The Smalltalk Report

The International Newsletter for Smalltalk Programmers

November–December 1993 Volume 3 Number 3

THE HP DISTRIBUTED SMALLTALK IDL LANGUAGE BINDING

by Jeff Eastman

Contents:

The Object Management Group's Common Object Request Broker Architecture (CORBA) specifies the architecture for an Object Request Broker (ORB), which provides a standard communication mechanism between object systems in a distributed environment. An ORB's job is to communicate with ORBs on other systems, to locate the objects that can perform requested services, and to communicate requests that can be processed by those remote objects. A critical part of the ORB's job is the translation (via a language binding) between its language-neutral Interface Definition Language (IDL) and the local language (such as Smalltalk, C, or C++). The translation process uses the IDL definition to convert client object requests expressed in the client's language into request packets that can be decoded by the server object and for converting result packets produced by the server into the appropriate local language entities.

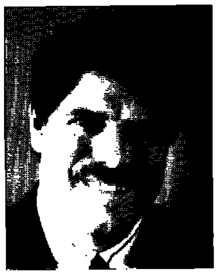
In order to use the ORB, it is necessary for programmers to know how to access ORB functionality from their particular programming language. Hewlett-Packard recently introduced HP Distributed Smalltalk, a full CORBA implementation for the Smalltalk-80 language as delivered by ParcPlace Systems of Sunnyvale, CA. This article describes the manner in which the constructs of CORBA are made available to Smalltalk programmers in HP Distributed Smalltalk.

INTERFACE DEFINITION LANGUAGE

To allow objects that are implemented in different programming languages to interoperate via the ORB, it is necessary to define their behavior in an abstract manner and then for each implementation to provide a mapping from this abstract description to its particular language. The HP Distributed Smalltalk binding from IDL to the Smalltalk-80 programming language provides the Smalltalk programmer with mechanisms for expressing the following IDL concepts:

- References to objects defined in IDL
- Invocations of operations, including passing parameters and receiving results
- Exceptions, including what happens when an operation raises an exception and how the exception parameters are accessed
- IDL basic datatypes
- IDL constructed datatypes
- References to constants defined in IDL
- Access to attributes
- Signatures for the operations defined by the ORB, such as the dynamic invocation interface, the object adapters, etc.

continued on page 21...


John Pugh

Paul White

EDITORS' CORNER

Although distributed computing and object-oriented systems have already been proven important technologies for the 90s, it is their union in the form of distributed object systems that has even greater potential to address many of the problems facing application developers in the next few years. Distributed computing addresses the need for more open, scalable, reconfigurable systems—systems whose elements may reside on different processors at various network locations, but which function as an integrated whole. Object-oriented technology addresses the need for systems which provide for better reuse of software components, more robust and extensible software, and which more accurately model the application domain.

In this month's lead article, Jeff Eastman, the architect of Hewlett-Packard's Distributed Smalltalk product and the designer of its Interface Definition language binding, gives us a first look at distributed computing with Smalltalk. HP Distributed Smalltalk is a set of approximately 150 classes built on top of ParcPlace Systems' Visualworks product, which provides the first complete implementation of the Object Management Group's Common Object Request Broker Architecture (CORBA), a standard specification for how objects make requests and receive responses in a distributed environment. In HP Distributed Smalltalk, messages may be sent to objects without regard for whether the intended receiver is a local or remote object. To the Smalltalk programmer, access to remote objects is completely transparent.

Another new product is also causing quite a stir in Smalltalk circles. SmalltalkAgents from Quasar Knowledge Systems is a new implementation of Smalltalk for the Macintosh. It has generated a lot of discussion both for its departure from traditional Smalltalk syntax and semantics with extensions patterned after C and Lisp, and for its exciting and innovative features. In the former category we find that all objects are equal to true, except 0, nil, and false, which are all equal to each other. In the latter, we find scoped, nestable public name spaces, support for finalization, and true multitasking. In the first of two reviews entitled "Shoot-out at the Mac Corral," Jan Steinman takes an in-depth look at SmalltalkAgents. In our next issue, Jan will review the new version of Smalltalk/V for the Mac from Digitalk.

Also in this issue, Susan Griffin introduces What If?, a protocol for object validation. Susan argues that while some domain objects can live a productive life without a specification for validation, those with complex rules that are at the mercy of a user can benefit from standardizing their validation techniques. Working with clients on large Smalltalk systems, we find that many of them find the notion of use cases (à la Jacobson) a powerful modeling tool. In this edition of her Putting It in Perspective column, Rebecca Wirfs-Brock describes her experiences bridging system requirements, object design, and user interface design through the application of use cases. In this issue's GUI column, Ray Horn describes how to extend the popular WindowBuilder product from ObjectShare by adding a custom pane and its associated editing dialog. Finally, Kent Beck looks at the contentious issue of whether Smalltalk needs a case statement. We vote no!

John Pugh [illegible]

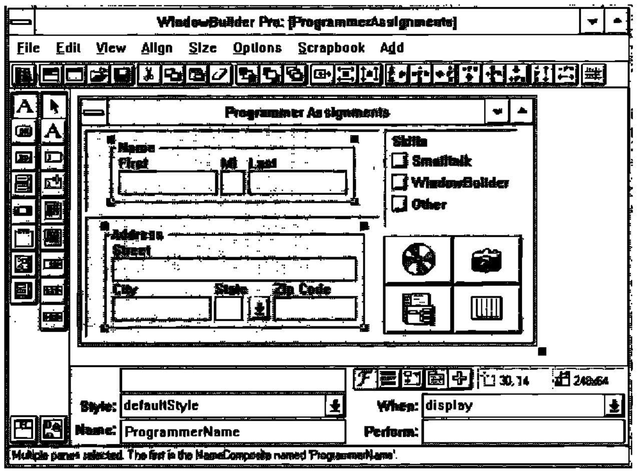
THE SMALLTALK REPORT (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by SIGS Publications Inc., 588 Broadway, New York, NY 10012 212.274.0640. © Copyright 1993 by SIGS Publications. All rights reserved. Reproduction of 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, $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. **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.

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.

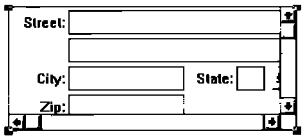


Multiple panes selected. The first is the NameComposite named 'ProgrammerName'.

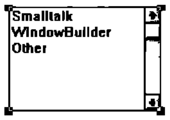
WindowBuilder Pro/V is available on Windows for $295 and OS/2 for $495. Our standard WindowBuilder/V is still available on Windows for $149.95 and OS/2 for $295. We offer full value trade-in for our WindowBuilder customers wanting to move up to Pro. These products are also available in ENVY®/*Developer* 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.

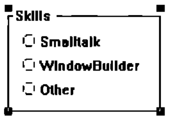
Some of the exciting new features...

- 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.

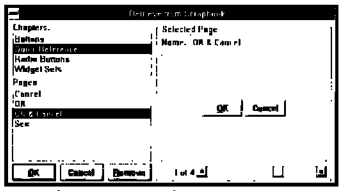


- Morphing: Allows the developer to quickly change from one type of control to another, allowing for powerful "what-if" style visual development. The flexibility allowed by morphing will greatly enhance productivity.

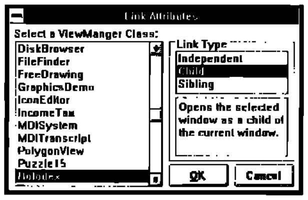




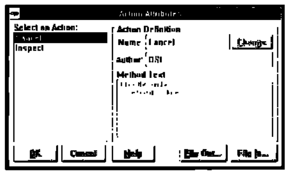
- ScrapBook: Another new feature to leverage visual component reuse, ScrapBooks provide a mechanism for developers to quickly store and retrieve predefined sets of components. The ScrapBook is a catalog of one's favorite interface components, organized into chapters and pages.



- 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

WHAT IF? A PROTOCOL FOR OBJECT VALIDATION

Susan Griffin

Encapsulation mandates that an object maintain the integrity of its own state. After all, nobody else will. There's nothing more dangerous to a domain object than a system that allows a user (an often inconsistent object!) to manipulate its state through some kind of interface. Unfortunately for the object, that manipulation is usually the purpose for its existence. In many "exception-based" domains, such as scheduling or configuration by users, validating requests for a change in state comprises the bulk of the services provided by domain objects.

The situation is complicated by the fact that business rules are often varied and rarely absolute. A secretary maintaining his manager's calendar may need permission to schedule meetings that overlap. A client may ask a sales representative to add components to a complex equipment order, as long as the ship date is not affected or the budget exceeded.

The presence of exception handlers (see Bob Hinkle and Ralph Johnson's series in THE SMALLTALK REPORT, Vol. 2, No. 3 & 4) certainly improves the situation for developers. Interfaces that allow users to manipulate domain objects can simply attempt to make a change the user requested rather than testing for validity before the attempt is made. I often picture this communication as a dialog between the manipulator (some interface) and the object (an AppointmentBook in this example):

an Interface:	*"Here, add this meeting. Don't bother me unless there's a problem."*
an AppointmentBook:	*"Hey, there's a problem!"* or *"Hey, there may be a problem!"* (an exception is raised.)
an Interface:	*"Uh-oh. I better tell the user that he'll have five minutes to get from the home office to the client site! (I wonder if it's OK?)"*

This certainly simplifies matters for the interface. However, many steps may be required of the domain object before it discovers that there is a problem. The validation process can become difficult to develop and maintain if a coherent strategy for performing validation is not employed. The remainder of this article discusses a way to organize the work that gets done between the request for a change and the resulting success or exception. It also provides several examples of the options a domain object has in raising an exception.

WHAT IF?

In several projects, I have successfully used a protocol within my domain objects that I like to call the *what-if technique*. It allows me to isolate my error validation code so I can concentrate on the business rules themselves and the severity of any violations, rather than when and where I will perform the validation.

Let's go back to our interface/domain object dialog. We'll adopt the perspective of the AppointmentBook. Someone has just asked us to add a meeting to ourselves. We need to interrogate this meeting object and some others before we allow the meeting to be added. A rapid fire of questions will result:

"Hey, meeting, what is your start and end time? Do I have a conflict? What meetings are scheduled before and after this one? Where are you located? Oh yeah? Hey, location, how far are you from the location of the meeting before this one? How about the one after you?..."

