.

In contrast, a good data structures library will have many different structures, each with variations and trade-offs. These structures will require more sophisticated cursor mechanisms; exploiting the structures fully will require us to use them.

For instance, a standard data structures technique is the use of cursors to exploit locality of reference. Consider a sorted collection in which items are to be located using binary search. Although there may be no fixed pattern to the searches that would let them be directly optimized, we know that consecutive searches often look for items that are close together. Using this information, we can store the position of the last object searched for and use that position as a starting point for the next search. Although there are no guarantees that any particular search will be faster using this technique, the average search time may be significantly improved. More complicated structures and search schemes might exploit several cursors for greater improvements. Cursors aren't the only way of exploiting these types of patterns. Another method is to reorganize the data structure itself for greater efficiency on particular queries. This is the basis of some list-organizing heuristics and of data structures like splay trees.

STORING CURSORS

The USENET discussion that started all this began by comparing two different Eiffel libraries: those from ISE and SIG Computer.

Paul Johnson (paj@gec-mrc.co.uk) writes:

The ISE library has the notion of a cursor within the object being traversed. Hence to traverse a list you put the cursor to the beginning of the list and then at every step you move it forwards...In the SIG library, the cursors (called iterators) are separate from the objects.

This raises an issue we haven't considered—where cursors should be stored. At first glance, the ISE idea of storing a cursor inside the object seems very strange. There's no major advantage over using a stream and it would make multiple simultaneous iterations very difficult.

I suspect that some of the reasons for doing this have to do with Eiffel's limitations. Although I'm not very familiar with Eiffel, I don't think it has blocks or an equivalent that would make operations like do: possible. Without these, all iterations would have to use an explicit cursor, and storing the cursor inside the collection would make basic iteration code the simplest.

Multiple simultaneous operations are handled by providing methods that save and restore the cursor state. This makes multiple iterations possible, but code must explicitly guard against that possibility, reducing generality.

This technique does have one significant advantage, however. That is in dealing with iteration over changing collections.

ITERATING OVER CHANGING COLLECTIONS

A standard Smalltalk error is to iterate over a collection and have the iteration block modify the collection. This can cause very strange effects and be difficult to track down.

While experienced Smalltalk programmers rarely do anything as obvious as:

```
aCollection do: [:each |
  aCollection remove: each].
```

there are more subtle variations that can catch the best of us. They've certainly caught me. One moderately subtle example is iterating over the collection of all active windows, closing them. Unfortunately, when you close a window, it gets removed from the collection of active windows. A cursor embedded in the collection itself could automatically be updated to compensate for these kind of changes.

A more general approach is to make a copy of a collection being streamed over or as soon as it is modified. This adds overhead and complication to the implementation, but should be invisible to the user and makes the semantics more consistent. Improved semantics alone may make the cost worthwhile.

Peter Deutsch (deutsch@smli.eng.sun.com) recommends this approach, for which he credits Xanadu's Smalltalk-to-C++ development environment:

The bookkeeping for doing copy-on-write is much less than the actual cost of the copy, so if there aren't any read/write collisions, you never make the copy; if there *is* a read/write collision, it may still be possible to copy incrementally. I, for one, would rather pay the cost to have well-defined semantics than implementation-dependent happenstance...
Taking a leaf from databases: reading should create a virtual copy, but writing should be exclusive.

To reduce copying expense, it is worth looking at the work on "persistent" data structures to reduce the amount of copying. This is a different use of the word *persistent* than is common in O-O circles. It refers to data structures that allow updates as well as access to all previous states without just copying at each step.

CONCLUSION

Writing truly general and reusable libraries requires a lot more thought and design than just assembling a few useful bits of code. Good abstractions require careful consideration of the needs of different domains as well as ways the system may need to be extended.

Alan Knight is a researcher in the Department of Mechanical and Aerospace Engineering at Carleton University, Ottawa, Canada, K2C 3P3. He can be reached at 613.788.2600 x5783, or by email at knight@mrco.carleton.ca.

PRODUCT ANNOUNCEMENTS

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

ParcPlace Systems has introduced VisualWorks, an application development environment for corporate developers who need to create graphical client/server applications that are instantly portable across PC, Macintosh, and UNIX platforms. The key components of VisualWorks include a graphical user interface (GUI) builder, database access capabilities, a reusable application framework, and instant cross-platform portability. VisualWorks is based on ParcPlace's ObjectWorks\Smalltalk, a mature, fully object-oriented programming environment. Using VisualWorks, MIS departments can quickly create graphical applications to fill increasing user demands for information access, as well as leverage an object-oriented architecture to address complex application development needs.

