The Smalltalk Report

The International Newsletter for Smalltalk Programmers

September 1993 Volume 3 Number 1

BUILDING OBJECT-ORIENTED FRAMEWORKS

by Nik Boyd

Contents:

Object system architects have long understood the value of frameworks. Frameworks provide a powerful way to organize and build interactive object systems. While classes define the structure and behavior of individual objects, frameworks define the structure and behavior of interactive object systems and subsystems (architectures). Just as classes provide leverage from the reuse of solutions to component problems, frameworks provide leverage from the reuse of solutions to systemic problems. Classes and frameworks complement each other for object modeling coordination.

Object system architects have sought ways to discover, describe, and define useful frameworks. This article explores some issues related to designing and building object systems, especially using frameworks. This article proposes that frameworks can be made first-class objects and describes the implementation of a Framework superclass for Smalltalk.

First-class frameworks provide a way to formalize the relationships between the objects in a system and factor out their patterns of interaction. Framework classes provide new opportunities for design, development, and reuse in object systems. They can be used to create very general or specialized event-driven systems. By making frameworks first-class objects, they derive and supply the same benefits as other objects: They can be built and reused with existing tools.

HOW THIS WORK EVOLVED

Smalltalk's browsers provide essential tools for quickly building and evolving objects. These tools organize and present objects and their definitions. The internal workings of these browsers can be quite complex. As a result, the classes that implement these browsers tend to have many methods.

The complexity of these browser classes contributes significantly to the difficulty of developing new tools for Smalltalk. This observation leads naturally to the following question: How can these browsers be broken down into more easily integrated and reusable components? The Model-View-Controller (MVC) framework[1] and its alternatives[2,3] provide great value, but do not completely resolve the problem of component integration.

Early experiments with refactoring some new tools led to ways of loosely coupling their components using a kind of "smart" linkage. These component connections included their own behavior. After exploring some alternatives, it became obvious that these experiments had produced a way of implementing mediators.[4] Patterns began to emerge when the browser components were coupled together using mediators. This observation led naturally to the realization that some of these interaction patterns could be factored out and reused. Such refactoring created first-class framework objects whose behaviors are governed by interaction contracts.[5] Framework classes map interaction contracts directly onto inheritance hierarchies.

continued on page 4...

John Pugh

Paul White

EDITORS' CORNER

It's been a busy spring and summer for conferences. Here are a few Smalltalk-related perspectives on those that one or the other of us has attended recently.

In May, Digitalk held their second conference for developers, DEVCON'93 in Costa Mesa, CA. The audience, which was populated by many representatives from banking and insurance companies, reflected very much the move of the MIS community into Smalltalk development. The conference program catered to this community with a heavy emphasis on the use of Smalltalk/V and PARTS in client-server computing. In one of the liveliest presentations, Amarjeet Garewal from the Bank of America described his firm's client-server development, ACA (A Cooperative Application). ACA facilitates distributed computing using Smalltalk and legacy systems, and was an award winner in the Object Applications category at the recent ObjectWorld conference. Watch for an upcoming article from Amarjeet in the REPORT. For Smalltalk aficionados who wanted to learn more of the "meta-world" of Smalltalk, Dave Smith from IBM give an inimitable reprise of his "Behavior of Behavior" presentation. SMALLTALK REPORT columnist Kent Beck dispelled a few Smalltalk myths and provided some invaluable insights into how to write high-performance Smalltalk programs. It was also Digitalk's 10th anniversary—they threw a good party!

June was the month for the large ObjectWorld conference in San Francisco. The Smalltalk story of note there was the demonstration of Hewlett-Packard's Distributed Smalltalk product—the first complete implementation of the Object Management Group's CORBA specification for distributed computing. Using Distributed Smalltalk, programmers can access distributed objects transparently without regard for whether the objects are local or remote. At the conference, users of Distributed Smalltalk in the HP booth were able to access objects residing in a Gemstone database in the Servio booth. Distributed Smalltalk consists of approximately 150 classes that sit on top of ParcPlace Systems' VisualWorks product. Watch out for upcoming articles on distributed computing with Smalltalk in future issues.

For the past few years, many people have been discussing the issue of frameworks as a mechanism for achieving reuse in object-oriented systems. For most, however, the issue of finding these frameworks is elusive, to say the least. As this month's lead article, Nik Boyd provides a description of how frameworks can be made first-class objects by introducing a Framework abstract class to Smalltalk and provides examples illustrating how best to use it.

Three of our columnists check in this month. Alan Knight addresses the issue we raised in our last editorial, namely making extensions to the base Smalltalk environment. In his column, he reports on "home-brewed" enhancements that have been posted to the Internet news group. In his column this month, Kent Beck continues his discussion of using inheritance effectively by introducing a pattern to be applied when attempting to make decisions concerning the factoring of subclasses. Greg Hendley and Eric Smith are back, describing how to take advantage of the object-dependents mechanism provided by Smalltalk when trying to keep multiple windows that are displaying inter-dependent information in sync. Finally, Dan Lesage reviews Dan Shafer's new book, SMALLTALK PROGRAMMING FOR WINDOWS.

Enjoy the issue—and welcome to our third year!

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. Submit articles to the Editors at 509–885 Meadowlands Drive, Ottawa, Ontario K2C 3N2, Canada, 613.225.8812 (v), 613.225.5943 (f).**
PRINTED IN THE UNITED STATES.

The Smalltalk Report

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

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

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

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

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

Art/Production
Kristina Joukhadar, Managing Editor
Susan Culligan, Pilgrim Road, Ltd., Creative Direction
Karen Tongish, Production Editor
Gwen Sanchirico, Production Coordinator
Robert Stewart, Computer Systems Coordinator

Circulation
Stephen W.Soule, Circulation Manager

Marketing/Advertising
James O. Spencer, Director of Business Development
Jason Weiskopf, Advertising Mgr—East Coast/Canada
Holly Meintzer, Advertising Mgr—West Coast/Europe
Helen Newling, Recruitment Sales Manager
Sarah Hamilton, Promotions Manager—Publications
Jan Fulmer, Promotions Manager—Conferences
Caren Polner, Promotions Graphic Artist

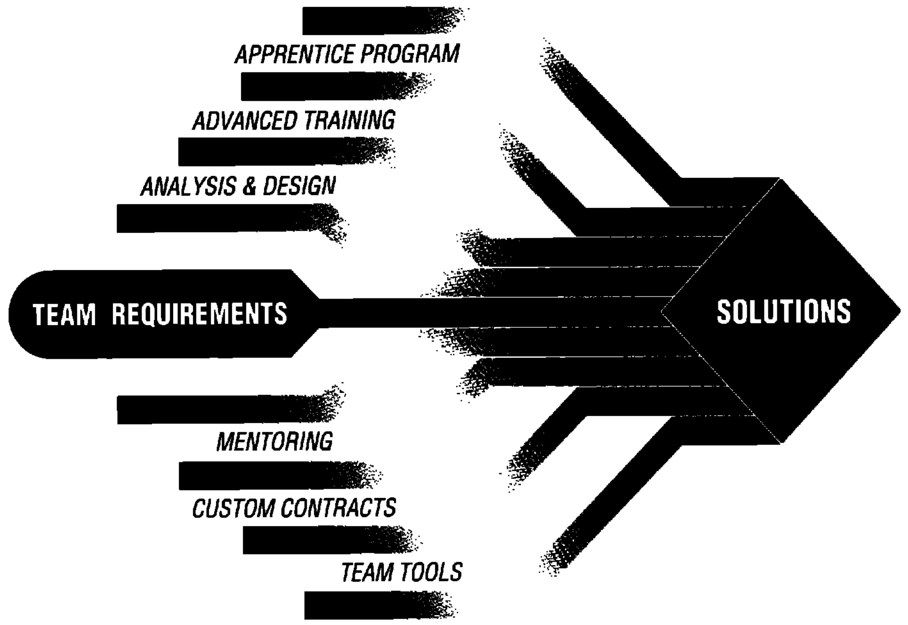
Administration
David Chatterpaul, Accounting Manager
James Amenuvor, Bookkeeper
Margot Patrick, Assistant to the Publisher
Claire Johnston, Conference Manager
Cindy Baird, Conference Technical Manager

Margherita R. Monck
General Manager

SIGS PUBLICATIONS

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

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

...continued from page 1

This article describes the results of these experiments: The role that frameworks can play in system design, and how framework classes can be used to define the structure and coordinate the behavior of objects in systems. We begin by exploring some issues related to object design and system design.

OBJECT DESIGN AND SYSTEM DESIGN

We often solve large problems by breaking them up into smaller problems and combining the solutions (divide, understand, integrate: *solve e coagula)*. Just so, we can divide large systems of interacting objects into smaller collaborations, or subsystems. This allows us to better understand and manage the structure and behavior of the larger system.

Two key concerns of object system architects are the *right factoring* of behavior and the *right coupling* of objects. Although different aspects of a design, factoring and coupling decisions often influence each other. For example, creating a new object class presents a question that arises frequently in object system design: Where does the new class belong in a class hierarchy? This critical design activity incorporates both factoring and coupling decisions because objects serve as the essential unit for both factoring and coupling in object systems.

The class location decision can be made easier by looking at the proposed service responsibilities of the new class and asking some questions. Does the new class provide the same (or substantially similar) services when compared to another existing class? Does it add new services or change the implementation of some services? Does it remove any services? When a new class shares (and perhaps adds to) the public interface of an existing class, the new class is a good candidate for subclassing the existing class. When the public interface of the new class is not substantially similar, but needs the services of an existing class, the new class should be a client of the existing class. When a new class shares some portion of the public interface of an existing class, the hierarchy may need to be revised, splitting out the shared interface into a new, more general superclass shared by both the existing and newer subclasses. Finding the best location for object behaviors is the essence of right factoring.

RIGHT FACTORING

Factoring characterizes how well responsibility for services are distributed throughout an object system or class hierarchy. Ideally, each unique piece or pattern of behavior has a unique location within each object system or class hierarchy.

Classes may be organized initially based on data and the operations on that data. However, classes should finally be organized based on their service responsibilities and collaborations. Each object in a system is assigned responsibility for providing certain services to its clients. Responsibility-based design (RBD) takes the client/server approach to its logical conclusion in the design of finegrained objects and collaborative subsystems.[6-9]

Many experienced object designers have suggested that good class hierarchies tend to be *deep and narrow*. A hierarchy is considered deep when there are many intermediate superclasses between the most specialized classes and the top of the hierarchy. A hierarchy is considered narrow when each class in the hierarchy adds relatively few public services.

Class libraries tend to evolve over time until they become stable and mature. However, we must be careful if we don't want such stability to mean that they ossify! This can happen in large systems when a few basic objects are used repeatedly, creating many dependencies. The stability created by such dependencies may argue against redesign, creating a kind of inertia.