Certainly, these questions need to be asked. The question is, where's the best place to ask them? Even with extensive testing protocols in our Meeting and Location objects, a first attempt might still yield a rather messy #add: method. (All examples use exception handling facilities in Objectworks\Smalltalk 4.1.)

```
add: aMeeting
  "Add a meeting to my list of meetings. Raise an exception if any rules
  are violated. Notify my dependents if I add the meeting."
  self meetings do: [:eachMeeting |
        (eachMeeting conflictsWith: aMeeting)
              ifTrue: [ self class invalidModelSignal
                    raiseRequestErrorString: self meetingConflictString ]].
  (((self meetingBefore: aMeeting) notTooFarFrom: aMeeting)
        and: [ (self meetingAfter: aMeeting) notTooFarFrom: aMeeting])
              ifFalse: [ self class invalidModelSignal
                    raiseRequestErrorString: self tooFarAwayString ].
  self meetings add: aMeeting.
  self changed
```

Our instincts immediately tell us to separate the tests into separate methods, yielding a cleaner #add:.

```
add: aMeeting
  "Add a meeting to my list of meetings. Test to determine if rules are
  violated.
  Notify my dependents if I add the meeting."
  self checkForConflictsWith: aMeeting.
  self checkTravelTime: aMeeting.
  "exceptions will be raised and handled. If I get to the next line,
  everything's OK."
  self meetings add: aMeeting.
  self changed.
```

This certainly looks better, but barriers still loom ahead. Suppose the users are allowed to schedule meetings with inadequate travel time, as long as the system warns them and lets them decide? No problem; we raise proceedable exceptions and still add the meeting if the user (via the interface and corresponding exception handler) decides to proceed. But the writing is on the wall. The users will want some sort of visual feedback if there is inadequate travel time, which means our AppointmentBook will need to include testing protocol. We end up creating testing methods that look a lot like our validation methods.

```
isInadequateTravelTime
  "Answer true if there are any meetings with inadequate travel time
  between them."
  self meetings do: [:eachMeeting | |nextMeeting|
      ((nextMeeting := self meetingAfter: eachMeeting) notNil and:
          [eachMeeting tooFarFrom: nextMeeting])
              ifTrue: [^true]].
  ^false
```

What's happened here? Previous tests for travel time involved checking the proposed meeting, aMeeting, against my list. Our new testing method requires no argument, since its purpose is to test the current state of the object. The validations need to be available for both the current state of the object and the proposed state of the object. That's what what-if is all about. If we can create an object that looks like the one the manipulator wants ("What if I added this meeting?"), we can use the same testing method to ask it if it is valid. And we can define a structured protocol that simplifies both the methods that perform the validation and those that invoke them. For example, I'd prefer not to change my #add: method every time a new rule is introduced about adding meetings, since we know how often these kinds of requirements can change.

TURNING TECHNIQUE INTO PROTOCOL

Let's see what the *what-if* approach is all about by restructuring the code. The #add: method becomes very simple.

```
add: aMeeting
  "Add a meeting to my list of meetings. Notify my dependents if I add
the meeting."
  self proposedChange: #add: with: aMeeting.
  "exceptions will be raised and handled. If I get to the next line,
everything's OK."
  self meetings add: aMeeting.
  self changed.
```

This is looking better. Any future changes to the #add: method will be only due to changes in the semantics of adding a meeting, not because of new validation rules. Our simple method is no longer overwhelmed by error checking code. The new addition is the method #proposedChange:. The AppointmentBook is proposing to add a particular meeting to itself. If the proposal fails, exceptions will be raised. The following methods in AppointmentBook illustrate the rest of the protocol.

```
proposedChange: anAspect with: anArgument
  "Private - What if anAspect of me was changed? Would it be a problem?
  I expect exceptions to be raised during validation if there is a problem."
  self validating
      ifTrue: [ (self validatingCopy perform: anAspect with:
aParameter) validate: #newMeeting]

validating
  "Private - Answer whether I am currently validating all changes to me."
  ^validating

validating: aBoolean
  "Private"
  validating := aBoolean

validatingCopy
  "Private - Answer a copy of myself which accepts any proposed
  changes until asked to validate. The validating copy
  must remember who originated the copy so that any resultant
  exceptions will carry the appropriate parameters."
  ^self copy
      original: self;
      validating: false

validate: aValidationCategory
  "Perform all validations associated with aValidationCategory."
  (self validationTests at: aValidationCategory)
      do: [:eachTest | self perform: eachTest]

validationTests
  "Private - Answer a dictionary of validation tests. The key is a symbol
  representing the category of the change.
  The value is a list of selectors for each test that should be performed.
  This really should be cached in a class variable."
  ^IdentityDictionary new
      at: #newMeeting put: #(checkTravelTime checkConflicts);
      at: #removeMeeting put:
          #(checkRequiredMeetings notifyAttendees);
      yourself.
```

So what happened? When the AppointmentBook proposed to make a change to itself, it actually created a what-if copy. The copy was a standard copy with a pointer back to the original object and a validation flag turned off. Then, the original object's "proposed change" (adding the meeting) was actually made to the copy. After the meeting was added, the original AppointmentBook asked the new copy to validate itself.

To avoid blanket error checking each time a change is made, a category of validations was specified. Each category can be associated with an array of validation methods appropriate to the situation. In this case, the methods #checkTravelTime and #checkConflicts were the relevant tests for adding a meeting. The final step was to perform the tests.

The protocol support methods above need only be coded once. We can now focus on the validation and testing methods, which are structured differently in light of what if.

```
checkTravelTime
  "Check for meetings with inadequate travel time and raise an
  exception if any are found."
  self meetings do: [:eachMeeting | | nextMeeting |
      ((nextMeeting := self meetingAfter: eachMeeting) notNil and:
          [eachMeeting tooFarFrom: nextMeeting])
              ifTrue: [ self class invalidModelSignal
                          raiseRequestWith: self original
                          errorString: self tooFarAwayString ]].

isInadequateTravelTime
  "Answer true if there are any meetings with inadequate travel time
  between them.
  If exceptions are raised during the check, I know there is inadequate
  travel time, and I
  can answer true in the handler."
  self class invalidModelSignal
```

```
        handle: [:ex | ^true]
        do: [self checkTravelTime] .
    ^false
```

Now we have one method, #checkTravelTime, which contains the logic to check for adequate travel time between meetings. The accompanying testing method #isInadequateTravelTime can simply be defined in terms of the validation. This is especially useful, since the presence of testing methods for every validation allows user interfaces to give early feedback to the user or perhaps disallow an invalid change.

It is important to note that the validations themselves are performed by the new what-if copy, not the original object. Any exception handlers for our signal must be aware that the originator of the exception is the validating copy. The parameter of the exception is the actual object we attempted to change. We could just as easily implement a new exception raising method, #raiseFrom:with:errorString:, that would allow us to declare the original object as the originator and the copy as the parameter. In either case, it is important to document the originator and parameters of any resultant exception. It's a good ideato provide the exception handler with both the original state and the *what-if* state for maximum flexibility handling the problem.

WHY BOTHER WITH WHAT IF?

Was it worth it? We may have had some ugly code to start with, but look at all the code we had to write for what if. For an object with few validation tests, this may be overkill. However, if your domain is heavy on rules for state validation, you have a lot to gain. My current implementation abstracts most of the protocol in a superclass, ValidatingModel. My subclasses simply define the categories of exceptions and the logic for the tests themselves. And as always, the more a process is structured, the more it can be automated. I can build simple code generators that allow me to define the test categories and validation logic and generate appropriate setters and test methods at the push of a button. Guess what? I'm suddenly concentrating on the rules and how to express them, rather than where to put them.

CAVEATS

With every convenience comes a price. If I generalize the validations, they will ultimately test more conditions than necessary. For example, when I add a meeting, I should only have to check the travel time for the previous and next meeting. Generalizing the test means I iterate through all the meetings, including those that may have been tested when they were added previously. Making the validating copy certainly costs more space, even if temporarily. In domains with complex and changing rules, these drawbacks have been acceptable for me, but keep them in mind during performance profiling. Optimization may be necessary if a particular validation is expensive in terms of time or space.

EXTENSIONS

Possible extensions to this technique are numerous. Rich protocol can be added to the ValidatingModel to include block changes (anAppointmentBook validateAfter: [some changes]), delayed validation (anAppointmentBook validationOff), and categories of initialization. I am currently exploring techniques for employing a what-if approach without using inheritance as the framework for reuse.

Code generators could become a substantial project, providing many options for initialization, categorization, and specification of the validation logic. Rules could be specified with a rules language rather than Smalltalk code. Once a rules definition language is defined, the rules could be specified dynamically using input scripts.

CONCLUSION

While some domain objects can live a productive life without a specification for validation, those with complex rules that are at the mercy of a user can benefit from standardizing their validation techniques. Any technique that can help separate the error tests from the semantics of the "normal" course of action produces code that is easier to read and maintain. It also helps to separate two very distinct tasks—maintenance of an object's services and maintenance of its rules. ■

Susan Griffin is an independent Smalltalk developer assisting clients in the design and development of systems in Smalltalk/V and VisualWorks. Her interests include discovering reusable designs and improving the usability of systems. Prior to founding Griffin & Griffin, she was a Smalltalk instructor at Knowledge Systems Corporation. Further discussions of this topic or others are welcomed via phone at 919.676.2294 or email at 72147.2656@compuserve.com.

Kent Beck

It's just not the case

The topic of this month's column is case statements: practical necessity or pernicious contaminant? My interest in the topic comes from several areas at once. SmalltalkAgents has added a form of case statement to their Smalltalk for the Macintosh. CompuServe has hosted a lively discussion of isKindOf: and its relatives. Finally, net news has had a discussion of case statements. What's the deal?

Cutting right to the punch line, I think case statements are an inappropriate holdover from procedural thinking. While vital in procedural languages, their use in object programs is obviated by the much more powerful mechanism of the polymorphic message send. Anytime you find yourself wishing for or using a case statement, you have an opportunity to take advantage of objects instead. The non-case version will yield a more maintainable, more flexible, more readable, and faster solution.

Of course, I can't just say case statements are bad, I have to demonstrate how to avoid or eliminate them. Here is the first of two patterns that go a long way toward getting rid of the need for case statements.

PATTERN: TURN CLASS TESTS INTO MESSAGES

Context

To get code running, you occasionally have to insert an explicit test for the class of an object, either through sending it the message class or isKindOf:, or by introducing a testing method like isInteger, which is implemented in Integer to return true and in Object to return false.

Problem

Class tests, explicit or implicit, are a maintenance nightmare. An operation like refactoring an inheritance hierarchy can break seemingly unrelated code. How can you eliminate class testing?

Constraints

- ***Limited impact.*** You'd like the solution to affect as little code as possible.
- ***Readability.*** The solution should reveal more of the programmer's intent than the original code.
- ***Maintainability.*** The solution should yield code that is less susceptible to breaking because of unrelated changes than the original.

I did a little research into the various images' use of class tests. Table 1 provides the raw results. These numbers need a little interpretation. There are legitimate uses for isKindOf:, like writing generic comparison methods. There are also legitimate uses of class. It is used heavily in V Mac 2.0 to return instance-invariant information.

The most interesting comparison in Table 1 is between V Win 2.0 and V Mac 2.0. Both images come from a common base and share a lot of code. The Mac image shows the effects of being worked on after Digitalk bought Instantiations, which brought a new sense of discipline to Digitalk's code. Both the reduction in the reliance on isKindOf: and in the increase in the use of class, not for class testing, but for instance-invariant behavior seem to be the result of the strict programming style developed in Portland.

Solution

Replace the test with a message. Implement the conditionally executed code as the method in the class tested for. Implement the conditionally executed code as the method in the class tested for. Implement an empty method (or one that returns a default answer) in all the other classes the object could be.

Example

Here is an example from the V Win 2.0 image. The method

Table 1. How various images use class tests.