VisualWorks is ParcPlace's first product in the emerging ADE market. Currently, software vendors from various categories are entering this market by offering products that integrate interface builders, database access tools, and application logic for client/server applications. VisualWorks sets a new standard by providing the components of an ADE with the added advantages of instant cross-platform portability, and object-oriented foundation and an application framework.

ParcPlace Systems, 999 E. Arques Ave., Sunnyvale, CA 94086, 408.481.9090, fax: 408.481.9095

ObjecTime Ltd. has released an upgrade to its object-oriented CASE tool for distributed, event-driven systems. **ObjecTime 4.0** supports executable specification and design models for real-time interworking via TCP/IP sockets. For example, various hardware or software entities can be controlled directly from an executing ObjecTime model. ObjecTime supports an advanced methodology for the analysis and design of distributed, event-driven systems known as real-time object-oriented modeling (ROOM). ROOM includes graphical design concepts and a highly interactive development process that help to eliminate error-prone discontinuities between the various phases of software development. As a unique feature, ObjecTime enables the creation of executable analysis and design models that can be tested in an extensive workstation-based runtime environment.

The high-level, object-oriented concepts are independent of programming language. Either C++ or ObjecTime's Rapid Prototyping Language (based on Smalltalk-80) may be used at the detailed design level. To broaden the appeal of ObjecTime's ability to intermix graphical design content with detailed level programming, version 4.0 contains improved C++ support. The package supports both the GNU compiler and AT&T's cfront.

ObjecTime Ltd., 340 March Rd., Ste. 200, Kanata, Ont., Canada K2K 2E4, toll-free 800.567.TIME, 613.591.3400, fax: 613.591.3784 sales@objectime.on.ca

Digitalk Inc. is shipping the 32-bit version of its object-oriented Smalltalk/V development environment for OS/2 2.0. The new version results in Smalltalk/V applications that are up to 100% faster and 50% smaller than 16-bit OS/2 applications.

The 32-bit package offers many improvements over its 16-bit predecessor, including the ability to call both 16-bit and 32-bit Dynamic Link Libraries, a debugger with enhanced single-stepping capability, improved support for bitmaps, double-byte character set characters in Smalltalk/V code, and support for OS/2's common dialog boxes.

Digitalk Inc., 9841 Airport Blvd., Los Angeles, CA 90045, 310.645.1082, fax: 310.645.1306

Servio Corp., developer of the GemStone object database management system (ODBMS), and **Object Technology International (OTI) Inc.**, developer of the ENVY/Developer software engineering environment, have announced a cooperative relationship to support Smalltalk applications' development and delivery. The integration of ENVY/Developer and GemStone, through the GemStone Smalltalk Interface (GSI), provides developers with a complete client/server-based application development and delivery environment for building robust Smalltalk applications.

Through the cooperative partnership, Smalltalk applications developed using ENVY/Developer will have access to all of GemStone's Smalltalk object database support, including integrated garbage collection of persistent Smalltalk objects and support of cooperative client/server applications. ENVY/Developer is used to coordinate team development, version control, and configuration management. GemStone, in turn, is used to store and retrieve Smalltalk objects, providing high performance and active server functionality in support of distributed Smalltalk applications.

Servio Corp., 950 Marina Village Pkwy., Ste. 110, Alameda, CA 94501, 510.814.6200, fax: 510.814.6227

RECRUITMENT

IT'S TIME
to become a
CONSULTANT

Numerous Immediate Southern California Consulting Opportunities for
OOP PROFESSIONALS

-- Smalltalk
-- ENFIN
-- C++; OS/2 (Heavy Experience)
-- Other Significant OOP Exp.

Software Management Consultants, Inc.
500 No. Brand Blvd., Suite 1090
Glendale, CA 91203

Voice (818) 240-3177
Fax: (818) 240-7189

We are a rapidly growing consulting company with many state of the art openings.

✦

LONG TERM ASSIGNMENTS
HIGHEST COMPENSATION

SMALLTALK 80

RMS

COMPUTER CORPORATION

1212 Avenue of the Americas, New York, NY 10036, 9th Floor
(212) 840-8666 • (800) 843-9119 • Fax (212) 768-7188

SMALLTALK IDIOMS *continued from page 11*

counting on the return value, the "doesNotUnderstand" case will cause an error. Besides, I like having a notifier pop up during development if I send a message no one understands. It's usually because I have made a mistake.

WrapperSend: has made my life much easier. I now use wrappers as they were intended: independent bits of functionality that can be composed in different ways with abandon. I no longer pause to think if any of my code depends on the configuration of the wrapper chain.