Early design evolution should be encouraged in order to prevent premature stability. Object modeling[10] can help to accelerate the process of evolution during class and system design. Design iteration provides opportunities for revisiting and revising object and system designs through *refactoring*.[11]

Refactoring applies one or more kinds of behavior preserving transformation to an object model. The behavior of the modeled objects is redistributed so that they are simpler and provide better opportunities for reuse. Even fairly stable class hierarchies may be improved by subjecting them to refactoring.[12]

One frequently used example of refactoring is generalization. When two or more subclasses share some common behavior, a new more general superclass can be created by factoring out the shared behavior.

Many of the transformations permitted by refactoring can be automated. Automating the refactoring process could eventually lead to the development of a kind of "lint" eliminator for object designs.

RIGHT COUPLING

Coupling characterizes the relative visibility and independence of objects in relation to each other. Ideally, objects and classes should only be visible to those clients that need to see them.

When one object depends implicitly on another, they are *tightly coupled*. Object instances are tightly coupled to their classes. When one object depends directly on the visibility of another, they are *closely coupled*. Smalltalk instance, class, and pool variables are are closely coupled to the instances that reference them.

When one object references another only indirectly through an opaque reference or through some accessing or structural traversing message(s), it depends only on some portion of the other's public interface and may be *loosely coupled*. Table 1 summarizes the relationships between visibility and coupling.

Thus, appropriate visibility is essential for achieving right coupling. Often, the success of a large programming project hinges on right coupling. Right coupling can only be achieved if the system architect has an awareness of coupling and visibility

Table 1. Relationships between visibility and coupling.

Visibility	Coupling
Implicit	Tight
Immediate	Close
Opaque	Loose
None	None

Now! Automatic Documentation

For Smalltalk/V Development Teams — With Synopsis

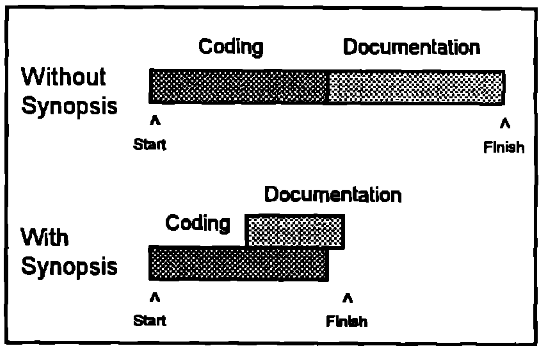
Synopsis produces high quality class documentation automatically. With the combination of Synopsis and Smalltalk/V, you can ***eliminate the lag between the production of code and the availability of documentation.***

Synopsis for Smalltalk/V

- Documents Classes Automatically
- Provides Class Summaries and Source Code Listings
- Builds Class or Subsystem Encyclopedias
- Publishes Documentation on Word Processors
- Packages Encyclopedia Files for Distribution
- Supports Personalized Documentation and Coding Conventions

Dan Shafer, Graphic User Interfaces, Inc.:
"Every serious Smalltalk developer should take a close look at using Synopsis to make documentation more accessible and usable."

Development Time Savings



Products Supported:

Digitalk Smalltalk/V Windows	$295
Digitalk Smalltalk/V OS2	$395

(OS/2 version works with Team/V and Parts)

Synopsis Software
8609 Wellsley Way, Raleigh NC 27613
Phone 919-847-2221 Fax 919-847-0650

issues, and has tools that provide him with real options for dealing with those issues.

Component classes, module classes,[13] and framework classes complement one another in controlling coupling and visibility in Smalltalk systems. They also provide complementary mechanisms for factoring. The issues raised regarding the factoring of behavior and the coupling of objects can be dealt with formally by designing objects using contracts.

DESIGNING WITH CONTRACTS

Contracts are design abstractions. They provide high-level descriptions of:

- The behavior (and structure) of a component object
- The collaborations between the components that form a subsystem
- The interactions between the participants in a framework.

Classes define the service capabilities of their instances. These services can be organized using *protocols*. Protocols are generally used to represent the contracts provided by objects. Protocols generally characterize the services they organize using descriptions derived from verb phrases such as *initializing-releasing* (instances), *accessing* (some state information), *computing* (some value).

Sometimes a complex set of related services can best be implemented and simplified by assigning responsibility for some contract(s) to a separate class. The set of resulting classes can then be organized as collaborators in a subsystem. Responsibility-based design[9] can be used when defining and refining the contracts fulfilled by components and subsystems.

In Smalltalk, module classes[13] can be used to organize and provide opaque access to subsystems. Like component classes, module classes can be instantiated. Whether through the module class or one its instances, each module serves as a gateway, providing access to the services of its internal subsystem.

Interaction-oriented design can be used when defining and refining the interaction contracts fulfilled by frameworks. In interaction-oriented design, the interactions between objects are first-class entities in the design space.[5] Using framework classes, these first-class designs can be implementated as first-class objects.

THE FRAMEWORK SUPERCLASS

Listings 1 and 2 provide the Smalltalk source code that implements the Framework superclass. The Framework superclass is intended to be subclassed to create both general and specialized frameworks. The Framework superclass is responsible for providing the following services:

- Building a framework from participants
- Resolving roles for participants
- Defining roles and their responsibilities
- Validating participants for roles
- Translating events into messages

When a framework instance is built, some of the participants are components, but some may be other frameworks. These nested frameworks are given special treatment during the assembly of the framework in which they are embedded. Each nested framework is checked for unresolved roles. If any unresolved roles are found, they are filled using participants from the embedding framework by matching their role names. Thus, naming the roles and participants in a network of frameworks is an important activity.

This feature allows system architects to design and build networks of interlocked frameworks. Small frameworks and their components can be integrated so that events propagate through the network to produce the overall behavior of a large system.

Within a framework, each object has a role and must supply certain services in order to fulfill that role. An interaction contract defines the responsibilities of the objects that form a behavioral composition. The services each object must render in order to participate in a role may be defined explicitly as part of a framework class. When these specifications are defined for the roles of a framework class, they are verified when each instance of the framework is assembled.

Although framework role validation is feasible within any language system, it is easiest to implement when the language supports reflection directly. Reflection provides objects with access to information regarding their own behavior. Sometimes this language feature is described as object self-knowledge. Smalltalk is one of the few commercial languages that support reflection.

The use of reflection by framework classes for validating role participants presents an interesting opportunity. This reflective information can be used to support the intelligent assembly of object frameworks. In Listing 1, the #assembleAs: method shows how the Collection class may be extended to support framework assembly from anonymous participants.

If the service requirements defined for each role differ sufficiently, they may be used to identify the role players needed from a collection of anonymous participants. Each anonymous participant can be examined to determine its most likely role within a framework based on the service requirements of each role. Once the roles of all the participants have been identified, the framework can be built without any need to explicitly specify their roles.

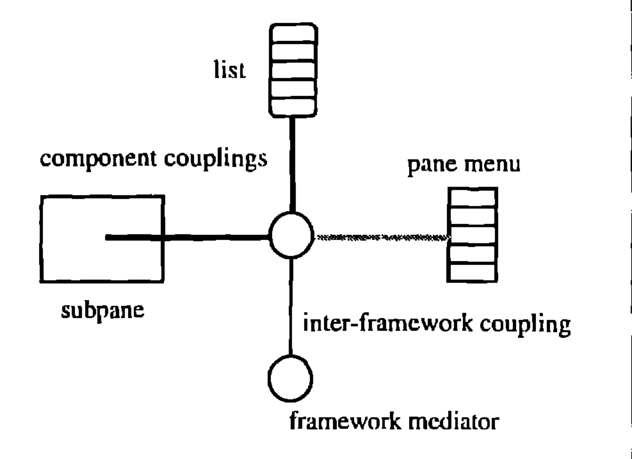


Figure 1. Key diagram.

EVENT NOTIFICATION AND TRANSLATION

The MVC framework and other similar ones typically broadcast event and change notifications to dependents. While this may be sufficient for simple frameworks, more complex frameworks need something more: the ability to target specific framework participants for event or change notification. For this reason the Framework class supports both kinds of notification mechanisms:

```
self notify: #someParticipant
  that: #somethingHappened.

self someParticipant
  notifyThat: #somethingHappened.
```

The #notify:that: request extends the Object class to provide event notification targeted at specific named dependents. The #notifyThat: request extends the Object class to provide broadcasting of events to all dependents (see Listing 1). The Framework class overrides #notify:that: to support targeting specific named participants. It also overrides #notifyThat: to translate events into actions.

SOME EXAMPLE FRAMEWORKS

The first two examples are described in Reference 5. Listing 3 shows a framework class that captures the SubjectView contract. The SubjectView contract manages a collection of views so that they all reflect the current value of a subject. By factoring out the behavior related to the contract into a separate framework class, the services that the subject and view classes must support are drastically reduced. This factoring allows these classes to be simplified to their essential behavior without concern for how they are used in a broader context.

Listing 4 shows how ButtonGroup, a specialization of the SubjectView contract, can be captured as a framework subclass. The ButtonGroup shows which button of a group of radio buttons is selected. Here again, the behavior required of the Button class is reduced, eliminating its need to retain any framework specific behavior.

The next example is derived from efforts to refactor some browser classes. A brief overview will suggest how such refactoring may proceed. The Framework superclass is subclassed by a hierarchy that supports the redirection and translation of the SubPane events used in Smalltalk/V. The class SubPaneMediator guides the interactions between one of the SubPane subclasses (i.e., Button) and some other component(s).

The component used by these mediators in addition to the subpanes is a SelectionList. The SelectionList class remembers the selection of a single item from a list of items. The item list may be either an IndexedCollection or an OrderedDictionary. The selection index of the list is either an ordinal number or an or-

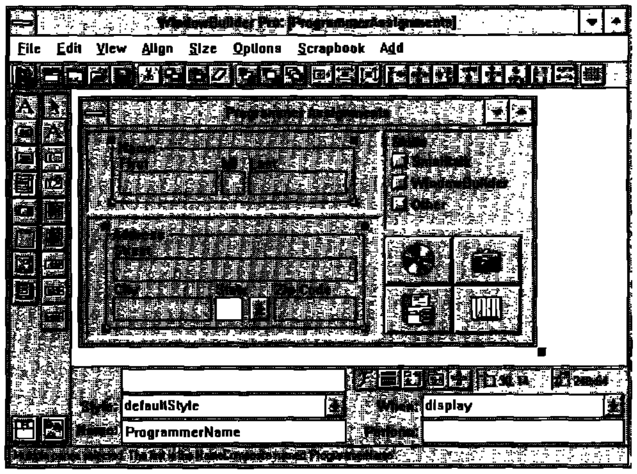
continued on page 14...

SHARE
OBJECT SYSTEMS
INC.

WINDOWBUILDER PRO!