	V Win 2.0	V Mac 2.0	VisualWorks 1.0	ENVY for VisualWorks
Senders of isKindOf:	44	26	161	214
Senders of isMemberOf:	3	1	26	26
Senders of class	156	810	573	823
is... methods in Object	43	18	11	13

“ I think case statements are an inappropriate holdover from procedural thinking. ”

ApplicationWindow>>isTextModified returns true if any of the pane's children has modified text. It looks like this:

```
ApplicationWindow>>isTextModified
    children
        detect: [:each | (each isKindOf: TextPane)
            and : [each modified]]
        ifNone: [^false]
    ^turn
```

This method will break if you add new text editing panes that don't inherit from TextPane. Using the transformation described above, we implement two methods:

```
TextPane>>isTextModified
    ^self modified "This is the conditionally executed code"

Pane>>isTextModified
    ^false
```

Then the original method simplifies to:

```
ApplicationWindow>>isTextModified
    children
        detect: [:each | each isTextModified]
        ifNone: [^false]
    ^true
```

This transformation has done two things. First, the code is easier to read. I can read it as saying, "I have modified text if any of my children have modified text." No such simple statement can be made about the original. Second, the intent of my code is much clearer. If I want to create a subclass of Pane that edits text, it is clear from browsing the code in Pane that I will have to override isTextModified. Before, whatever behavior depends on checking for modified text (like prompting before closing a window), would simply not have worked, and you would have a chore figuring out why.

OTHER PATTERNS

You may be able to factor the implementations of the blank methods higher in the hierarchy (Move Common Methods Up).

What if you have an object that can be in one of three states, and you have to take the state into account in several methods? Seems like a natural use of a case statement, doesn't it? In my next column, I'll present the Multiplexer pattern, which improves your design in such a situation at the same time it eliminates the need for a case statement ■

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.

Rebecca Wirfs-Brock

Designing scenarios: making the case for a use case framework

Experienced object designers explore the design space from many different angles. They refine ideas of how their systems should respond while they are in the middle of building and discarding ideas about how their designs should work. Getting a design to gel involves making assumptions, seeing how they play out, changing one's mind or perspective slightly, and reiterating. Design is a difficult, involved task. It inherently is a nonlinear process. Yet, we are asked to trace our design results back to system requirements. And if we uncover some implications during design, we'd like to tune our system requirements to reflect necessary design compromises.

To meet these challenges, we need solid conceptual bridges to help us straddle the concerns of what the system must do (analysis) and how it will be accomplished (design). We also need techniques for adding detail and driving out different perspectives during this process. In this column, I'll describe experiences we have had bridging system requirements, object design and user interface design by applying use cases.

WHAT IS A USE CASE?

Use cases, scenarios, or *scripts* are roughly synonymous terms forimportant ways to focus our design activities. I prefer the term *use case* (although quickly saying it three times can leave your tongue tied) because it emphasizes usage.

A use case is a textual description of a sequence of interactions between an actor (roughly corresponding to an external agent or class of users) and the system we are designing. Use cases were first described by Ivar Jacobson in OBJECT-ORIENTED SOFTWARE ENGINEERING: A USE CASE-DRIVEN APPROACH.[1]

Use cases have been around in various forms for quite some time. Jacobson, however, made the keen observation that use cases can be treated as refineable, extensible, and even reusable specifications of system requirements. We've had these same goals for object designs. We know that it is harder to actually accomplish them than it is to talk about them.

Use cases are a pretty powerful modeling concept, once we know how to effectively build them. What sounds good in theory needs to be practically applied within a basic system development framework. A flock of questions come to mind:

- What process can you use to build good ones?
- How should they be captured?
- How detailed should they be? Are there different levels of detail?
- When are you done finding and describing them?

I've had heated discussions about these exact same issues for object design. It isn't surprising that these themes keep recurring. People who build and describe software systems want to know how much they should describe before they truly understand what they are building. The answer to this question depends on how one intends to apply that descriptive information.

Many people claim to be using use cases. It's a trendy concept. Yet they all seem to be applying good use case construction techniques at completely different levels of detail! This can be incredibly confusing to an innocent bystander, manager, student of design technique, or end-user!

Being the pragmatic type, I really want to get to the heart of the matter. I've known for a long time that you really need to be aware of what perspective you are taking during a discussion.

I was stumped by the question of what's a good use case? until I read about the what vs. how dilemma in Alan Davis' excellent book on software requirements.[2] Davis discusses

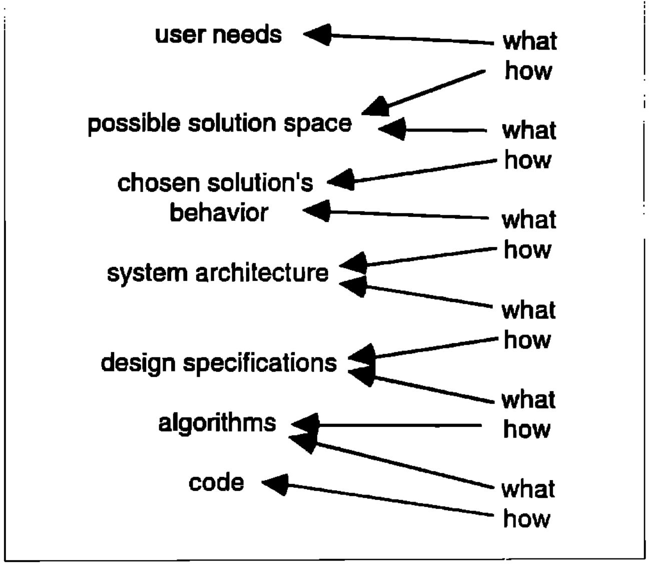


Figure 1. Requirements framework.

the requirements analysis dilemma. Claiming that requirements are a statement of what not how is extremely simplistic (and insufficient for us to know how to pick the level we want to be working at). Many prominent requirements techniques are really modeling different requirements! Davis presents a nice framework for discussing various methods (Figure 1).

Each item in this figure can be said to rightfully be a requirement. From one perspective, each item can be considered a what (a reasonable thing to include in a statement of requirements), or a how (something beyond scope). Depending on your viewpoint, you can argue for or against it. This is fun reading. It's good food for thought the next time you find yourself debating with your manager or colleague about whether some design activity is appropriate.

The key to solving our use case dilemma and keeping our sanity is to realize that one person's what is always another's how. Just as one size of requirements doesn't fit all, one canonical use case format won't work either. We need to formulate use cases slightly differently if we want to apply them to different purposes. Yet if we are careful not to get too arcane, these use case descriptions can be understood by a wide range of people.

What we really need is a conceptual map for describing and refining use cases, similar to the one Davis proposes for thinking about requirements. Once we have this framework, I am perfectly content to initially state, "It depends," and try to ascertain real needs before coming up with a reasonable answer to the question of what makes a good use case?

One answer doesn't have to fit all! New object designers need descriptions broken down into fairly detailed steps. Experienced designers are comfortable leaving out details that are easy to infer from the rest of the design. Those focused on designing user interactions need additional information. Teams transitioning formal system requirements on a large project into object designs undoubtedly need lots of precision.

Here's my proposal for a use case framework (Figure 2). This framework could plug into Davis' picture after the system architecture and present an augmented view of his design

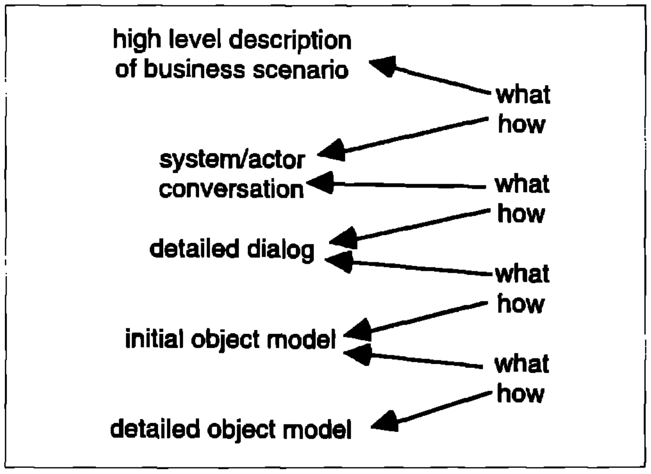


Figure 2. A use case framework.

specification models. We have experimented with several of these forms (with varying degrees of expressiveness and formalism) on several projects and in numerous mentoring and teaching situations.

A FIRST ATTEMPT AT DESCRIBING A USE CASE

Let's take a quick stab at writing a very high-level scenario for our hypothetical automated teller machine (ATM). We'll revisit and tune this use case to suit various needs.

Use Case: Performing an ATM Financial Transaction

A bank customer can select a financial service from several available transactions. These transactions include cash withdrawal or deposit, account balance inquiries, and funds transfer between two accounts. Once a customer has selected a financial service, she will be prompted to enter information necessary for performing the financial transaction. Upon completing a transaction,the bank customer may perform additional transactions or indicate that she wishes to terminate the ATM session.

This description is so non-specific that we could present any design (a human teller might satisfy these requirements or a fairly ridiculous design that has users entering their bank account numbers or cash amounts in Morse code) and argue that it met the requirements. Design students and developers need more guidance.

For large systems, there will be a wealth of additional requirements (ranging from user interface guidelines to detailed business function descriptions to banking regulations to process specifications, and on and on). All this information still needs to be distilled into a comprehensible form in order to commence design. We can always refer to the wealth of supporting requirements material, we just don't want to be overwhelmed.

In less formal design efforts, we need to supply more information. In either case, let's see how we might add more detail to this nearly content-free description.

OUR FIRST REFINEMENT

We have found it useful to clearly demarcate actor actions from system responses. This allows us to add more or less detail to either side of the conversation, as you'll see shortly. There are two central parts to this system/actor conversational form:

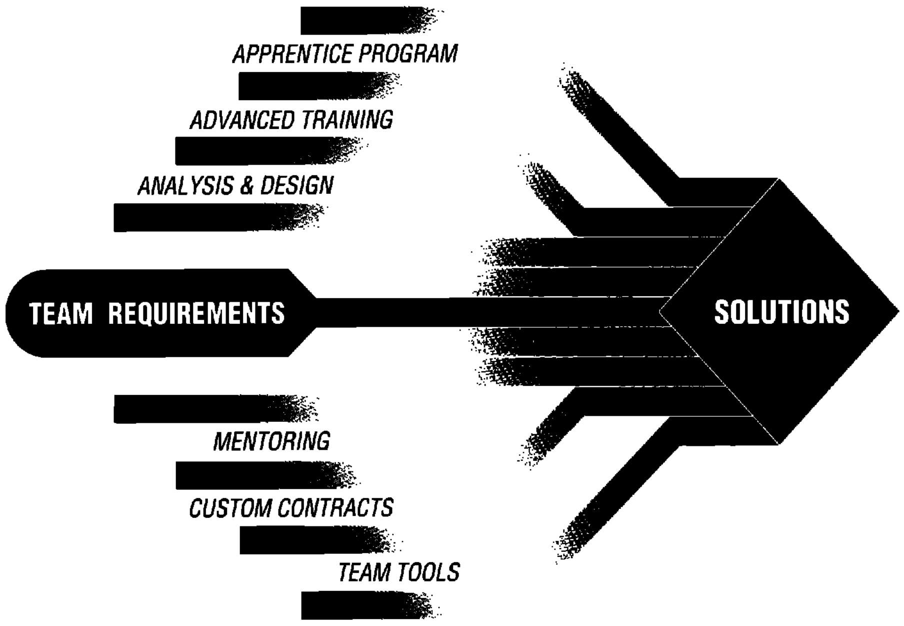
1. A description of the actors inputs to our system
2. A corresponding description of our system's responses

Together, these side-by-side narratives capture a dialog between an actor and our system. There is also a list of alternatives to the main course of the use case. These alternatives represent a reasonably complete list of conditions that system designers must be able to detect and to design appropriate responses for.