Another useful implication is that in using the wrapper pass through mechanism described above I had to implement my messages in three places: VisualComponent for some default behavior, Wrapper for delegation, and wrapper subclass for implementing the method for the real behavior. With wrapperSend:, I only put it in one place: the class that really implements it. All the other wrappers pass it along automatically.

CONCLUSION

Wrappers make amazingly powerful and flexible interface objects. Coloring, highlighting, visibility, selection, and double buffering are some of the activities that used to be built into interface objects that can now be factored into their own wrapper. Once you have a library of wrappers you can create new interface objects by composition, never having to create new classes. The possibilities are mind boggling and not nearly fully explored. If you find new wrappers, send them to me and I'll put them in this column. Call me at 408.338.4649 or fax me at 408.338.1115. ■

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.

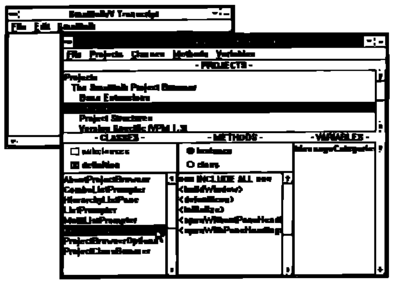
The Smalltalk Project Browser

Source Code Management System for Smalltalk/V

Version 2.00

Empower Software is pleased to announce the release of The Smalltalk Project Browser, version 2.00. The Smalltalk Project Browser allows developers to track and manage changes to Smalltalk/V systems, and to develop and analyze code more effectively. Version 2.00 extends this toolset to include the following:

Project Browser - project source code management
Project Class Browser - source code editing & tracking
Source File Browser - external source code analysis
Class Interface Browser - class/protocol editing
Navigator - enhanced Inspector for inter-object traversals
Program Shell - simple Smalltalk/EXE program launcher



Available now for $125 ($25 for registered users): price includes support for both VWIN (1.0-2.0) and VPM/VOS2 (1.3-2.0), full source, and user manual. Add $5 shipping in U.S.A. or Canada, $15 elsewhere. CA residents add 8.25% sales tax. Call for quantity or dealer discounts, or for more information. Corporate POs welcome.

Empower Software
279 S. Beverly Drive, Suite #217
Beverly Hills, Ca. 90212
Voice: (213) 878-2327 CIS: 71031,2640

■ EDITORS' CORNER *...continued from page 2*

Research Center reports on the use of MiTTS (Minimalist Tutorial and Tools for Smalltalk). MiTTS is a research project at IBM that aims to foster an approach to learning Smalltalk based on building an initial understanding of object-oriented design. The work of Mary Beth and her colleagues is important for all of us who help programmers climb the Smalltalk mountain. As Smalltalk continues to be widely adopted by the MIS community, training organizations are being presented with the challenge of teaching Smalltalk to programmers whose previous experience has been largely in COBOL or 4GL.

Also in this issue, Kent Beck describes the Wrapper idiom introduced in release 4.1 of Objectworks\Smalltalk 4.1 and describes how, with a library of wrappers, new interface objects can be created by composition rather than by creating new classes. In this month's Getting Real column, Juanita Ewing described the pros and cons of using scripts (code stored in file-in format) to perform tasks such as declaring globals and pool variables and class initialization. Finally, Alan Knight, with help from USENET contributors, looks at the issues involved in designing a data structure library.

Best wishes to all our readers for 1993.

VOSS

Virtual Object Storage System for

Smalltalk/V

Seamless persistent object management for all Smalltalk/V applications

- Transparent access to all kinds of Smalltalk objects on disk.
- Transaction commit/rollback of changes to virtual objects.
- Access to individual elements of virtual collections for ODBMS up to 4 billion objects per virtual space; objects cached for speed.
- Multi-key and multi-value virtual dictionaries for query-building by key range selection and set intersection.
- Works directly with third party user interface & SQL classes etc.
- Class Restructure Editor for renaming classes and adding or removing instance variables allows applications to evolve.
- Shared access to named virtual object spaces on disk; object portability between images. Virtual objects are fully functional.
- Source code supplied.

Some comments we have received about VOSS:

"...clean ...elegant. Works like a charm."
–Hal Hildebrand, Anamet Laboratories

"Works absolutely beautifully; excellent performance and applicability." *–Raul Duran, Microgenics Instruments*

Logic ARTS

VOSS/Windows $1950, VOSS/286 $1450, VOSS/OS2 in development. Quantity discounts from 30% for two or more copies. (Ask for details) Visa, MasterCard and EuroCard accepted. Please add $15 for shipping. Logic Arts Ltd 75 Hemingford Road, Cambridge, England, CB1 3BY TEL: +44 223 212392 FAX: +44 223 245171

Highlights

Excerpts from industry publications

SPECIFICALLY SMALLTALK

It has been ten years since I implemented my first object-oriented language: our first version of Smalltalk. And so, it is interesting at this point to reflect on the man advances and changes that have taken place over the last ten years and speculate on what the future holds for object-oriented technology. I think there is a unifying theme in all of this and that it relates to making people productive, to allowing programmers and analysts to do a better job and meet the changing requirements of their companies and their own personal situations.