The New Power in Smalltalk/V Interface Development

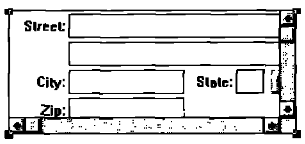
Smalltalk/V developers have come to rely on WindowBuilder as an essential tool for developing sophisticated user interfaces. Tedious hand coding of interfaces is replaced by interactive visual composition. Since its initial release, WindowBuilder has become the industry standard GUI development tool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



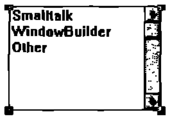
WindowBuilder Pro/V is available on Windows for $295 and OS/2 for $495. Our standard WindowBuilder/V is still available on Windows for $149.95 and OS/2 for $295. We offer full value trade-in for our WindowBuilder customers wanting to move up to Pro. These products are also available in ENVY®/*Developer*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

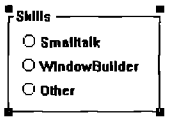
Some of the exciting new features...

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

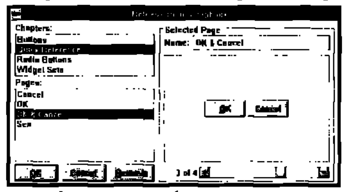


• Morphing: Allows the developer to quickly change from one type of control to another, allowing for powerful "what-if" style visual development. The flexibility allowed by morphing will greatly enhance productivity.

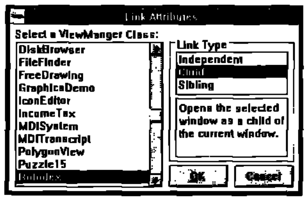


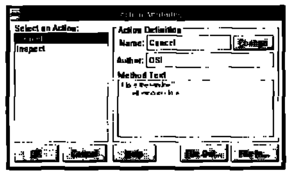


• ScrapBook: Another new feature to leverage visual component reuse, ScrapBooks provide a mechanism for developers to quickly store and retrieve predefined sets of components. The ScrapBook is a catalog of one's favorite interface components, organized into chapters and pages.



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





• 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	**5 Town & Country Village**
Fax: (408) 727-6324	**Suite 735**
CompuServe 76436,1063	**San Jose, CA 95128-2026**

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

Kent Beck

Inheritance: the rest of the story

In the June issue where I took on accessor methods, I stated that there was no such thing as a truly private message. I got a message from Nikolas Boyd reminding me that he had written an earlier article describing exactly how to implement really truly private methods. One response I made was that until all the vendors ship systems that provide method privacy, Smalltalk cannot be said to have it. Another is that I'm not sure I'd use it even if I had it. It seems like some of my best "reuse moments" occur when I find a supposedly private method in a server that does exactly what I want. I don't yet have the wisdom to separate public from private with any certainty.

On a different note, I've been thinking about the importance of bad style. In this column, I always try to focus on good style, but in my programming there are at least two phases of project development where maintaining the best possible style is the farthest thing from my mind. When I am trying to get some code up and running I often deliberately ignore good style, figuring that as soon as I have everything running I can simply apply my patterns to the code to get well-structured code that does the same thing. Second, when I am about to ship a system I often violate good style to limit the number of objects I have to change to fix a bug.

What got me thinking about this was a recent visit I made to Intelliware in Toronto. Turns out Intelliware is two very bright but fairly green Smalltalkers, Greg Betty and Bruno Schmidt (he's not nearly as German as his name). They hired me to spend two days going over the code they had written for a manufacturing application. The wonderful thing was, they had made every mistake in the book. It's no reflection on their intelligence; everyone makes the same mistakes at first.

What made their boo-boos so neat was that I was able to go in and, in two days, teach them a host of the most advanced Smalltalk techniques just by showing them how to correct errors. I'd say, "Oh, look, an isKindOf:. Here's how you can get rid of that and make your program better at the same time." Because I had a concrete context in which to make my observations, they could learn what I was teaching both in the concrete ("Yes, that does clean up the design") and the abstract ("Oh, I see. I can do that any time I would have used isKindOf:").

So, go ahead. Use isKindOf:. Use class == and == nil. Access variables directly. Use perform: a lot. Send a message to get an object that you send a message to. Just don't do any of these things for long. Make a pact with yourself that you won't stand up from your chair (or go to bed, or ship the system, or go to your grave. . .) without cleaning up first.

Some people are smart enough to write clean code the first time. At least, that's what they tell me. Me, I can't do that. I write it wrong, and then fix it. Hey, it's not like we're writing in C++ and it takes an hour to compile and link our programs. You may as well be making your design decisions based on code that works. Otherwise, you can spend forever speculating about what the *right* way to code something might be.

PATTERN: FACTOR A SUPERCLASS

As an alternative to the Separate Abstract from Concrete pattern, I'd like to present the way Ward Cunningham taught me to make inheritance decisions. It is very much in keeping with what I wrote above about letting your "mistakes" teach you the "right" thing to do. When you are programming like this, it feels like the program itself is teaching you what to do as you go along.

CONTEXT

You have developed two classes which share some of the same methods. You have gotten tired of copying methods from one to the other, or you have noticed yourself updating methods in both in parallel.

PROBLEM

How can you factor classes into inheritance hierarchies that share the most code? (Note that some people will say that this isn't the problem that inheritance should be solving. You wouldn't use this pattern if that was your view of inheritance.)

CONSTRAINTS

You'd like to start using inheritance as soon as possible. If you're using inheritance you can often program faster because you aren't forever copying code from one class to another (what Sam Adams calls "rape and paste reuse"). Also, if you are using inheritance, you don't run the risk of a multiple update problem, where you have two identical methods, and you change one but not the other. Ideally, for this constraint, you'd like to design your inheritance hierarchy before you ever wrote a line of code.

On the other hand, designed inheritance hierarchies (as opposed to derived inheritance hierarchies) are seldom right. In fact, by making inheritance decisions too soon you can blind yourself to the opportunity to use inheritance in a much better way. This constraint suggests that you should make inheritance decisions only after the entire system is completed.

SOLUTION

If one of the objects has a superset of the other object's variables, make it the subclass. Otherwise, make a common superclass. Move all of the code and variables in common to the superclass and remove them from the subclasses.

EXAMPLE

It is difficult to come up with an example of inheritance that isn't totally obvious. The problem is that before you see it, you can't imagine it, and after you see it, you can't imagine it any other way. So, if this example seems contrived, don't worry, your own problems will be much harder.

Here is an example in VisualWorks I ran across a couple of months ago. I had Figure1, a subclass of VisualPart. It had to be dependent on a several other objects, and it had to delete those dependencies when it was released.

```
Class: Figure1
    Superclass: VisualPart
    Instance variables: dependees

Figure>>initialize
    dependees := OrderedCollection new
```

Rather than use the usual addDependent: way of setting up dependencies, I implemented a new message in Figure1 called dependOn:.

```
Figure1>>dependOn: anObject
    dependees add: anObject.
    anObject addDependent: self
```

When the figure goes away, it needs to detach itself from everyone it depends on.

```
Figure1>>breakDependents
    dependees do: [:each | each removeDependent: self].
    super breakDependents
```

Then I created a Figure2. To get it up and running quickly I just copied the three methods above to Figure2 and set about programming the rest of it.

It was when I went to create Figure3 that I decided to take a break and clean up. I created DependentFigure as a subclass of VisualPart, gave it the variable dependees and the three methods above, made Figure1 and Figure2 subclasses of it, deleted their implementations of initialize, dependOn: and breakDependents, and then implemented Figure3.

OTHER PATTERNS

While you are factoring the code is often a good time to apply Compose Methods so you can move more code into the superclass.

CONCLUSION

I have presented a pattern called Factor a Superclass as an alternative to Separate Abstract from Concrete for creating inheritance hierarchies. Using Factor a Superclass, you will end up with superclasses that have more state. I'm not sure if this is a good thing or not. On the plus side, you will probably be able to share more implementation. On the minus side, you may find yourself applying the pattern several times to get the final result. You might factor two classes to get a third, then notice that once you look at the world that way you can factor the superclass with a previously unrelated class to get a fourth, and so on.

Beware of juggling inheritance hierarchies too much. You can waste lots of time factoring code first one way, then another, and find that in the end you aren't that much better off than you were when you started. Objects can survive less-than-optimal inheritance much better than they can encapsulation violations or insufficient polymorphism. Most expert designers agree that great inheritance hierarchies are only revealed over time. Make the changes that you can see are obvious wins, but don't worry about getting it instantly, absolutely right. You are better off getting more objects into your system so you have more raw material from which to make decisions. ▣

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.

DO YOU KNOW SMALLTALK?

At Boole & Babbage, we talk big about our UNIX and mainframe products. If you want an unparalleled technical opportunity to work with a world-class team in a company with 25 years experience as an innovator, bring your Smalltalk and OOD skills and talk big to:

Boole & Babbage

Group Staffing DRRSR
510 Oakmead Parkway
Sunnyvale, CA 94086
FAX: (408) 737-2649
or email (ASCII and Postscript only):
info@boole.com

EOE
principals only

THE BEST OF comp.lang.smalltalk

Alan Knight

Extending the environment (part 1)

The Smalltalk development environment is excellent in many ways, but stagnant. The basic tools haven't changed much from when I first used Apple Smalltalk-80 on a Lisa in 1986. At that time Smalltalk and LISP systems led the way in interactive development environments. Now these environments exist for many languages, some of them very competitive with Smalltalk.

To be fair, there have been great improvements in some areas, mostly in the area of add-on products. These include GUI builders, team programming tools, profilers, and database interfaces. The basic tools—the browsers, inspectors and the debugger—remain almost unchanged. This is not because they defy improvement.

Fortunately, one of Smalltalk's strengths is the ease with which it can be customized and extended. In this column, the first of two parts, I'll discuss some simple extensions to these tools. Part two will look at some of the packages available that make more substantial changes. The main focus will be on ideas or on code available over the net rather than commercial products which are better covered in a product review.

AREN'T IMPROVEMENTS THE VENDOR'S JOB?

Ideally, users shouldn't have to write or acquire extended tools. The development environment is a strong selling point for Smalltalk, and one might expect the vendors to put some effort into improving it. From the vendor's point of view, however, there are good reasons not to change the environment.

- *Backward compatibility.* Everybody gets annoyed when system code changes. If users don't think the changes are worth breaking their code for, they'll be upset.
- *Disagreement.* Any vendor-imposed changes to the environment will be unpopular with some users, and questionable changes run the risk of a backlash rivaling that was received by the New Coke.
- *Priorities.* Vendors have limited resources, and are kept very busy developing new products and fixing the major problems with existing ones. The base environment isn't bleeding too badly, so resources go elsewhere.
- *Lack of competition.* With the recent growth in Smalltalk's popularity, many users are new to the language and come from areas such as mainframe COBOL or 4GL development. They're still too dazzled by the very idea of an incremental development environment to complain about its deficiencies. Competition from other languages isn't strong enough yet to inspire changes. The most likely source of improvements may be new Smalltalk vendors who need to worry more about carving a niche than backward compatibility.
- *Extensibility.* There are relatively few complaints about the environment, because any user with sufficient time and skill can change it to suit themselves.

IT'S UP TO YOU

You can't count on the vendors for improvements, so it's up to you to take responsibility for your own development environment. You don't have to rewrite the debugger, but don't be afraid to make changes or to explore the changes others have made.