Use Case: Performing a Withdrawal Transaction

Actor: Bank customer
Overview: Bank customers can perform any number of finan-

Object Transition by Design



Object Technology Potential

Object Technology can provide a company with significant benefits:

- Quality Software
- Rapid Development
- Reusable Code
- Model Business Rules

But the transition is a process that must be designed for success.

Transition Solution

Since 1985, Knowledge Systems Corporation (KSC) has helped hundreds of companies such as AMS, First Union, Hewlett-Packard, IBM, Northern Telecom, Southern California Edison and Texas Instruments to successfully transition to Object Technology.

KSC Transition Services

KSC offers a complete training curriculum and expert consulting services. Our multi-step program is designed to allow a client to ultimately attain self-sufficiency and produce deliverable solutions. KSC accelerates group learning and development. The learning curve is measured in weeks rather than months. The process includes:

- Introductory to Advanced Programming in Smalltalk
- STAP™ (Smalltalk Apprentice Program) Project Focus at KSC
- OO Analysis and Design
- Mentoring: Process Support

KSC Development Environment

KSC provides an integrated application development environment consisting of "Best of Breed" third party tools and KSC value-added software. Together KSC tools and services empower development teams to build object-oriented applications for a client-server environment.

Design your Transition

Begin *your* successful "Object Transition by Design". For more information on KSC's products and services, call us at 919-481-4000 today . Ask for a FREE copy of KSC's informative management report: *Software Assets by Design.*

Knowledge Systems Corporation

OBJECT TRANSITION BY DESIGN

114 MacKenan Dr.
Cary, NC 27511
(919) 481-4000

Table 1. Typical user-ATM interaction.

Actor Action	System Response
User indicates she wants to perform a cash withdrawal	Present the user with a list of accounts
Selects a particular account	Prompt the user for cash amount
Indicates cash amount	Validate available funds on hand User cash amount must be in multiples of available denominations Update account balance Withdraw request must be within daily ATM limits and cash in account Log transaction on external record and prepare receipt information Dispense cash and sense when user has removed it from dispenser
User retrieves cash	Ask user if another transaction is desired
User indicates she is finished	Print out receipt and eject it Eject user's card

Alternatives:

1. Insufficient funds on hand.
2. Insufficient funds in customer account.
3. User doesn't respond to any part of thedialog (within a sufficient time period).
4. User wishes to perform an additional financial transaction.

cial transactions once they've presented the system with an unexpired, not-known-to-be-stolen bank card, and entered a valid personal identification number. The typical customer performs a single transaction before terminating a session with the ATM (see Table 1).

Use cases should be constructed by business domain–knowledgeable people. The key to building a good use case is to remember that it serves two purposes: It will guide developers and be reviewed with clients. Use cases should be written for both audiences. Actor interactions and system responses need to be described at a fairly high level.

On the other hand, descriptions of system responses must contain sufficient detail so that object designers working with analysts can design a reasonably detailed object model. Walking this fine line between sufficient detail and bogging down in details requires practice and critique.

Level of Detail

The sample use case we wrote still needs more detailed descriptions before we can design our system. In particular, the system responses need to be expanded upon to include:

- Steps and logical sequencing of actions that must be performed by the system.
- A description of necessary information that must be supplied by the actor. Reasonable defaults (if any) for information not supplied.
- Description of any data validation or business constraints that must be checked before performing a system action.
- Format of reports that are generated.
- Timing of and contents of any significant system feedback.

This could be captured in other documents, or less formally, much of this information might be directly placed in our object model. It needn't be crammed into either side of the conversation. Side notes and additional constraints make it hard to follow the thread of conversation.

We wrote our system/actor conversation for a concrete situation, withdrawing cash. We could have written it more abstractly, where the system response and actor conversations would describe any of the permissible transactions. If we wrote at this more abstract level, we'd have to remove a fair amount of detail from both the actor and system dialogs. For example, since not all transactions involve cash amounts specified by the user, we'd have to state that the user is prompted for additional information, if required, and that the bank customer enters information. We would also have to remove alternatives that don't apply.

This is too abstract for my tastes. This feels uncannily like the process I go through when I refactor responsibilities during detailed design. I do this only after I have responsibilities assigned to concrete classes. When I am carving up the systems responsibilities and finding more general ways of stating things, I also take pains to assign details to concrete classes. I'm not losing class-specific behavior during this refactoring.

I find it useful to keep use case conversations pretty specific for another important reason: They are understood by analysts, users, and developers. We can periodically review and verify correctness with clients. They will be refined over time to reflect actual implementation and to include more detail, particularly on the system-response part.

TUNING THE CONVERSATION

Conversations can also be worked on in order to tune the user interface of our application even before building a prototype. The most likely thread through the conversation is termed the main course. Other optional paths are alternatives. On a num-

continued on page 20...

Ray Horn

WindowBuilder: A do-it-yourself extension

WindowBuilder, originally developed by Cooper & Peters and now distributed by ObjectShare Systems, is a powerful GUI builder. WindowBuilder allows you to lay out user interfaces graphically, save them as classes with source code, and then edit these same interface classes graphically.

Cooper & Peters created WindowBuilder to be easily extended. This column explains how to extend WindowBuilder by adding a custom pane and its associated editing dialog. The editing dialog appears when you press the Other button in WindowBuilder.

A SAMPLE EXTENSION

The custom pane you will create is a list box with a horizontal scroll bar in addition to the vertical scroll bar that list boxes normally have. In Windows a list box gets a horizontal scroll bar when its horizontal extent is set to a number of pixels greater than its width.

You might reasonably expect the extent to be adjusted automatically so that horizontal scroll bars appear and disappear as needed, the way they do in MS Windows program groups. Unfortunately, this does not happen. So in this example the width must be set explicitly once. You may want to extend this example to automatically adjust the extent as list items are added or removed.

Step 1: Create the Custom Pane

Create HorizontalScrollListBox as a subclass of ListBox. HorizontalScrollListBox inherits the following method from ListBox for setting its horizontal width:

```
setHorizontalExtent: pixelWidth
    "Sets the width in pixels by which a list box can be
    scrolled horizontally. If the size of the list box is
    smaller than this value, the horizontal scroll bar will
    scroll items in the list box. If the list box is as large
    larger than this value, the horizontal scroll bar is
    disabled."
    self isHandleOk
    ifTrue:[
        UserLibrary
            sendMessage: self handle
            msg: LbSethorizontalextent
            wparam: pixelWidth
            lparam: 0
    ]
    ifFalse:[ self propertyAt: #horizontalExtent put: pixelWidth].
```

Override this inherited method so that the value of horizontalExtent is always at hand in the image:

```
setHorizontalExtent: pixelWidth
    "Make sure the value is always stored in the properties dictionary."
    self propertyAt: #horizontalExtent put: pixelWidth.
    super setHorizontalExtent: pixelWidth.
```

Now it is safe to write a simple method to get the horizontal extent:

```
getHorizontalExtent
    "Answer the width in pixels by which a list box can be scrolled
horizontally"
    ^self propertyAt: #horizontalExtent
```

Step 2: Create the Matching Interface Object

All subpanes edited in WindowBuilder have a matching interface object. InterfaceObject is a class-specific to WindowBuilder. Interface objects know how to draw themselves on a LayoutPane (another WindowBuilder class) and how to produce source code for the pane they represent. The interface object for ListBox is PListBox, a subclass of InterfaceObject. If some distant subclass of Subpane does not have an interface object named after it, then the interface object for the superclass is used. If you were to use HorizontalScrollListBox now, WindowBuilder would use PListBox to represent it and generate code.

To edit horizontal scroll list boxes differently from list boxes, you need to create the class PHorizontalScrollListBox, as follows:

```
PListBox subclass: #PHorizontalScrollListBox
    instanceVariableNames: 'horizontalExtent'
    classVariableNames: ''
    poolDictionaries: ''
```

The instance variable horizontalExtent is the horizontal scroll bar's width. Horizontal scrolling is done in units of pixels, rather than characters.

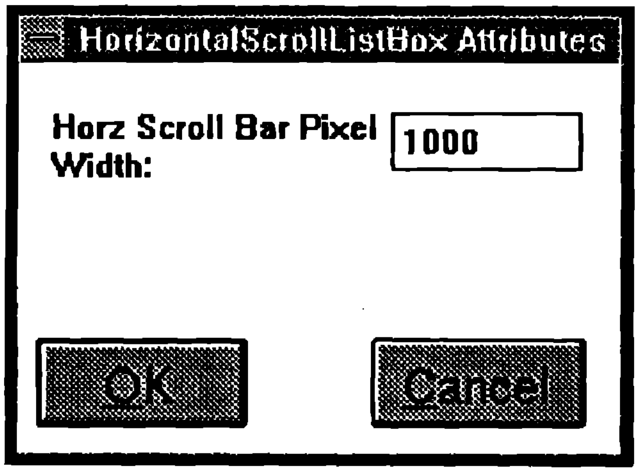
PHorizontalScrollListBox needs accessing methods called horizontalExtent and horizontalExtent:, a getter and setter.

```
horizontalExtent
    "Answers the horizontalExtent"

        ^horizontalExtent.

horizontalExtent: anInteger
    "Sets the horizontalExtent to anInteger"

    horizontalExtent := anInteger.
```


HorizontalScrollListBox Attributes

Horz Scroll Bar Pixel Width: 1000

Figure 1. Window displayed by the WBHorizontalScrollListBoxEditor.

WindowBuilder provides a number of standard methods that must be overwirtten by our custom InterfaceObject:.

- The attributeEditor method is called when the Other button is pressed.

```
attributeEditor
    "Answer a dialog for editing unique attributes."

    ^WBHorizontalScrollListBoxEditor new
```

- The copySpecificsTo: method makes a copy of the parameters specific to PHorizontalScrollListBox when the WBHorizontalScrollListBoxEditor OK button is pressed:

```
copySpecificsTo: aPane
    "Copy all of my attributes to aPane."
    aPane
        horizontalExtent: self horizontalExtent.
```

- The displayWith: method redraws PHorizontalScrollListBox when necessary.

```
displayWith: aPen
    "Redraw, with or without scroll bars, as appropriate."
    (self horizontalExtent > 0)
        ifTrue: [ self displayWithBothScrollBars: aPen]
        ifFalse: [ self displayWithLeftScrollBar: aPen].
```

- The initialize method initializes PHorizontalScrollListBox and sets horizontalExtent to 0.

```
initialize
    "Initialize a new instance of this object."
    super initialize.

    horizontalExtent := 0
```

- The readSpecificsFrom: method reads the specific parameters from the real HorizontalScrollListBox.

```
readSpecificsFrom: aPane
    "Read the specific settings from aPane."

    self horizontalExtent: aPane getHorizontalExtent.
```

- The storeSpecificsOn: indentString: method writes source code for the HorizontalScrollListBox during a save operation.

```
storeSpecificsOn: aStream indentString: indentString
    "Store the specific settings on aStream."

    (self horizontalExtent = 0)
        ifFalse: [
            aStream
        nextPutAll: ';'; cr;
        nextPutAll: indentString,'setHorizontalExtent: ', self
horizontalExtent asString.
        ].
```

It is not widely known that WindowBuilder maintains a dummy invisible instance of TopPane during an editing session. WindowBuilder uses this invisible TopPane to obtain the various parameters that were originally set in the source code.