Object insider: George Bosworth, OBJECT MAGAZINE, 11-12/92

STANDARDS

. . . We may be looking forward to the electronic equivalent of the Tower of Babel if everyone insists on doing things their own way—trying to lock up all of the market with mutually exclusive approaches. I have a colleague who says that the need for standards is a middle-age disease. Standards are unquestionably dull, but they are precisely what make telephones and fax machines so useful (and widely used). We need to apply some of the same logic to the next round of operating environments.

Industry watch: What do Microsoft, IBM and Apple have in common?, Richard Dalton, WINDOWS, 8/92

RECRUITMENT

TO PLACE A RECRUITMENT AD,
CONTACT HELEN NEWLING AT
212.274.0640

Object-Oriented Systems Professional

CHALLENGE US

make a difference

For more than a century, Eli Lilly and Company has been dedicated to improving the quality of life throughout the world. Today, we are a $5+ billion global research-based corporation that develops, manufactures, and markets pharmaceuticals, medical devices and diagnostic products, and animal health products.

Central to our success is our dedication to the quality of our products and the quality of our employees. We carefully select applicants and provide for their development through ongoing training programs, a promotion-from-within policy, and opportunities to improve the skills needed to grow in their careers.

We currently seek a systems professional with object-oriented project experience to work in our corporate technology group, with responsibility for developing an object-oriented technology strategy for the company and defining the roles needed in the OO environment. The successful candidate will help us integrate OOT into our software development process, promote the appropriate application of OOT, help to demystify OOT, and facilitate communication among OOT projects.

A BS or MS degree and minimum 3 years experience in OO design and programming are required. Strong communication, interpersonal and problem-solving skills with the proven ability to handle multiple projects in a team environment also needed.

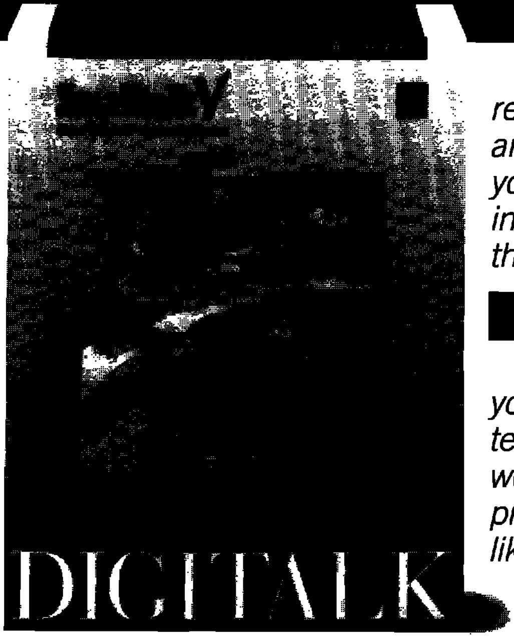
Eli Lilly and Company offers competitive compensation, extensive development programs, and a superb work environment. We will reimburse reasonable moving expenses for those who join us at our corporate headquarters in Indianapolis, Indiana. Indianapolis is an active growing city offering an excellent quality of life and an affordable cost of living, as well as broad-based cultural, recreational, and sports activities.

For prompt consideration, please send a cover letter and resume to: Eli Lilly and Company, Corporate Recruitment - Systems Dept. 001S, Lilly Corporate Center, Indianapolis, IN 46285.

We are an equal opportunity employer committed to diversity in the workplace.

Lilly

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



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

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

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

ONE-STOP SHOPPING.

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

Which you can benefit from on-site, or at our training facilities in Oregon. Either way, you'll learn from a staff that literally wrote the book on object-oriented design (the internationally respected "Designing Object Oriented Software").

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

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

IMMEDIATE RESULTS.

Digitalk's training gives you practical information and techniques you can put to work immediately on your project. Just ask our clients like IBM, Bank of America, Progressive Insurance, Puget Power & Light, U.S. Sprint, plus many others. And Digitalk is one of only eight companies in IBM's International Alliance for AD/Cycle—IBM's software development strategy for the 1990's. For a full description and schedule of classes, call (800) 888-6892 x410.

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

100% PURE OBJECT TRAINING. DIGITALK