At this point, careful readers may recall my March/April 1993 column, where I urged great caution in making system changes. This appears to be a contradiction, but it's really just a trade-off. To be sure, there are risks in changing the system. New releases or add-on products will need to be checked more carefully for conflicts and small mistakes can destroy an image. Frequent back-ups are in order.

On the other hand, changing the browsers or inspectors is much less risky than changing deep system components such as the compiler or the process scheduling mechanisms. Even with the risks, the increased productivity can be well worth the trouble. As always, it's best to limit changes in system methods to small "hooks" that call your own code. This helps minimize the problems with new releases.

WHAT NEEDS CHANGING

Development environments are a religious issue, and everyone has a different opinion on the perfect environment. Nevertheless, here's a short wish list of ideas. Note: Not all these ideas have been implemented, and if they have, the author is not necessarily in a position to distribute the code. The best place to look for code is the Smalltalk ftp archives (st.cs.uiuc.edu or mushroom.cs.man.ac.uk), where the authors have gone to the trouble of cleaning things up and releasing them to the public. Code written for personal use often requires significant effort to adapt and separate from other extensions.

This column mentions extensions from three different people on the net. **Deeptendu Majumder** (dips@cad.gatech.edu) has released his extensions up in a package called ISYSE, available from the archives.

Bruce Samuelson (bruce@ling.uta.edu) may get around to cleaning up and releasing his code, but is not in a position to do so at this time. **Gene Golovchinsky** (golovch@ie.toronto.edu) hasn't packaged his extensions, but is willing to be pestered about them.

Automatically writing access methods

One of the most common system extensions is a mechanism to generate access methods for instance variables. These methods aren't difficult to write by hand, but they occur so frequently that a tool can be very convenient.

It's important that the tool be selective. Not all variables should have access methods (or some of them should be clearly marked private, depending on your philosophy) so the user must be able to select which methods to generate. The tool should also provide documentation in the method. The user should be able to (if not forced to) provide information on the type of the variable and its purpose. This information should already be in the class comment, but it doesn't hurt to duplicate it. A really sophisticated tool would check the class comment for the information and update it if necessary.

Find class

I use the "Find class" feature very frequently, especially in Digitalk dialects. Unfortunately, the basic Digitalk implementation is brain-dead, and the ParcPlace one, while better, still doesn't do what I want.

- Ignore case. This is much faster and more convenient. (Is it Filename or FileName?)
- If the name matches a class (e.g., set), go directly to it without presenting a useless list of one class to choose from. In general, I prefer tools that can skip over lists with only one item.
- If the name doesn't match a class, append a wildcard and present a list of those it matches (e.g., sett gives me a list of #(Settee Setter Settlement).
- If I explicitly type a wildcard, always give me the list (e.g., set* gives #(Settee Setter Settlement)).

Smalltalk/V's debugger

If you've used both Smalltalk-80 and Smalltalk/V, one of the most frustrating things about V is its debugger. To the untrained eye, both debuggers are very similar, and in fact V offers the nice additional feature of breakpoints. The problem is that when evaluating an expression inside the debugger, V evaluates it as a method in **self** (the receiver of the current message), *not* the context of the current method. In the Smalltalk-80 debugger you can highlight any text in the current method and evaluate it. In the Smalltalk/V debugger this only works if the text doesn't reference method arguments or locals.

The most irritating thing about this problem is that I don't know how to fix it. Digitalk hides the source to their compiler, and although I've come up with a few bizarre ideas that might work, I've never had time to really work on it. If anybody has a fix for this, please let me know.

Browsing inherited methods

I don't know how many requests I've seen for a for a browser that shows all methods in a class, including inherited methods. The basic functionality is very simple, and the real problem is providing a good user interface. ParcPlace does provide this capability with the FullBrowser, but it's a poor implementation and only available in the APOK add-on package. It's a good example of why we might not want the vendors deciding for themselves how to improve the environment. Most of the extended environments described in part 2 provide this capability in some form.

Resizing panes

Bruce Samuelson describes a useful feature to augment the browser with:

> **...buttons for resizing browser windows horizontally and vertically, and reproportioning the line separating the upper panes from the method**

This is an increasingly common feature in user interfaces, and one that can be very useful. Smalltalk/V Mac has a convenient "zoom" feature that makes the text editing area fill the entire window, but this would be more flexible.

Gene Golovchinsky writes:

> **I would like to see more buttons on the screen for common commands rather than entries in pop-up menus. I invariably pick the wrong one, or keep moving between copy, paste, and accept. Then I accidentally pick cancel, and have to repeat the whole process again!**

I'm not sure we want to add too many buttons, but a few in the right place would be nice. Certainly, it's much nicer having buttons in the debugger for single stepping than having to use a pop-up menu. For operations like cut and paste I prefer to have keyboard short-cuts.

Renaming classes in Smalltalk/V

Smalltalk/V still doesn't support renaming classes or changing the definition of classes with instances. It shouldn't be that hard to implement, and I believe the capabilities are available as part of their Team/V package. Why is such a basic capability bundled into a team programming tool and not in the base image? Only Digitalk can tell.

COGNITIVE OVERLOAD

While all of the above are useful, they are only minor improvements. There are more general issues that need to be addressed. **Deeptendu Majumder** raises the issue of cognitive overload in the Smalltalk environment:

One thing that irritates me more and more these days is how my screen gets out of control with a multitude of windows.... I sometimes wonder if there is some kind of study...about determining the most suitable ST programming environment.... I sometimes *very strongly* feel the environment can be "smarter" about...reducing the cognitive overload *and* maintaining easily identifiable cues about what info is available only for a mouse click.

Controlling windows

The largest single factor in cognitive overload must the number of windows Smalltalk produces. I usually have 10 to 20 windows open simultaneously and I'm sure I get as high as 50 now and then. With this many windows, it's vital to have mechanisms to control the complexity.

Craig Latta (latta@xcf.berkeley.edu) writes:

I find that simply having a good window manager goes a long way toward reducing the cognitive load. The main problem I would have otherwise is with hordes of windows crowding the screen, and subsequently losing track of particular windows. Things like icon managers (as in 'twm' on X platforms) reduce this problem significantly.

A good window manager and a large screen are vital elements for Smalltalk work. One technique I use is to make use of window and icon positions. Certain windows (e.g., the system transcript, a workspace with useful expressions, my list of things to do) are always open, and I make a point of always keeping them in the same place. I also try to keep their icons in standard places, but not all window managers maintain the position of icons (MS-Windows doesn't).

"Writing Smalltalk code is akin to authoring hypertext"

Another technique is to put more information into window titles. By hooking into the browser selection mechanism, the window title can be made to indicate the current class and method. This makes navigating among icons easier, and can also be used with window managers that allow you to find windows by title. With a bit more effort, it should be possible to change the window icon to convey more information.

If your window manager doesn't manage windows and icons well, it's possible to make up some of the difference in Smalltalk. Gene Golovchinsky writes:

I added an entry to the Launcher menu that displays a list of all current Smalltalk windows, and indicates the minimized ones. If I pick from this menu, it raises that window. Just today I saw that something similar is available in the archives!

Reducing the number of windows

Managing windows is all very well and good, but do we really need all those windows in the first place? Jaap Vermeulen (jaap@sequent.com) doesn't think so. He writes:

With new tools to replace the browsers that allow better indexing, searching, shortcuts, and backtracking, you might need fewer windows. Finally, if the inspectors and debugger would become a little smarter and not throw up windows all over the place, we really would start talking.

Inspectors are one of the worst culprits in creating excess windows. A tool that allowed graphical inspecting of many objects at once, following links between them, could reduce this considerably. There is a simple tool of this type included with the HotDraw application framework. I believe First Class Software (408.338.4649 (voice), 408.338.3666 (fax), or 70761.1216 on CompuServe) has a graphical inspecting tool for Smalltalk/V.

Too many browser operations spawn a new window in which to present their results. The only concept of backtracking is to go back to the window you started the operation from. For operations like **senders**, this is simple to change and makes the function easier to use. Gene Golovchinsky writes:

I've augmented the MethodListBrowser to add the ability to add a specific method to the list. It works like the Messages menu item, but instead of spawning a new window, it adds the entry to the list. If there is more than one item, it prompts for the one to add. I find this tool handy for traversing long chains of message sends and keeping them all in one place.

Unfortunately, it's not so easy to reduce the number of windows generated by some of the other operations.

HYPERTEXT MECHANISMS

Gene Golovchinsky writes:

Writing Smalltalk code is akin to authoring hypertext; perhaps some insight can be gained from perusing that literature. Along those lines, this environment seems like an ideal vehicle for implementing all sorts of hypertext behavior. In fact, the existing browsers have many of these features already.

Indeed, Smalltalk browsing shares many characteristics with hypertext browsing and suffers many of the same problems. There's an enormous amount of information, only a small part of which is relevant at any given time, and it's easy to become lost in the irrelevant.

Messages

Many browser improvements are intended to quickly find relevant information while avoiding that which is not relevant. If you can follow a link directly to what's important, you don't need as many windows open looking for it.

One such feature is the messages menu item mentioned above. This allows you, when browsing a method, to find implementors or senders of any of the messages sent by that method. The messages sent become hypertext links.

One problem is that the number of methods found can be too large to work with. Thus, it's useful to restrict the methods considered. One way is to allow "local" senders/implementors, selecting only methods within the current class or perhaps within its sub/superclasses.

Bruce Samuelson has another mechanism:

> **...'my senders', 'my implementors' which only look at the changes file...**

Also, we may want to browse a method that isn't sent from the current message, or we may be in a text editor instead of a browser. **Gene Golovchinksy** describes a menu item that opens a browser on the class or method named by the currently selected text. I have a similar extension, but I separate the browse/senders/implementors/class references behavior and use keyboard shortcuts to invoke them. Keyboard shortcuts are a little faster, and work in workspaces as well as browsers, but are less mnemonic and not as flexible.

Operating on text is a nice feature, but one that works best for zero- or one-argument messages. Multi-keyword messages don't usually occur in text in the right form. It should be possible to use the Smalltalk parser to extract possible message names, but I haven't tried this.

Deeptendu Majumder added a feature for finding implementors of a method whose name is not known. The base image allows wildcard searches on method names, but force a choice from a menu of possible names without seeing implementations.

> **...all I did was add an extra list to the browser that grabs all those things that otherwise show up in the menu. When I am not sure exactly which method I am looking for, I can select entries from this list one after another and browse their various implementations. I can then change the selection template from within the list and grab a whole new set of message names.**

Searching for strings

The link you need may not be the name of the method or a message that it sends. Just today I wanted to search for a method that didn't send a particular message, but contained the name of that message in a comment. I had previously commented out that message send, closed the window, and forgotten the method name. **Bruce Samuelson** writes of a feature he implemented:

> **...search for a string (e.g., open:) in methods and class comments. This can operate on...categories, classes, or protocols. This is useful for maintaining comments and for finding code for which standard searches break down.**

Lost in hypertext tools

All the mechanisms listed above are valuable tools for search-

EC-Charts

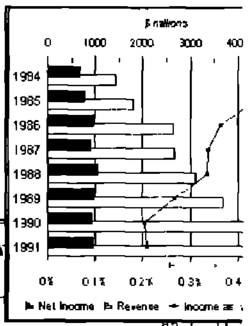
Just touch a button to put a chart view in your window!