WindowBuilder uses the InterfaceObject method storeOn: indentString:. to write source code for each control being edited. WBRealLayoutPanes send storeOn: indentString: to InterfaceObjects from generateCode.

Step 3: Create the Other Editor

The WBHorizontalScrollListBoxEditor is responsible for displaying

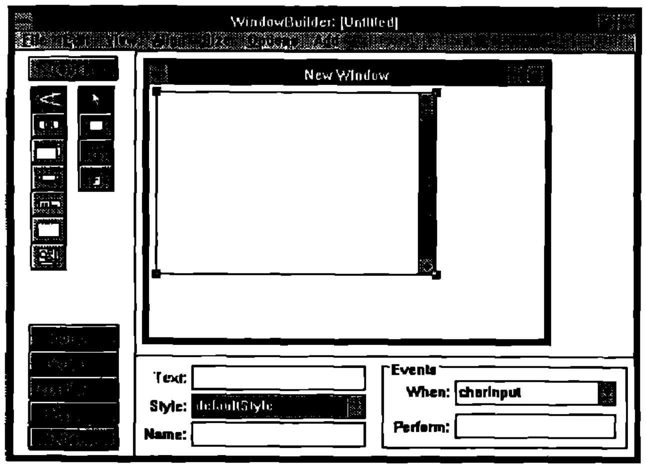


Figure 2. Layout pane with a normal HorizontalScrollListBox or ListBox on it.

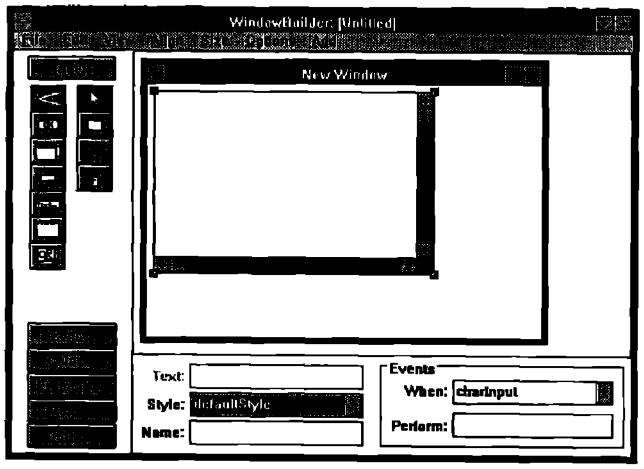


Figure 3. Layout pane with a horizontal scroll bar defined by using the Other pushbutton editor.

the window shown in Figure 1. The user enters the width of the horizontal scroll bar as an integer pixel value as shown in the figure.

To edit the HorizontalScrollListBox's parameters we must create the WBHorizontalScrollListBoxEditor.

```
changedHorzScrollWidthEF: aPane
    "This method is called for every keystroke the user enters to validate
the entry."
    | aValue |

    aValue := aPane contents.
    ((aValue notNil) and: [aValue asInteger > 0])
        ifTrue: [ (self paneNamed: 'horzScrollBarWidthST') enable]
        ifFalse: [ (self paneNamed: 'horzScrollBarWidthST') disable].

initWindow
    "The window is initialized with the opening values."
    | horzExtent |

    (self paneNamed: 'horzScrollBarWidthEF')
        contents: (horzExtent := thePane horizontalExtent) asString.

    (horzExtent > 0)
        ifTrue: [ (self paneNamed: 'horzScrollBarWidthST') enable]
        ifFalse: [ (self paneNamed: 'horzScrollBarWidthST') disable].

ok: ignore
    "The Ok button has been pressed so save the changed parameters
and close   the dialog."

    thePane
        horizontalExtent: (self paneNamed: 'horzScrollBarWidthEF')
contents asInteger.

    self close
```

Step 4: Tying it all together

The final step is to test the HorizontalScrollListBox custom pane. Figure 2 shows a layout pane with a normal HorizontalScrollListBox or ListBox on it. The layout pane contains one HorizontalScrollListBox object. HorizontalScrollListBox can be added to a layout pane by using the Add Custom Pane. . . menu. Refer to page 19 of the WindowBuilder reference manual for more details concerning how to manipulate custom window panes.

Figure 3 shows a layout pane with a HorizontalScrollListBox that has a horizontal scroll bar defined by using the Other pushbutton editor.

CONCLUSION

WindowBuilder can be extended rather readily. The result is an easier to use, more powerful interface builder. The HorizontalScrollListBox sample in this column is quite simple, but the same techniques can be used for a more elaborate extension of the WindowBuilder environment. ■

Ray Horn is an independent consultant with Hierarchical Applications Limited (HAL) in Cary, NC. He has over three years of extensive experience with object-oriented software design and development in Smalltalk/V WIN/PM/VOS2, Smalltalk-80 R4.1 and Envy Developer R1.41. Ray may be contacted via email at HBNH98A@Prodigy.com or through the American Information Exchange (AMIX).

Jan Steinman

Shoot-out at the Mac corral

One of life's little ironies may be drawing to a close. It is argued that Smalltalk's greatest mark on the world has been via its direct influence on Apple and the highly successful Macintosh line of computers. This Mac-Smalltalk connection has forever changed even the greater world of computing, due to its subsequent influence on the X Window System and Microsoft Windows.

Yet the Macintosh has never really had a credible Smalltalk implementation. An implementation of Smalltalk-80v1 was developed at Apple in the early Mac days, but its performance was inadequate for anything but O-O research. Digitalk produced a useful Mac implementation, but let it languish while they lavished resources on their popular MS-DOS products. In keeping with an otherwise admirable policy of "portability at all costs," ParcPlace's Mac implementations had difficult access to native platform features and were perceived as painfully slow on all but the top-end Macs.

Thus the machine that owed the most to Smalltalk paid it back the least. Mac fans who were also Smalltalk fans suffered the benign neglect of a second-string product; Smalltalk fans who were also Mac fans worked on Suns and PC-compatibles by day and went home to their Smalltalk-less Macs at night.

This situation is about to change with a brand-new, original Mac implementation of Smalltalk, and a major update to a venerable existing product. This article explores an exciting, though vexing, newcomer to the Smalltalk world, SmalltalkAgents, from Quasar Knowledge Systems. In a future article, we'll take a look at Digitalk's Smalltalk/V for Macintosh, version 2.0.

ENTER SmalltalkAgents

Now Mac fans have an alternative to the ParcPlace and Digitalk offerings in SmalltalkAgents from Quasar Knowledge Systems. It follows the Digitalk tradition of making a clean break with its Smalltalk-80 heritage—the class hierarchy and user interface are much more different from Smalltalk-80 and Smalltalk/V than they are from each other. (For convenience, we'll refer to the three dialects as ST-80, ST/V, and STA.)

STA departs from traditional Smalltalk in many ways, most of which are good ideas. For instance, the global name space has been segmented, so that class names can be repeated in different contexts, or libraries. There are a few not-so-good ideas, however, such as a sprinkling of C semantics that leaks through, ostensibly for performance reasons. For now, at least, I'll list all such *differences* as *features*, and will discuss the implications of some of these later.

FEATURES

Following is a quick list of features claimed by STA. At the time this article was written, QKS was shipping version 1.0.1 at a discount. It was missing some features slated for the "final" release, including a useful manual. The features not available at press time are noted below. Some of these features may be available in the currently shipping product.

- Tight, seamless integration with the Macintosh OS; direct access to most Macintosh services, including XCMDs and XFCNs, Quickdraw, access to Mac resources, machine and OS interrupt handling, and class-level interfaces to advanced Mac concepts, such as tear-off menus.
- Scoped, nestable public namespace, rather than global namespace, "Smalltalk" (instance of SystemDictionary) is replaced with multiple instances of Library. The "Environment" library has global scope; the others are accessed via their containment relationship to Environment. Any class can declare any library as a pool, thereby accessing that library's name space. One class can appear in multiple libraries, under different names, which can be useful for class name aliases, and portability between Smalltalks with different class names.
- A unified object model, without concern for the differences between named or indexed storage, or the size of storage elements. Object state can be named or indexed, and is dynamically sized. All indexable objects understand streaming protocol.
- Any object can contain "structured storage," which is a block of memory that is interpreted independently of the object, for example, a Mac resource or a C struct. Accessing such storage is direct; it does not carry the overhead of copying.
- All objects are capable of functioning as Strings. Strings hold 8-, 16-, or 24-bit elements in any combination.
- Objects have property lists, which indicate such things as whether the objects is immutable, or if it is fully resident in memory. (All objects have the basic capability to be known by proxy.)

Table 1. A comparison of Smalltalk implementations available for the Macintosh.

	STA 1.0.1		STV 1.2		ST/V 2.0		ENVY/VisualWorks	
start-up time	42%	20	15%	7.3	71%	34	100%	48
image save time	100%	18	18%	3.3	94%	17	88%	16
slopstone (no FPU)	100%	0.076	71%	0.054	60%	0.046	13%	0.037
slopstone (FPU)	74%	0.17	30%	0.070	27%	0.061	100%	0.23
smopstone (no FPU)	100%	0.10	46%	0.046	34%	0.034	39%	0.039
smopstone (FPU)	59%	0.13	27%	0.060	22%	0.050	100%	0.22
required memory	85%	3,500	35%	1,465	65%	2,654	100%	4,096
preferred memory (K)	60%	6,000	20%	1,953	36%	3,584	100%	10003
image size (K)	71%	2944	16%	669	50%	2073	100%	4121
number of classes	45%	369	20%	165	68%	553	100%	811
number of methods	34%	5892	20%	3490	59%	10190	100%	17268

- Objects receive events such as finalization prior to garbage collection.
- Source code is stored as styled text, and names may contain 16- or 24-bit characters. Syntax is added for literal styled text.
- Blocks can take a variable number of arguments, and be reflexive via the pseudo-variable blockSelf. Syntax is added to explicitly declare local or global scoping of block arguments, which allows much better performance if block locals can be used. This is also an improvement over ParcPlace's "clean/copying/full" hidden block semantics, since only sophisticated developers fully understand how to write a block for maximum perfyormance.
- Class Switch supports case statements.
- Local variables are automatically declared—they need not be declared at the top of the method, as in other Smalltalks.
- Call-by-reference is supported: formal parameters can be modified, and the modification is reflected in the actual parameter in the sending context.
- C-style syntax for Integer bit operations; C-semantics for true and false. All objects are equal to true, except 0, nil, and false, which are all equal to each other.
- Arbitrary-precision primitives. This could be extremely useful in financial software, since Float should not be used for counting money!
- Syntax added to support compiler directives. This has potential for improving performance by providing "type hints" to the compiler.
- Pre-emptive, time-sliced (UNIX-style) processes.
- General-purpose catch-throw exception handling mechanism, similar to the C setjmp/longjmp.
- An event-driven (as opposed to polled), non-MVC GUI frameworks.

FEATURES MISSING FROM BETA VERSION

The following features are referred to in the beta manual, but were not available at the time of review.

- Only part of the GUI builder was present: The source code was protected, and no documentation was provided.)
- Inspectors were partially operational, without access to properties, structured storage, or self.
- The Binary object loader/unloader was not available, nor was there automatic source code management (limited to file-in, file-out, with no crash recovery).
- WorldScript/Unicode was incomplete.
- The debugger was partially operational, with no access to self or modification of temporaries.
- The compiler had debug/trace code installed and lacked speed.
- DAL interface.
- 32K limit on text views.
- No network classes.