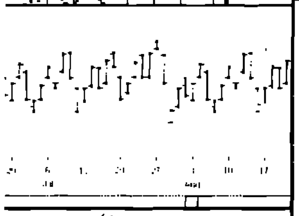
Add charts to your VisualWorks palette

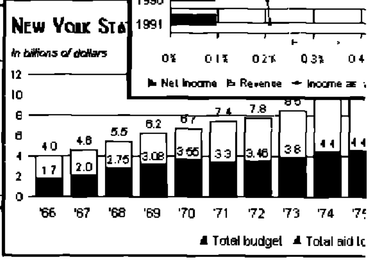
Dynamic Add or change data points, with minimal screen repainting. Add or remove data series to/from the chart.

Interactive Select data points with the mouse—EC-Charts informs your application.

Uses screen space effectively Scroll the chart view in one or both directions. Mark values of summary functions in the axis areas. Show thresholds using grid lines.







Now only **$350**

No runtime license fee

Call for a technical paper on EC-Charts

East Cliff Software

(408) 462-0641

21137 East Cliff Dr · Santa Cruz · CA 95062

VisualWorks is a trademark of ParcPlace Systems, Inc.

ing. Unfortunately, if we implemented them all in a single image I suspect users would merely find themselves lost in hypertext mechanisms instead of (or as well as) lost in the code. As **Deeptendu Majumder** writes:

> **There are so many small enhancements that can be done that I found it is not very productive to undertake the effort without a serious study of overall needs rather than trying to attack small segments of the problem.**

Next month, we'll examine some more radical extensions that replace the basic tools instead of patching or adding a few features.

ERRATA

In the June 1993 column I published code for testing dictionary performance under ObjectWorks\Smalltalk release 4.0. Unfortunately, I didn't test this code adequately, and Bruce Samuelson, the author, has pointed out that, due to changes, this code does not work with release 4.1 or VisualWorks. There are two problems. First, the way hashing is done has changed, so the results will be in error. Second, the method sortedElements has been removed, so the method will produce a walkback. A new version, which will also work with other hash table classes, is available from the Smalltalk archives at st.cs.uiuc.edu. ▣

Alan Knight works for The Object People. He can be reached at 613.225.8812 or by email as knight@mrco.carleton.ca.

continued from page 6

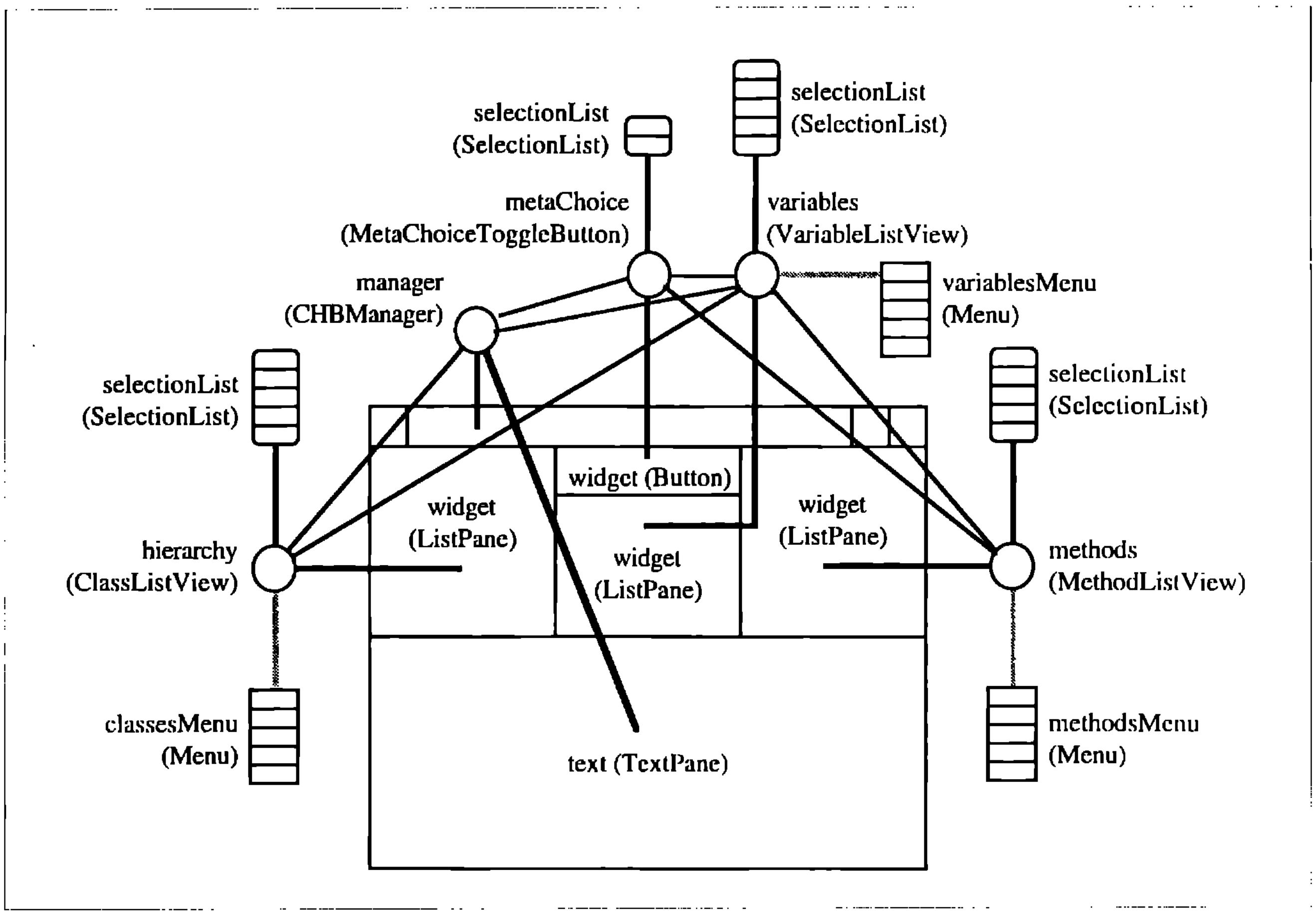

selectionList (SelectionList)
selectionList (SelectionList)
metaChoice (MetaChoiceToggleButton)
variables (VariableListView)
manager (CHBManager)
variablesMenu (Menu)
selectionList (SelectionList)
selectionList (SelectionList)
widget (Button)
widget (ListPane)
widget (ListPane)
widget (ListPane)
hierarchy (ClassListView)
methods (MethodListView)
classesMenu (Menu)
methodsMenu (Menu)
text (TextPane)

Figure 2. CHB frameworks.

dered dictionary key. SelectionLists also notify their dependent mediators when their list or selection changes:

```
"from within #list:"
self notifyThat: #listChanged.

"from within #select:"
self notifyThat: #selectionChanged.
```

Listing 5 shows the code for the SubPaneMediator classes. The kinds of SubPaneMediators that use SelectionLists include those depicted in the following hierarchy:

```
Object
  Framework
    SubPaneMediator
       ListItemChooser
          ListViewer
          ListButton
             MenuButton
             ToggleButton
```

The ListItemChooser class manages the interactions between a SelectionList and a SubPane (GUI widget). The ListViewer class manages the interactions between a SelectionList and a ListPane. The ListButton classes manage the interactions between a SelectionList and a Button in two varieties. The MenuButton class pops up a menu of the list items when clicked, allowing one of the items to be selected. The ToggleButton cycles through the list of items, showing the next item description on the button face.

Now, consider how these small framework classes might be used to refactor a browser such as the Smalltalk/V ClassHierarchyBrowser (CHB). The CHB has five subpanes: a class hierarchy ListPane, a variables ListPane, a methods ListPane, a RadioButton group, and a TextPane.

For this discussion, we will replace the RadioButton group with a specialization of the ToggleButton. This MetaChoiceToggleButton framework will use a two item list: #(class instance) for selecting either class methods or instance methods.

For each of the ListPanes, we specialize the ListViewer framework with ClassListViewer, VariableListViewer, and MethodListViewer frameworks. Each of these small frameworks serves as the owner for their respective subpanes. As such, they accrete the behavior from the CHB related to those panes, including menus, list maintenance, item selection, and propagation of notifications and changes throughout the overall framework network (see Figures 1 and 2).

This brief outline indicates how such refactoring can proceed. However, note that further evolution and improvements can be made through additional refactoring and framework creation. In the end, the responsibility of the browser class can be reduced to assembling a network of objects that together produce the overall browser behavior.

TUNING COMPONENT COUPLING

The Framework superclass uses loose coupling as a technique for achieving component integration and coordination. The implementation suggested in this article makes use of a kind of Dictionary to bind framework participants into their roles. This technique of loose binding allows frameworks to be evolved and extended quickly through several iterations.

Although this technique requires little in the way of overhead, a small amount of performance can be lost when the role participants are resolved dynamically. A number of options exist for tuning the performance of frameworks built using these techniques.

The Framework class uses a class named SmartDictionary (see Listing 1). In addition to the messages understood by IdentityDictionary, SmartDictionary responds to the typical accessor idioms:

```
componentName "getter"

componentName: anObject "setter"
```

These protocols are supported by overriding the #respondsTo: and #doesNotUnderstand: methods. These protocols are also supported by the Framework class. In addition to this implicit form of component access, the Framework class supports the following form of indirect access:

```
componentName "indirect getter"
  ^self partnerNamed: #componentName!

componentName: anObject "indirect setter"
  self
      for: #componentName
      use: anObject.!
```

This support for the dynamic binding of roles can be replaced by ordinary instance variables and their accessors. However, in order to gain the benefits of rapid design evolution, this should be done (if done at all) only after the design of the framework class has stabilized.

```
componentName "direct getter"
  ^componentName!

componentName: anObject "direct setter"
  componentName := anObject.!
```

PARTICIPANT INTERACTION STYLES

One of the principal uses of any framework class is to mediate the interactions of its participants. Because participants are loosely coupled, the methods of a framework class have this peculiar aspect: Participants are *always* accessed through requests to self. So, some of the framework methods provide access to components or their state(s), while others translate events into actions.

The event handling methods of a framework class serve as templates that guide the exchange of information between the framework participants. The expressions used by these event handling methods generally fall into one of the following basic patterns:

```
eventName
"Request information or a change of state."
  ^self someComponent request

eventName
"Exchange information between components."
  self someComponent binaryKeyword:
        self anotherComponent request.

eventName
"Notify another participant (framework) that something happened
(translating the event name)."
  ^self
        notify: #frameworkX
        that: #somethingHappened

eventName
"Forward this event to another participant (framework)."
  ^self
        notify: #frameworkX
        that: #eventName
```

SPECIALIZING FRAMEWORKS

New frameworks can often be discovered when reusing existing ones. Sometimes it is more convenient to attach custom behavior to an existing framework rather than create a new framework subclass.