PROBLEMS

STA is a young product, and it shows in many ways. While application crashes were rare, they did occur. On the other hand, QKS technical service was especially responsive, both in my personal experience and as related by others.

The STA virtual machine proved fairly stable for any software named "1.0.1." Only a few application crashes occurred during my use. However, there does not appear to be the equivalent of a "changes file," and all source code is currently kept in the image. In this situation, one quickly learns to save often! On the list of promises is a source-code database of some kind, with multi-user facilities.

Short of outright crashes, a number of features had problems that might be expected in such a young product. For example, the messages in the debugger window were often wrong or misleading. I became used to routinely not trusting what it said, and opening a full debugger to see what was *really* happening. Even then, the debugger is but a shadow of what Smalltalkers have come to expect. You cannot evaluate expressions in the debugged context, nor can you change the method and re-start, return arbitrary expressions, or modify temporaries.

I was unable to send messages to "self" in either an inspector or debugger, which greatly reduced the usefulness of the debugging environment, since one often goes into an inspector to evaluate expressions, like self halt problemMethod. On the promise list is a true breakpoint facility, which would be a welcome improvement over the typical Smalltalk habit of inserting halts in the source code.

In general, the development environment has rough edges. The browsers are impressively colorful, but lack the spatial efficiency that users of other Smalltalks take for granted. For example, over half the area of a typical inspector is taken up with a large Properties check-box area, which apparently displays static flag information that might be more efficiently displayed as a binary or hex number and legend. Code browsers devote nearly an inch of the view to iconic buttons. This is not a product that is comfortable to use within a 640 by 400 screen!

The euphemistically named *Beta Manual*, which seemed to be galley proofs of several chapters combined with some design objectives, was nearly useless, especially to an experienced Smalltalker. Most of it was an overview of O-O principles, and some of the example code did not even work. Hopefully, QKS is shipping a final manual by the time you are reading this.

Mac users are generally resigned to incompatibilities with various system extensions. I found that Now Software's WYSI-WYG Menus 3.0.1 caused the format menu to be unusable, and that Adobe Type Reunion 1.1 disabled the font menu. No other system extensions proved troublesome. (Although I did not try STA with every system extension in the world, I normally have three rows of icons at boot-time!)

Compared to ST-80 (and even ST/V), there are few comments in the code. This is a terrible thing to do in a system that relies so heavily on source code access, and is especially troubling to those familiar with a different Smalltalk, since many methods are close enough to make you think you know them, but they operate differently. The lack of a manual would be little trouble to experienced Smalltalkers if the source were better commented.

STA handles modified keystrokes in a non-conventional way, following the currently installed keyboard resource (KCHR) for only shift, caps lock, and option modifiers. This is annoying and frustrating for Dvorak typists and those who remap their function keys to their liking. QKS argues that there is no standard for other modifiers, but if Claris and Microsoft can agree that command plus whatever key is defined as "Q" always quits an application, QKS should, too! If you use a macro program such as QuicKeys, be prepared to lose your custom function key mappings while running STA.

Following Digitalk's lead, QKS has protected some of its source code. Just when things start to get interesting in a debugger, you are confronted with

```
** QKS has removed the source **
```

This is a regrettable, but growing, trend in Smalltalk—some may choose to stick with ParcPlace simply for their complete source code access, especially to the compiler classes. Given the lack of a credible manual, the lack of *any* source, commented or not, is inexcusable. QKS has several years headstart on any new competitor (and ParcPlace and Digitalk aren't paying attention to QKS yet), so source code protection is no more forgivable than copy protection.

Speaking of which, a real nuisance is the serial-number-entry style of copy protection QKS has implemented. Moving an image causes it to assume that it has just been installed, and it requests a 13-character key before doing anything useful. People who need to test their work on different platforms will hate this. I had particular problems with this, since I was moving a single image between two machines for comparative benchmark measurements. QKS should follow the lead of all major software vendors, and exhibit some trust in the ethics of their customer base.

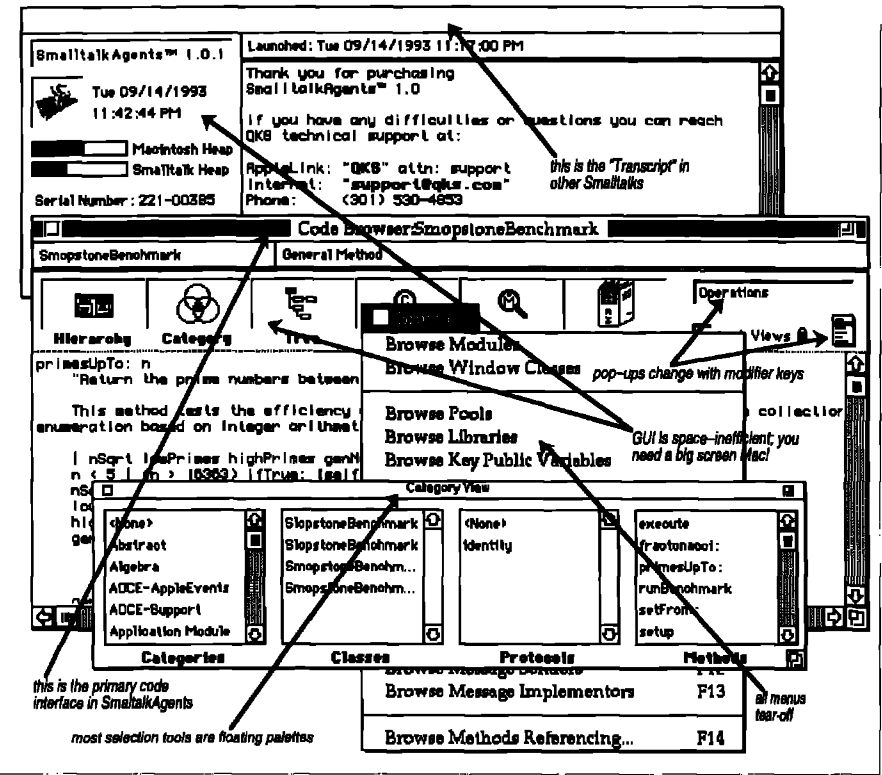
COMPATABILITY

This is a burning issue of such importance that it may make or break the future of any new Smalltalk implementation. QKS has made a number of controversial decisions that impact portability with existing Smalltalk dialects, decisions that change the very relationship between basic system classes.

Protocol has arbitrarily been changed from the de facto Smalltalk standard. For example, in other Smalltalk dialects, the message "|" is only sent to a Boolean, and it answers the logical OR of the argument and the receiver. STA defines it as a numeric OR, which is called "bitOr:" in all other Smalltalks. This is much more severe a change than the spelling of *metaclass*, for example. (ParcPlace calls it Metaclass, Digitalk calls it *MetaClass*.)

This is simply wrong—it doesn't matter so much if one wants to change the *implementation* of classes (as QKS has with collectives), but the very definition of an object is its *behavior.* Luckily in this case, it is not such a problem, since boolean operators that evaluate their argument are discouraged in favor of "short-circuit" boolean messages, such as and: and or:.

But wait—we're not out of trouble yet. Quick, what is 347 and: [true]? If you have been with Smalltalk for some time, you might answer that it doesn't make sense to send and: to 347, but in STA, this statement evaluates to true. An interesting side


SmalltalkAgents™ 1.0.1
Tue 09/14/1993
11:42:44 PM
Macintosh Heap
Smalltalk Heap
Serial Number: 221-00385
Launched: Tue 09/14/1993 11:17:00 PM
this is the "Transcript" in other Smalltalks
Code Browser:SmopstoneBenchmark
SmopstoneBenchmark
General Method
Hierarchy
Category
Operations
Views
Browse Modules
Browse Window Classes
Browse Pools
Browse Libraries
Browse Key Public Variables
Browse Message Implementors F13
Browse Methods Referencing... F14
pop-ups change with modifier keys
GUI is space-inefficient; you need a big screen Mac!
Category View
Categories
Classes
Protocols
Methods
this is the primary code interface in SmalltalkAgents
most selection tools are floating palettes
all menus tear-off

```
Thank you for purchasing
SmalltalkAgents™ 1.0

If you have any difficulties or questions you can reach
QKS technical support at:

AppleLink:  "QKS" attn: support
Internet:   "support@qks.com"
Phone:      (301) 530-4853
```

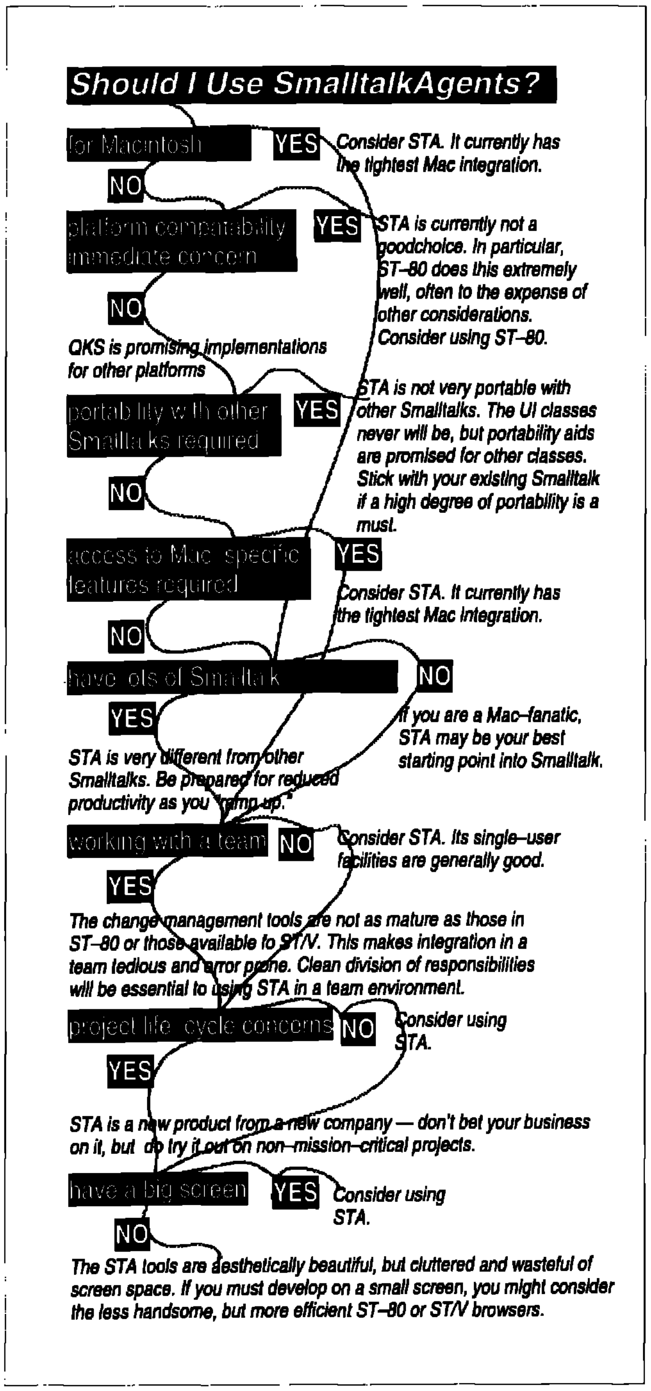
```
primesUpTo: n
    "Return the prime numbers between

    This method tests the efficiency
enumeration based on Integer arithmet

    | nSqrt lowPrimes highPrimes genN
    n < 5 | (n > 16383) ifTrue: [self
```

Categories	Classes	Protocols	Methods
<None>	SlopstoneBenchmark	<None>	execute
Abstract	SlopstoneBenchmark	identity	fractonacci:
Algebra	SmopstoneBenchm...		primesUpTo:
AOCE-AppleEvents	SmopstoneBenchm...		runBenchmark
AOCE-Support			setFrom:
Application Module			setup

Figure 1. SmalltalkAgents.