The Framework superclass supports the prototyping of new behavior by allowing the usage of blocks as components. When a message is redirected through #doesNotUnderstand:, the Framework superclass checks to see if a block has been defined to handle the message selector. If the framework can handle the message with a block, the block is evaluated with the message receiver and its arguments (if any).

After a new framework has stabilized, the developer may decide to create a new framework subclass, moving its specialized behavior from blocks into methods. When this occurs, the developer is faced with a decision: What should the scope of visibility for the new class be? Very general frameworks should probably be visible to the whole Smalltalk system. However, some frameworks should only be visible to the class(es) that need them. Module classes[13] can be used to hide specialized framework subclasses.

For example, in our consideration regarding browsers, we found that they will often need specialized frameworks for managing the interactions between the subpanes from which they are composed. Each of the ListItemChooser subclasses can be further specialized to create customized mediators that manage the overall interactions between the various subpanes that make up a browser. Rather than expose these specialized frameworks to the whole of Smalltalk, they can be hidden within the browser class if it is implemented as a module.

GENERAL OBSERVATIONS

Many patterns of interaction between objects in a system appear over and over again in other systems. Sometimes these patterns are formed into a loose composition of abstract classes like the MVC framework.[1] Following the flow of messages through such an implicit "second-class" framework can be difficult. However, these patterns of interaction can be captured and reused explicitly by framework classes. Because the message flow is more explicit in framework classes, they are much easier to understand.

As noted perviously, good class hierarchies tend to be deep and narrow. The hierarchies created by framework classes tend to be deep, narrow, and *thin*. The methods themselves tend to be small (thin), because they coordinate only the interactions between the objects that participate in the framework.

Many object designers have claimed that frameworks are difficult to find. Actually, frameworks are not hard to find at all! They simply have not been noticed much. They tend to be like thin oils that lubricate the meshings of larger objects. Any pattern of interactions between objects may be captured as a framework. However, the resulting framework may be so specialized that it is better to leave the interactions built into the collaborating classes. Frameworks serve best when they capture and factor out the semantics of event-driven interactive systems.

Sometimes it is expedient during prototyping to develop a system that is closely coupled. After completing the prototype, some parts of the design can be revisited and the coupling loosened for better reusability. Loosely coupled objects tend to be more reusable and more resilient to design and system evolution. Framework classes provide a new option for refactoring through decoupling.

FUTURE WORK

The current implementation of the Framework superclass uses a simple collection of method names for role validation. It would be better if each role were defined using a specification object, in particular an object type. Object types use method signatures to specify the types of each argument and the method result. When these specification objects become available, framework role validation can evolve to use them. Object types will provide better constraints to qualify components for roles.

CONCLUSION

This article has presented a new view of object frameworks: How framework classes can simplify the design of component classes by factoring out the behavior found in interactive systems. Component objects become simply clients and/or service providers, reducing or eliminating the additional responsibilities of complex coordination between objects. In addition to simplifying existing components, refactoring may create new components. Such refactoring improves the reusability of all the components that form a system and creates reusable framework objects. ■

Acknowledgments

Several individuals inspired me with their interest and thoughtful critiques during the evolution of these ideas. Special thanks to Jean-Francois Cloutier, Tracy Tondro, Oleg Arsky, and Jim Carlstedt.

References

1. Krasner, G.E., and S.T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80, JOURNAL OF OBJECT-ORIENTED PROGRAMMING 1(3):26–49, 1988.
2. Shan, Y-P. An event-driven model-view-controller framework for Smalltalk, OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, ACM, New Orleans, LA, 1989.
3. Shan, Y-P. MoDE: A UIMS for Smalltalk,. OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, ACM, Ottawa, ONT, 1990.
4. Sullivan, K.J., and D. Notkin. Reconciling environment integration and component independence, TRANSACTIONS ON SOFTWARE ENGINEERING, ACM, Ottawa, ONT, 1990.
5. Helm, R., I.M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems, OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, ACM, Ottawa, ONT, 1990.
6. Wilkerson, B. How to design an object-based application, DEVELOP, Apple Computer, Cupertino, CA, April, 1990.
7. Wirfs-Brock, R., and R.E. Johnson. A survey of current research in object-oriented design. COMMUNICATIONS OF THE ACM 33(9):104–124, 1990.
8. Wirfs-Brock, R., and B. Wilkerson. Object-oriented design: A responsibility-based approach, OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, ACM, New Orleans, LA, 1989.
9. Wirfs-Brock, R., B. Wilkerson, L. Wiener. DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice Hall, Englewood Cliffs, NJ, 1990.
10. Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. OBJECT-ORIENTED MODELING AND DESIGN, Prentice Hall, Englewood Cliffs, NJ, 1991.
11. Opdyke, W.F. Refactoring object-oriented frameworks, PhD thesis, University of Illinois at Urbana-Champaign, 1992.
12. Cook, W.R. interfaces and specifications for the Smalltalk-80 collection classes, OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, ACM, Vancouver, BC, 1992.
13. Boyd, N. Modules: Encapsulating behavior in Smalltalk, THE SMALLTALK REPORT 2(5), 1993.

Nik Boyd is a Principal Member of the Technical Staff at Citicorp Transaction Technology in Santa Monica, CA. His research interests include instance-centered and class-centered object systems, as well as tools and techniques that support object-oriented software engineering. Nik may be contacted via email at 74170.2171@compuserve.com or through the American Information Exchange (AMIX).

Listing 1. Framework Support.

```
"The following code extends the baseline Smalltalk classes to support
certain aspects of framework assembly, event handling, and role
validation."

!Collection methods !

assembleAs: frameworkClass
"Answer a new framework assembled from the receiver."
  | framework |
  framework := frameworkClass new.
  self do: [ :each | view useBestRoleFor: each ].
  ^framework resolveRoles! !

!Object methodsFor: 'accessing named dependents' !

dependentNamed: name
"Answer the named dependent, or nil."
  ^self dependentNamed: name ifNone: [ nil ]!

dependentNamed: name ifNone: aBlock
"Answer the named dependent, or evaluate aBlock."
  ^self namedDependents
        detect: [ :d | d name = name ] ifAbsent: aBlock!

namedDependents
"Answer any named dependents attached to the receiver."
  ^self dependents
        select: [ :each | each respondsTo: #name ]! !

!Object methodsFor: 'performing optional behaviors' !

ifUnderstood: selector do: aBlock
"Evaluate aBlock if the receiver understands selector."
  ^(self respondsTo: selector)
        ifTrue: aBlock
        ifFalse: [ self ]!

ifUnderstoodPerform: selector
"Answer the result of the selected method, or the receiver."
  (self respondsTo: selector) ifFalse: [ ^self ].
  ^self perform: selector!

ifUnderstoodPerform: selector with: argument
"Answer the result of the selected method, or the receiver."
  (self respondsTo: selector) ifFalse: [ ^self ].
  ^self perform: selector with: argument!

ifUnderstoodPerform: selector withAll: arguments
"Answer the result of the selected method, or the receiver."
  (self respondsTo: selector) ifFalse: [ ^self ].
  ^self perform: selector withArguments: arguments! !

!Object methodsFor: 'notifying dependents of events' !

notify: name that: eventName
  ^(self dependentNamed: name)
        notifyThat: eventName!

notify: name that: eventName with: argument
  ^(self dependentNamed: name)
        notifyThat: eventName
        with: argument!

notify: name that: eventName withAll: arguments
  ^(self dependentNamed: name)
        notifyThat: eventName
        withAll: arguments!

notifyThat: eventName
"Answer the final result of notifying all the dependents that eventName
occurred."
  | result |
  self dependents do: [ :d |
        result := d notifyThat: eventName ].
  ^result!

notifyThat: eventName with: argument
"Answer the final result of notifying all the dependents that eventName
occurred."
  | result |
  self dependents do: [ :d |
        result := d notifyThat: eventName
                    with:argument ].
  ^result!

notifyThat: eventName withAll: arguments
"Answer the final result of notifying all the dependents that eventName
occurred."
  | result |
  self dependents do: [ :d |
        result := d notifyThat: eventName
                    withAll: arguments ].
  ^result! !

!Object methodsFor: 'responding to requests' !

respondsToAll: symbolSet
"Answer whether the receiver responds to all of the messages in
symbolSet."
  symbolSet do: [ :each |
        (self respondsTo: each) ifFalse: [ ^false ] ].
  ^true!

servicesRejectedFrom: symbolSet
"Answer those service requests from symbolSet to which the receiver
does not respond."
  ^symbolSet reject: [ :each | self respondsTo: each ]!

value
"Answer the receiver."
  ^self! !

!UndefinedObject methodsFor: 'catching dependents access' !

namedDependents
"nil has no dependents."
  ^Array new!
```

```
notifyThat: eventName
"Do nothing, as nil has no dependents."!

notifyThat: eventName with: argument
"Do nothing, as nil has no dependents."!

notifyThat: eventName withAll: arguments
"Do nothing, as nil has no dependents."! !

IdentityDictionary subclass: #SmartDictionary
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!SmartDictionary methods !

respondsTo: selector
"Answer whether the receiver can respond to the message selector."
  | colons |
  (super respondsTo: selector) ifTrue: [ ^true ].
  (self includesKey: selector) ifTrue: [ ^true ].
  colons := selector occurrencesOf: ($:).
    colons = 1

doesNotUnderstand: aMessage
"If the receiver can handle aMessage selector, do so. Otherwise, treat
aMessage like super would."
  | name |
  name := aMessage selector.
  (self respondsTo: name) ifFalse: [
        ^super doesNotUnderstand: aMessage. ].
  "handle getter."
  (self includesKey: name) ifTrue: [ ^self at: name ].
  "handle setter."
  name := name asString copyWithout: ($:).
  ^self at: name asSymbol
        put: aMessage arguments first.! !
```