Should I Use SmalltalkAgents?
for Macintosh
YES
Consider STA. It currently has the tightest Mac integration.
NO
platform compatability immediate concern
YES
STA is currently not a goodchoice. In particular, ST–80 does this extremely well, often to the expense of other considerations. Consider using ST–80.
NO
QKS is promising implementations for other platforms
portability with other Smalltalks required
YES
STA is not very portable with other Smalltalks. The UI classes never will be, but portability aids are promised for other classes. Stick with your existing Smalltalk if a high degree of portability is a must.
NO
access to Mac specific features required
YES
Consider STA. It currently has the tightest Mac integration.
NO
have lots of Smalltalk
NO
If you are a Mac–fanatic, STA may be your best starting point into Smalltalk.
YES
STA is very different from other Smalltalks. Be prepared for reduced productivity as you ramp up.
working with a team
NO
Consider STA. Its single–user facilities are generally good.
YES
The change management tools are not as mature as those in ST–80 or those available to ST/V. This makes integration in a team tedious and error prone. Clean division of responsibilities will be essential to using STA in a team environment.
project life cycle concerns
NO
Consider using STA.
YES
STA is a new product from a new company — don't bet your business on it, but do try it out on non–mission–critical projects.
have a big screen
YES
Consider using STA.
NO
The STA tools are aesthetically beautiful, but cluttered and wasteful of screen space. If you must develop on a small screen, you might consider the less handsome, but more efficient ST–80 or ST/V browsers.

Figure 2. The pros and cons of SmalltalkAgents.

effect is that such boolean messages are no longer commutative: 347 and: [true] answers true, but true and: [347] answers 347. This is going to be a maintenance nightmare—when someone changes the order of short-circuit boolean messages to improve performance, the behavior of entire systems may change.

(Perhaps one should not be so hard on QKS for this. Starting with VisualWorks, ParcPlace changed the semantics of and:—along with all other messages that take a no-argument block—simply by implementing "value" in Object to return self. The statement true ifTrue: 347 evaluates to 347, but takes much longer to execute than true ifTrue: [347]. "Hidden-time" is a ParcPlace weakness (as with different block semantics); it would be regrettable to see QKS follow that path.

QKS defends their boolean equivalence policy vigorously, citing C and Lisp (while conveniently ignoring Pascal, Modula-2, Ada, et al.), and VM efficiency as reasons. They have taken considerable criticism from the Smalltalk community on this, and may yet yield to pressure to conform with established boolean semantics.

Other compatibility issues are more easily justified, and generally have workarounds. For example, much of the collection and stream hierarchy is subsumed by class List, but with multiple libraries (any of which can be declared as a pool), it is simple to create aliases for List called Array, OrderedCollection, WriteStream, etc. QKS is promising eventual file-in compatibility for major base classes, but as of version 1.0.1, it is safer to assume that there simply is no compatibility—even basic class creation methods, such as needed to file in, are completely different and incompatible.

FEATURE COMPARISON

Table 1 lists some items for comparison among some Smalltalk implementations available for the Macintosh. All measurement figures are rouded to two significant figures, which is within the variation we observed between trials.

The Samuelson benchmarks (The Smalltalk Report, June 1993) were run on two different platforms, both as a consistency check, and to assess the contribution of a floating point processor. The "no FPU" tests were performed on a Mac PowerBook Duo 210, which has a 25MHz 68030 and 12MB RAM. All other measurements were made on a Mac IIci, which has a 25MHz 68030 with a floating point processor, 20MB RAM, and a 32K cache card. All measurements were made under "maximum performance" conditions—no system extensions or other applications running, 1-bit video, and no power-saving in effect—using System 7.1.

With two exceptions, all measurements were performed using the "preferred memory" partition—if an implementation did not "prefer" extra memory, I did not offer it. I felt that changing the memory partition for one or more implementations would be arbitrary, and difficult to do for all of them in a fair manner.

The exceptions are the two "no FPU" measurements for VisualWorks. The Duo's 12 MB of RAM was not enough to allow VisualWorks to run with its preferred memory partition—the Finder indicates that those measurements were run in 9,560K. Note the dramatic, 6:1 reduction in performance in these cases.

Due to portability problems, many of the individual benchmark tests had to be modified to run under STA. Two SmopstoneBenchmark tests, primesUpTo: and streamTestsOn: would not complete, and were removed from the suite. The smopstone numbers are the geometric mean of the remaining tests in the suite.

The only VisualWorks image I had available was ENVY/Developer R1.41a as delivered by Object Technology International. This difference should not greatly impact most measurements, except for the number of classes and methods. (A PPS-supplied VisualWorks image is 4,694K and has 685 classes and 14,055 methods.)

The measured start-up time was for a pure image, as delivered from the vendor. The measured save time was for the image plus the benchmark code. All disk-related measurements were made from a Fujitsu 1.2 GB drive with 11mS average access time.

The first column for each implementation lists that implementation's needs relative to the greatest in the group. In general, lower is better, although items such as "number of classes" are better the higher the number.

SUMMARY

The SmalltalkAgents "post-Beta, but pre-real-thing" that I reviewed is brimming with promise. It pushes the envelope - ParcPlace and Digitalk are busy fighting over the Fortune 500 MIS Sun/PC marketplace, and have neglected innovation on the Mac.

STA's performance seems remarkable. Although it benchmarks somewhat slower than VisualWorks, it *feels* faster—windows open quicker, menus pop up faster, etc. VisualWorks employs dynamic translation of Smalltalk bytecodes into cached native machine code, a technique that excels with small, repeated loops, which is what these benchmarks measure. The Samuelson Benchmarks have acknowledged shortcomings, foremost of which is the lack of "huge operation" measurements; such "Shopstones" could have vastly different results than those obtained with low- or medium-level measurements.

Both exciting and troubling are basic changes QKS has made to Smalltalk. The addition of structured storage has the promise of greater integration of Smalltalk with other languages and systems, and the unified object model with weak references holds the possibility of less obtrusive garbage collection, and simple persistent storage and remote object facilities.

Yet this review was a difficult task: I *wanted* to like STA; there is much to like about it, and I want it to succeed, but version 1.0.1 is simply not ready for use as a workhorse commercial development environment.

The realities of business sometimes dictate that products are marketed and delivered before they are ready - unfortunately, this has become the norm in software, rather than the exception. Given that the largest name in software produced their first credible windowing system with a version of "3.0," I would encourage early-adoptors to purchase and become familiar with SmalltalkAgents in non-critical applications until the *real* "real thing" is available.

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

...continued from page 12

ber of projects, we have focused a lot of attention on designing the particulars of conversations. This fine-tuning really pays off for systems where improving the quality of human-computer interactions is a high priority.

In these situations, we pay particular attention to finding the main course and painstakingly ensure that this indeed is the most common task that users want to perform. We also use a variety of user interface design techniques to construct and test theories about preferred conversation patterns. This is lot of work. Conversations can be crafted by interface specialists (and iteratively prototyped by designers) if the projects size and scope warrants this attention, or they can be done more informally.

Even though it is a good idea to separate the details of user interface and information presentation from our business objects, the way a user converses with our software can have a significant impact. In my experience, no application area is immune. I personally know about oscilloscope control software, customer information systems, and trip-planning applications where seemingly minor interface features placed significant demands on the underlying object model. Any effort to work out the interface issues early made our job of developing the other parts of the software that much easier.

CONCLUSION

Users can work with analysts and object designers to formulate and tune system requirements. People from business, analytical, and object design disciplines can come together, learn from each other, and generate meaningful descriptions of systems that are to be built. Each participant and each project has slightly different concerns and needs. Practical application of use cases can go a long way to improve our ability to deliver just what the customer ordered.

References

1. Jacobson, I., M. Christerson, P. Jonsson and G. Overgaard. Object Oriented Software Engineering: A Use Case Driven-Approach, Addison-Wesley, Reading, MA, 1991.

2. Davis, A. Software Requirements Objects, Functions and States, Prentice Hall, Englewood Cliffs, NJ, 1993.

Rebecca Wirfs-Brock is the Director of Object Technology Services at Digitalk and co-author of Designing Object-Oriented Software. She has 18 years of experience designing, implementing, and managing software products. For the last nine years she has focused on object-oriented software. She managed the development of Tektronix Color Smalltalk and has been immersed in developing, teaching and lecturing on object-oriented software. Comments, further insights, or wild speculations are greatly appreciated by the author. Rebecca can be reached via email at rebecca@digitalk.com. Her U.S. mail address is Digitalk, 7585 S.W. Mohawk Drive, Tualatin, OR 97062.

...continued from page 1

References to Objects

IDL interface definitions contain the definitions of IDL constructs that define the basic unit of an object's abstract type. Each interface groups a portion of the operations and related abstract behavior of the object into units that are meaningful to an external client. Interface definitions may be composed and interrelated using a multiple inheritance mechanism organizes and determines object roles in a distributed system. Groups of related interfaces and other declarations may also be grouped into larger units called *module definitions* that capture client and server behavior that is related to a particular application policy or protocol.

Local objects in Smalltalk are manipulated via object references supported by the virtual machine. Remote objects in HP Distributed Smalltalk are manipulated via local surrogate objects which contain identifiers to uniquely identify the object and its interface. Local calls on these surrogates are transparently intercepted by the ORB and the message is forwarded to the remote object using this information. During remote execution, the local process thread is blocked until the result values have been received and decoded into internal Smalltalk representation. At that point, the local thread is resumed and local execution continues. Since access to remote objects is transparent to the Smalltalk programmer, operations that have been defined in IDL interfaces may be invoked as though they were local methods on local objects.

Operations on Objects

In IDL, an operation invocation requires a reference to a target object, a description of the operation to be performed, and a specification of the argument values. This is completely consistent with the Smalltalk object model; thus, in HP Distributed Smalltalk, a remote method invocation is indistinguishable from a local invocation. The only difference to the programmer is the amount of time required to complete the request.

IDL and Smalltalk message syntaxes both allow zero or more input parameters to be supplied in a request. For return values, however, Smalltalk provides a single result object whereas IDL allows multiple output parameters to be returned from a single invocation. This is handled by returning an array with all of the output parameters included in the order of their declaration in the IDL operation declaration (the result value is given last). IDL operations declared to have a type void result, but have a single output parameter are returned as single values just like operations with a single result value and no output parameters: the single value without an enclosing array. All parameters are allocated and reclaimed from the Smalltalk heap.

In addition to in and out parameters, IDL also allows inout parameters to be defined. These parameters are expected to be supplied in the invocation and will be returned as out parameters in the resultant array. In HP Distributed Smalltalk, the in and out values will be distinct Smalltalk objects, rather than sharing some portion of the heap (as in C, for example). The programmer may use #become: with caution to achieve this effect if it is desired.

Exceptions

IDL allows each operation definition to include information about the kinds of runtime errors that may be encountered. These are specified in an exception definition that declares an optional error structure which will be returned by the operation in lieu of its normal results, should an error be detected. Exception handling is implemented using the normal Smalltalk-80 exception handling classes Exception and Signal. Thus to raise an exception, the programmer can merely invoke #error:. To return an appropriate error value, the #raiseWith:errorString: operation may also be used. Consider the example Smalltalk fragment that raises the BAD_INV_ORDER exception (one of the standard exceptions defined in interface Object):

```
^ErrorSignal raiseWith: (Array
        with: #'BAD_INV_ORDER'
        with: (Array with: minor with: #NO))
    errorString: ' routine invocations out of order'
```

To allow the ORB to return the error result structure correctly to the sender of the method, an array must be returned as the parameter of the error. Here, the symbolic name of the event is provided in an array along with the type-structure representation of the required error result values. These values will be processed by the ORB to ensure that the same exception is raised in the context of the client of the remote operation.

As with normal Smalltalk exceptions, a #handle:do: or other recovery method may be used to catch and recover from these exceptions. The main difference is that the ORB call context will have already unwound to the site of the remote call before the exception is raised. This greatly limits the extent to which recovery can be accomplished.

Basic Datatypes

Each of the parameters of an IDL operation definition has an associated data type that must be declared in advance, since IDL is a statically typed definition language. As a result, some operations that can be implemented in Smalltalk cannot be declared in IDL at all. This is also complicated by the fact that Smalltalk has no notion of type: All Smalltalk values are instances of a Smalltalk class. To be able to construct valid calls on IDL operations, however, a mapping must be devised. Fortunately, the following type-class mapping works well enough, and useful distributed systems can be constructed which use IDL definitions. What is needed is for the Smalltalk programmer to understand the mapping and its limitations.

In HP Distributed Smalltalk, the following classes are mapped directly to the required IDL basic datatypes. Instances of these classes are passed by value during remote method invocation. This means that a copy of the argument instance is presented to the server implementation.

Boolean values true and false are used to represent IDL boolean types. Character values are used to represent IDL char types. Float and double values are used to represent IDL float and double types. Integer values are used to represent IDL long and short integer types Character and SmallInteger values may be used to represent IDL octet types. String values may be used to represent IDL string types.

Constructed Datatypes

IDL Constructed types must also be mapped to Smalltalk constructs. Here there is more latitude for the mapping designer, and the choices are less clearly defined. The following mechanisms have proven themselves to be useful.

Enumerations An IDL type-enumeration value is constructed by using the Smalltalk symbol that is identical to the IDL enumerator string. This allows Smalltalk programmers to use symbols freely and to declare the symbol values that are legal in each parameter situation. To preserve the IDL ordering requirements on enumerations, each enumeration declaration produces a constant array which contains its values in declared order. This constant array may be accessed from HP Distributed Smalltalk's ORBConstants pool dictionary using the fully scoped name of the type.

For example:

```
module Chart {
      enum ChartStyle {line, bar, stacked, pie};
      ... };
```

allows the symbols #line, #bar, #stacked, and #pie to be passed as ChartStyle values in requests. Their sorting order is maintained in an array #(line bar stacked pie) named #'::Chart::ChartStyle' in the ORBConstants dictionary.

Sequences IDL sequence values are constructed using instances of the Smalltalk Collection subclasses. Since IDL sequences are all restricted to have homogeneous elements of the same type, however, this represents a limitation which the Smalltalk programmer must take into consideration when defining interfaces. For situations where the Smalltalk value of choice is a heterogeneous Collection subclass, consider the use of IDL type-structure instead.

IDL sequence values returned from remote operations are instantiated as Smalltalk OrderedCollections by default. This may be overridden to accept and return any Smalltalk class which has the ORB-required #at: and #at:put: methods by using a CLASS pragma in the IDL definition.

Structures IDL structure values may also be constructed using instances of the Smalltalk Collection subclasses. Heterogeneous instances of OrderedCollection subclasses may be passed as parameter values to IDL operations, assuming that their runtime elements correspond to the declared IDL types. At the server end, an OrderedCollection will be provided to the server, which has the same element values as were passed in by the client.

In addition, any Smalltalk class that has the ORB-required method selectors can be used as type-structure values for remote calls. To use this capability, the Smalltalk class name must have been declared in a CLASS pragma associated with the IDL type declaration, the class must implement accessor methods corresponding to each IDL struct field name, and either the class or the instance must include a combined method (of the form: #f1:f2:...fn: where each f-i is the *i*-th field name in the struct) to set the instance state. By convention, field names which contain underscore characters are converted to more conventional Smalltalk notation. For example, a field named "my_field" in IDL would require an accessing method named #myField.

For example:

```
struct Point {
      long      x;
      long      y;
      } CLASS = Point;
```

will allow Smalltalk points to be passed and returned without programmer intervention, since point has methods #x, #y, and also the class method #x:y:. Of course, points containing floating point or rational values must be handled differently.

Unions IDL type-unions are represented in Smalltalk by instances of the Association class, where the key of the association is the union's discriminator value and where the value of the association is the union's member. For proper operation during remote invocation, both the key and the value of the association must be of a type which is compatible with the respective union declaration roles.

For example:

```
enum   Numeric { Integer, Real, Fraction };
struct Rational { long numerator, denominator; };
union  Number switch ( Numeric ) {
  case Integer:    long        intVal;
  case Real:       double      fltVal;
  case Fraction:   Rational    fracVal;};
struct Point {Number x, y; };
```

allows the full range of Smalltalk point values to be (awkwardly) represented in IDL. An instance of such an IDL Point may be constructed in Smalltalk as follows: Array with: #Integer -> 5 with: #Fraction -> (3 / 4). Not a pretty sight, but it works.

Type Any In some situations, however, it is just not possible to know a parameter's type at IDL definition time. Thus, IDL provides a dynamic type-any that carries its associated typing information with it at runtime. In HP Distributed Smalltalk, instances of type-any are represented by an Association of the form (obj typeObject -> obj) where the #typeObject method returns a Repository meta-object that can properly encode/decode the object's value in an ORB packet. Since all Smalltalk objects are inherently typed, it is never necessary to explicitly create the type-any Association; the instance's #typeObject method will be called by the ORB as needed automatically. Thus, normal Smalltalk objects may be passed as type-any parameters. To assist with easy conversion of type-any Associations to normally typed Smalltalk instances, the method #value has been added to class Object to return the object itself.

Thus, potentially remote operations that yield type-any values can be handled uniformly by appending the #value method to the result. If the operation is local and is returning a local

Smalltalk object directly, then there is no net effect. However, if the operation is remote and actually returns an Association value, then the value of the Association (the intended result) results. For example:

```
struct Point { any x, y; } ;
```

allows Smalltalk Points to be passed as parameters to methods which remember to add a #value call to the point's coordinates before they are used. A better solution than with type union perhaps, but it is still awkward.

Constants

IDL allows constant expressions to be declared in interface and module definitions. In HP Distributed Smalltalk, such expressions are evaluated to produce constant values each time the Interface Repository is changed. During ORB operation, IDL constant values are stored in a pool dictionary ORBConstants under the fully qualified name of the constant.

For example:

```
interface foo {
  const long bar = 7;
};
results in the following:
(ORBConstants at: #'::foo::bar') = 7
```

Attributes

IDL attribute declarations are a shorthand mechanism to define pairs of simple accessor operations, one to set the value of the attribute and one to get it. Such accessor methods are common in Smalltalk programs as well, thus attribute declarations are mapped to standard methods to get and set the named attribute value, respectively.

For example:

```
attribute string title;
attribute string my_name;
```

means that Smalltalk programmers can expect to make #title and #title: calls to get and set the title attribute of the object. By convention, attribute names that contain underscore characters are converted to more conventional Smalltalk notation. For example, "my_name" results in selectors #myName & #myName:.

Signatures of Standard Interfaces

CORBA defines a minimal set of standard interfaces that define types and operations for manipulating object references, for accessing the Interface Repository, and for Dynamic Invocation of operations. These operations have been implemented in HP Distributed Smalltalk, and may be invoked using the operation binding discussed previously.

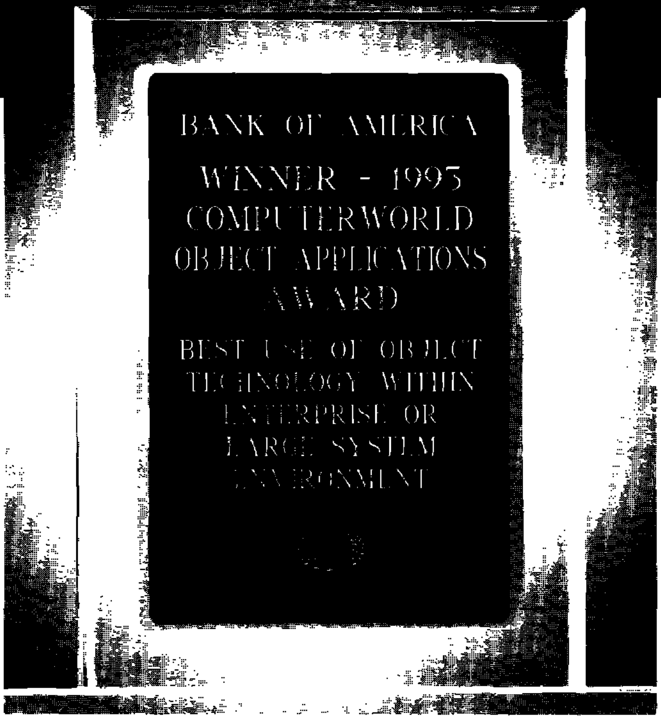
For example, an object reference to the Interface Repository meta object supporting an object's IDL interface is obtained by invoking the #getInterface method. Other calls, from the standard interfaces InterfaceDef, Container, Contained, and Object may also be invoked on this metaObjRef to further elucidate its nature. While the mechanisms provided in the Dynamic Invocation Interface (DII) may be used by the HP Distributed Smalltalk programmer to dynamically construct object requests, the more conventional #perform:withArguments: method works the same and is preferred to the more verbose DII.

SUMMARY

The preceding is a description of the IDL to Smalltalk language binding which is provided by HP Distributed Smalltalk. In addition, HP Distributed Smalltalk also contains a number of Common Object Services and Sample Applications which extend and apply the CORBA standard. These services allow the easy creation and manipulation of true distributed applications by providing standard distributed building blocks that can be used by developers. For more detailed information, contact the author. ▣

Jeff Eastman is a consulting engineer in Hewlett-Packard's Distributed Computing Program, where he is the architect of HP Distributed Smalltalk and the designer of its IDL language binding. He has over 18 years experience in software development at HP and has held positions in development, research, and management. His experience with Smalltalk dates back to 1980, and he has been active in object-oriented development at HP during the interim. Jeff holds a Ph.D. in electrical engineering from North Carolina State University. He can be reached at: jeastman@cup.hp.com. His mailing address is: Hewlett-Packard Company, 19447 Pruneridge Road, Cupertino, CA 95014.

LOOK WHAT HAPPENED WHEN DIGITALK BROKE INTO THE BANK.



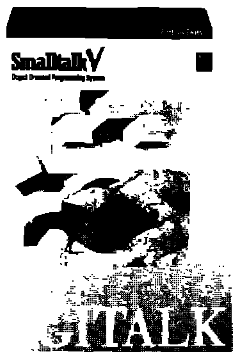
Congratulations to Bank of America on their new 11-state wide area network. A system they call "the most sophisticated distributed network in the world."

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