Listing 2: Framework class.

```
Object subclass: #Framework
  instanceVariableNames: 'name parts '
  classVariableNames: ''
  poolDictionaries: '' !

!Framework class methodsFor: 'creating instances' !

assemble: frameworkName from: parts
  ^self new
        name: frameworkName;
        parts: parts;
        resolveRoles!

new
  ^super new initialize! !

!Framework methodsFor: 'initializing - releasing' !

initialize
  name := nil.
  parts := SmartDictionary new.!

release
  | objects |
  objects := self parts.
  parts := SmartDictionary new.
  objects do: [ :each | each release ].
  ^super release! !

!Framework methodsFor: 'accessing components' !

name
  ^name!

name: partName
  name := partName.!

partNamed: partName
  ^self partNamed: partName ifNone: [ nil ]!

partNamed: partName ifNone: aBlock
  | part |
  part := parts at: partName ifAbsent: [ ^aBlock value ].
  part isNil ifTrue: [ ^aBlock value ].
  ^part!

partNames
  ^parts keys!

parts
  ^parts values!

parts: partsCatalog
  parts := partsCatalog.! !

!Framework methodsFor: 'assembling frameworks' !

bestRoleNameFor: part
"Answer the roleName that best fits the part, or nil."
  | roleName roleSize roleServices |
  roleSize := 0.
  roleName := nil.
  self class roleNames do: [ :each |
      self class ifUnderstood: each do: [
            roleServices := self class perform: each.
            roleServices size > roleSize ifTrue: [
                  (self canUse: part as: each) ifTrue: [
                        roleName := each.
                        roleSize := roleServices size ] ] ] ].
  ^roleName!

useBestRoleFor: part
  | roleName |
  roleName := self bestRoleNameFor: part.
  roleName isNil ifFalse: [
        self for: roleName use: part ].! !
```

```
!Framework methodsFor: 'defining roles' !

addRolesNamed: roleNames
  roleNames do: [ :roleName | self for: roleName use: nil ].!

for: roleName use: anObject
  (self binders includes: roleName) ifFalse: [
        ^parts at: roleName put: anObject ].
  self
        perform: (self binderFor: roleName)
        with: anObject.!

when: eventName do: aBlock
  self for: eventName use: aBlock.! !

!Framework methodsFor: 'binding components' !

resolveRoles
  | framework |
  parts associationsDo: [ :model |
        framework := model value.
        (framework isKindOf: Framework) ifTrue: [
                    framework name: model key.
                    framework resolveRolesFrom: parts ] ].
  self validateParts.!

resolveRolesFrom: partsCatalog
  | part |
  self unresolvedRoleNames do: [ :roleName |
        part := partsCatalog at: roleName ifAbsent: [ nil ].
        self for: roleName use: part ].
  self validateParts.!

unresolvedRoleNames
  ^parts keys select: [ :roleName |
        (self partNamed: roleName) isNil ]! !

!Framework methodsFor: 'triggering events' !

notify: partName that: eventName
"Answer the result of notifying the named part that eventName
occurred."
  ^(self partNamed: partName)
        notifyThat: eventName!

notify: partName that: eventName with: argument
"Answer the result of notifying the named part that eventName
occurred."
  ^(self partNamed: partName)
        notifyThat: eventName
        with: argument!

notify: partName that: eventName withAll: arguments
"Answer the result of notifying the named part that eventName
occurred."
  ^(self partNamed: partName)
        notifyThat: eventName
        withAll: arguments! !

!Framework methodsFor: 'translating events to messages' !

respondsTo: selector
  (super respondsTo: selector) ifTrue: [ ^true ].
  (parts respondsTo: selector) ifTrue: [ ^true ].
  ^false!

notifyThat: eventName
"Answer the result of performing eventName, or the receiver if
eventName has not been implemented."
  ^self ifUnderstoodPerform: eventName!

notifyThat: eventName with: argument
"Answer the result of performing eventName, or the receiver if
eventName has not been implemented."
  ^self ifUnderstoodPerform: eventName with: argument!

notifyThat: eventName withAll: arguments
"Answer the result of performing eventName, or the receiver if
eventName has not been implemented."
  ^self ifUnderstoodPerform: eventName
        withAll: arguments! !

!Framework methodsFor: 'validating role services' !

canUse: part as: roleName
  self class ifUnderstood: roleName do: [
        ^part respondsToAll:
                    (self class perform: roleName) ].
  ^true!

validate: part as: roleName
  | services |
  (self canUse: part as: roleName) ifTrue: [ ^self ].
  services := self class perform: roleName.
  services := part servicesRejectedFrom: services.
  ^self error:
        'Supplied ', roleName storeString,
        ' cant respond to ', services first storeString!

validateParts
  parts associationsDo: [ :each |
        self validate: each value as: each key ].! !

!Framework methodsFor: 'binding components - private' !

binderFor: roleName
"Answer the selector that can be used to bind a component to
roleName."
  ^( roleName, ':' ) asSymbol!

binders
"Answer all the selectors that can be used to bind the components of a
framework subclass."
  | supers methodNames |
  methodNames := Set new.
  supers := self class allSuperclasses removeLast; yourself.
  supers size > 0 ifTrue: [ supers removeLast ].
  supers do: [ :s |
        methodNames addAll: s methodDictionary keys ].
  methodNames := methodNames select: [ :n |
        n last == ($:) ].
  methodNames := methodNames collect: [ :n |
        n copyWithout: ($:) ].
  ^methodNames!

doesNotUnderstand: aMessage
"Try handling aMessage, assuming it is accessing the parts of the
receiver. If the part accessed is a block context, answer the result of
evaluating the block with the receiver and arguments from aMessage as
arguments. Otherwise, answer the accessed part. If aMessage does not
access a part, let the superclass handle aMessage."
  | part |
  (parts respondsTo: aMessage selector) ifFalse: [
        ^super doesNotUnderstand: aMessage ].
  part := self partNamed: aMessage selector ifNone: [
        ^parts perform: aMessage selector
                    withArguments: aMessage arguments ].
  part isContext ifTrue: [
        aMessage arguments isEmpty ifTrue: [
                    ^part value: aMessage receiver ].
        ^part value: aMessage receiver
                    value: aMessage arguments ].
  ^part! !
```

Listing 3: SubjectView Contract.

```
"The following example is derived from the contract SubjectView
described on page 171 of [HHG90]."

!SubjectView class methodsFor: 'validating roles' !

roleNames
  ^ #( subject view )!

subject
  ^ #( value value: )!

view
  ^ #( showValue: )! !

!SubjectView methodsFor: 'supporting subject' !

setValue: value
  self getValue = value ifTrue: [ ^self ].
  self subject value: value.
  self notify.!

getValue
  ^self subject value!

notify
  self views do: [ :view | self update: view ].!

attachView: aView
  self validate: aView as: #view.
  self views add: aView.!

detachView: aView
  self views remove: aView.! !

!SubjectView methodsFor: 'supporting views' !

update: aView
  self draw: aView.!

draw: aView
  aView showValue: self getValue.!

setSubject: aSubject
  self validate: aSubject as: #subject.
  self subject: aSubject.
  self views == self ifTrue: [ self views: Set new ].! !

"sample use of the framework"
SubjectView new
  setSubject: ValueHolder new;
  attachView: BarGraphView new;
  attachView: DialGaugeView new;
  attachView: PercentageView new;
  setValue: 75.!
```

Listing 4: ButtonGroup Contract

```
"The following example is derived from the refinement of the
SubjectView contract called ButtonGroup on page 173 of [HHG90]."

SubjectView subclass: #ButtonGroup
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''!

!ButtonGroup class methodsFor: 'validating roles'

view
  ^ #( value chosen: )! !

!ButtonGroup methodsFor: 'supporting buttons' !

select: aButton
  self setValue: aButton value.!

update: aButton
  self getValue = aButton value
        ifTrue: [ self choose: aButton ]
        ifFalse: [ self unChoose: aButton ].!

choose: aButton
  aButton chosen: true.!

unChoose: aButton
  aButton chosen: false.! !
```

Listing 5: SubPaneMediators

```
Framework subclass: #SubPaneMediator
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !
```

```
!SubPaneMediator methodsFor: 'binding components' !

supportedEventHandlers
  ^ #(
        clicked:
        doubleClickSelect:
        getContents:
        getMenu:
        getPopupMenu:
        select:
  )!

handlerFor: event
  ^self supportedEventHandlers detect:
        [ :evh | event = (evh copyWithout: ($:) ) ]
        ifNone: [ nil ]!

support: event for: subPane
  | selector |
  selector := self handlerFor: event.
  selector isNil ifTrue: [ ^self ].
  subPane when: event perform: selector.!

supportEventsFor: subPane
  subPane class supportedEvents do: [ :event |
        self support: event for: subPane ].!

claimOwnershipOf: subPane
  subPane ifUnderstood: #supportedEvents do: [
        subPane owner: self.
        self supportEventsFor: subPane ].!

for: partName use: anObject
  | selector |
  super for: partName use: anObject.
  self claimOwnershipOf: anObject. "if SubPane"! !

!SubPaneMediator methodsFor: 'handling events' !

clicked: subPane
  self notifyThat: #clicked.!

doubleClickSelect: subPane
  self notifyThat: #doubleClicked.!

getContents: subPane
  self ifUnderstood: #getContents do: [
        subPane contents: self getContents ].!

getMenu: subPane
  self ifUnderstood: #getMenu do: [
        subPane setMenu: self getMenu ].!

getPopupMenu: subPane
  self ifUnderstood: #getPopupMenu do: [
        subPane setPopupMenu: self getPopupMenu ].!

select: subPane
  self notifyThat: #selected.! !
```

```
SubPaneMediator subclass: #ListItemChooser
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!ListItemChooser class methodsFor: 'validating roles' !

roleNames
  ^ #( selectionList widget )!

selectionList
  ^ #(
        list:
        items
        selections
        selectIndex:
        selectedIndex
        selectItem:
        selectedItem
        select:
        selection
  )! !

!ListItemChooser methodsFor: 'accessing component states' !

getContents
  ^self listItems!

listItems
  ^self selectionList items!

listSelections
  ^self selectionList selections!

selectedIndex
  ^self selectionList selectedIndex!

selectedItem
  ^self selectionList selectedItem!

selection
  ^self selectionList selection!

selectionList
  ^self partnerNamed: #selectionList!

widget
  ^self partnerNamed: #widget! !

!ListItemChooser methodsFor: 'changing component state' !

changeList
  self selectionList list: self getList.
  "note: getList should be implemented by subclass
  method or prototype block"!

selectIndex: index
  self selectionList selectIndex: index.!
```

```
selectItem: item
  self selectionList selectItem: item.!

selectObject: selection
  self selectionList select: selection.! !

ListItemChooser subclass: #ListViewer
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!ListViewer class methodsFor: 'validating roles' !

widget
  ^ #(
        deselect
        restoreWithRefresh:
        selection:
        selection
  )! !

!ListViewer methodsFor: 'translating events' !

deselected
  self widget deselect.!

listChanged
  self widget restoreWithRefresh: self selectedItem.!

selected
  self selectIndex: self widget selection.! !

!ListViewer methodsFor: 'changing component state' !

showSelection
  self widget selection: self selectedIndex.! !

ListItemChooser subclass: #ListButton
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!ListButton class methodFor: 'validating roles' !

widget
  ^super widget, #( contents: )! !

!ListButton methodsFor: 'translating events' !

clicked
  self listSelections size > 1 ifTrue: [
        self widget contents: self nextSelection ].!

listChanged
  self selectionChanged.!

selectionChanged
  self widget contents: self selectedItem.! !

ListButton subclass: #MenuButton
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!MenuButton class methodsFor: 'validating roles' !

selectionList
  ^super selectionList, #( popUpItems )! !

!MenuButton methodsFor: 'changing component state' !

nextSelection
  ^self selectionList popUpItems! !

ListButton subclass: #ToggleButton
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!ToggleButton class methodsFor: 'validating roles' !

selectionList
  ^super selectionList, #( selectNext )! !

!ToggleButton methodsFor: 'changing component state' !

nextSelection
  ^self selectionList selectNext! !
```

Greg Hendley & Eric Smith

Keeping multiple views up-to-date

In many Smalltalk applications, it is possible for the end-user to have several independent windows providing views of the same information (Figure 1). There may be several instances of one kind of window or parent and child windows that, though very different in appearance, share some overlap in the information they present. To prevent inconsistencies between windows, the changed-update (also known as *Object Dependents*) mechanism can be used to insure that all views the end-user has opened on a particular object are kept up-to-date with that object's most recent idea of what it looks like.

For example, if the user has a view of a Customer object that he opened directly and another view of the same Customer that was opened as a consequence of browsing a ServiceAgreement object, any changes make to one view of the Customer should be immediately reflected in the other. The user should not see two different views and be left to figure out which one is the most current.

BACKGROUND

The application architecture outlined in previous columns (The Smalltalk Report, May 1992 and October 1992) will be employed here. For those who have not yet been exposed to the Interface-Control-Model architecture, a brief glossary of terms is provided here.

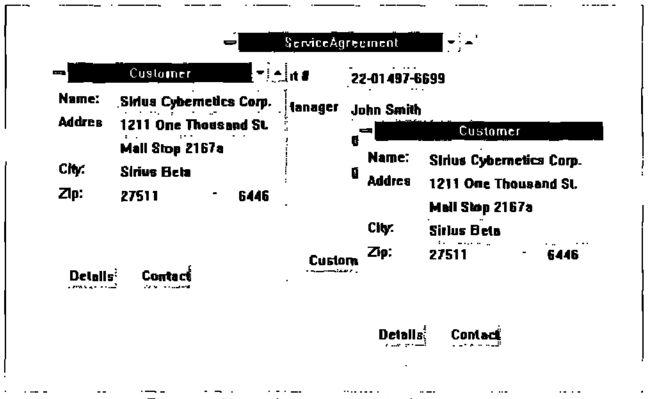
- *Interface.* The component of the user interface whose job it is to present information to the end-user and accept input events from same. The interface translates user input to semantic actions such as mouse-clicks to selection or menu selections to commands. The interface has very little knowledge of the structure of the application of which it is a part. It has virtually no knowledge of the domain model (see below).
- *Control.* The control layer of an application is the component that understands the semantics of the application as a whole. This is where commands identified by the interface are actually carried out. The application control understands the relationships among the various domain model objects it works with. It also knows about the consequences of commands. This is the point where all the "brains" of the application (as the end-user sees it) reside.
- *Model.* The model is the meat of the system. This is where the real information is modeled (hence the name). If we were working with a circuit design application, this layer is where objects such as Circuit, Transistor, Diode, etc. would be found. These objects have only the most limited understanding that there is a user interface above them. They have no direct knowledge of user interface issues.

OBJECT DEPENDENTS

Both major dialects of Smalltalk provide essentially the same Object Dependents facility. The idea is that a client object, which wants to be informed when some other object changes, registers itself as a *dependent* of that object. Since the requisite behavior for maintaining dependencies is implemented in the class Object, all objects may have dependents, be dependent on other objects, or both.

The detailed operation of Object Dependents is a topic for another time. We'll have to be satisfied with just a quick look at the top level of the behavior. In the simplest terms, an object which has changed and may have dependents sends itself a *changed* message. This results in each of the dependents, if any, of the object in question being sent a matching *update* message. A list of possible changed messages and their matching update messages is presented below:

```
changed message
changed: arg()
changed: arg() with: arg1
changed: arg() with:arg1 with: arg2
```


ServiceAgreement

Customer

Name: Sirius Cybernetics Corp.
Addres 1211 One Thousand St.
Mall Stop 2167a
City: Sirius Beta
Zip: 27511 - 6446

Details Contact

nt # 22-01497-6699
lanager John Smith

Customer

Name: Sirius Cybernetics Corp.
Addres 1211 One Thousand St.
Mall Stop 2167a
City: Sirius Beta
Zip: 27511 - 6446

Custom

Details Contact

Figure 1. Two windows on a single Customer.

“Whenever some aspect of Domain Object that might be of some importance to the outside world changes, the method of the domain object that actually changes the value performs an expression of the form self changed: attribute.”

```
update message
update: sender
update: arg()
update: arg() with: arg1
update: arg() with: arg1 with arg2
```

An object, A, may register itself as a dependent on another object, B, by sending B the message addDependent: with itself, A, as the argument. All dependents of an object are removed by sending the object the message release.

A NOTE FOR DIGITALK USERS

Digitalk does not provide one method that is very useful in dealing with Object Dependents. The missing method is Object>>removeDependent: and a possible implementation is:

```
Object>>removeDependent: aDependent
        "Remove a single object from my list of dependents."

  | dependents |

  (dependents := Dependents at: self ifAbsent: [^ nil])
                remove: aDependent
                ifAbsent: [].

  dependents isEmpty ifTrue: [self release]
```

Digitalk users should also beware of the confusion possible because ViewManagers implement their own independent changed-update framework, which is unrelated to Object Dependents though it uses much the same protocol. To avoid problems, we won't be sending changed messages to view managers.

TWO VIEWS ON ONE OBJECT

To keep all of the windows on a particular domain model object current, the domain model objects will generate self changed: messages whenever some aspect of their state has changed. It is assumed that when an view is opened on any dynamically updatable domain object, that the application control object registers itself as a dependent of the domain model object it is representing to the user. This will insure that the application control will receive the update: message when the state of the domain model object changes. It is also assumed that the responsibility for undoing the dependency link when the window is closed also resides with the application control object.

SETTING UP

When a window is opened on a domain object, using an openOn: message for example, the window informs its application control object that this domain object is to be its model object. It is at this time that the application control object should register itself as a dependent of the domain model. The following methods illustrate this set up:

```
CustomerEditor>>openOn: aCustomer
        "Scheduling — Open myself up as a window
        on the given Customer."

  self control domainModel: aCustomer.
  self open


CustomerEditorControl>>domainModel: aCustomer
        "Accessing — Set my reference to my domain model object.
        Make myself a dependent of this object."

  domainModel notNil ifTrue: [domainModel removeDependent: self].
  aCustomer notNil ifTrue: [aCustomer addDependent: self].
  domainModel := aCustomer
```

Given this set up, Figure 2 provides an illustration of a generic scenario for what happens when some attribute of a displayed domain object is changed by the user in one of two views on that object. In this example Control A and Control B are both dependents of Domain Object.

1. The user uses some control in the window to alter the value of an attribute of the domain object being presented to him. For example, a the name of a Customer may be changed.
2. As a result of manipulating a control, a command message is forwarded to the application control object of the window the user is working with. In the case of changing the customer's name, this might be a message like cmdSetCustomerName:.
3. In the course of processing the command message, Control A sends a message to the Domain Object to inform it that it must change some of its internal state. To continue the customer name example, this would likely involve sending Domain Object the message name:.
4. Whenever some aspect of Domain Object that might be of some importance to the outside world changes, the method of the domain object that actually changes the value per

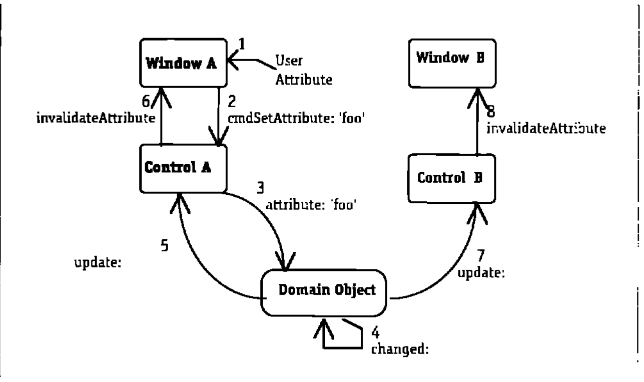


Figure 2. Keeping two windows up to date.

forms an expression of the form self changed: attribute. The parameter attribute varies depending on just what part of the domain model object was altered. For the change of name example, this argument would likely be *name*. In such a case, the setter method for name in the class Customer might look like the following:

```
Customer>>name: aString
        "Accessing — Set my name. Update anybody who's interested."
  name := aString.
  self changed: name
```

5. Whenever an object is sent the changed: message, as in event 4, all other objects which have been registered as dependents on the receiver of the changed: message receive update: messages. The argument passed along with the update: message is the same as that passed in with the original changed: message which started the process.
6. In processing the update: message, the application control compares the argument with those identifying aspects of the domain model in which it is interested. If a match is found, then the associated interface object is informed that some of the data it is displaying is no longer valid and must be updated. This is done by sending the interface object a message that tells it just what data needs to be redisplayed. If this were a view on the Customer as in the preceding examples, the method for update: would look, in part, like the following.

```
CustomerEditorControl>>update: aspect
        "Updating — Some part of my domainModel has changed. See if
         it is a part in which I'm interested. If it is, then direct
         the userInterface to update it."

  aspect = = #name
        ifTrue: [^ self userInterface invalidateName].
  aspect = = #company
        ifTrue: [^ self userInterface invalidateCompany].
  ^ super update: aspect
```

7. As Control B is also a dependent of Domain Object it will also receive an update: message of the same form as that received by Control A in event 5. This provides application B with an opportunity to keep its view of the domain object up to date even though application A was the source of the change. Application B does not need to know the source of the change. All it needs to know is what change took place. This update: message provides it with this information. The two views of Domain Object remain in sync.
8. Control B will handle the update: notification in much the same way as did Control A in event 6. In fact, if these are the same kind of views of Domain Object, then it will handle the message in exactly the same way. The end result is that a message will be passed on to Window B telling it that it must refresh the display of the changed item.

After the list of dependents of Domain Object is exhausted, that is each member of that list has received and processed the update: message, the process of changing an attribute of the domain model object is complete. Only at this point does the processing of the cmdSetAttribute message from event 2 complete.

Note that the domain model object did not need to know much about the application to provide this notification. All it needed to know is when to yell, "I've changed!" Other objects may or may not be interested. If they're not interested, they just won't listen.

“ Objects may or may not be interested. If they're not interested, they just won't listen. ”

CLEAN UP

When any of these windows are shut down, the dependency links with the domain model objects must be broken. This is best done using the removeDependent: message. When a window is closed it must, before it goes away entirely, pass on to its control object a message allowing it to clean up as well. A message like cleanUp will do nicely:

```
CustomerEditorModel>>cleanUp
        "I'm about to be terminated, clean up
        any messes I've left laying about."
  self domainModel removeDependent: self
```

The Object Dependents mechanism can be particularly useful for keeping collections of information up to date dynamically. This will be the topic of a future column. ▣

Greg Hendley is a member of the technical staff at Knowledge Systems Corporation. His specialty is custom graphical user interfaces using various dialects of Smalltalk and various image generators. Eric Smith is also a member of the technical staff at Knowledge Systems Corporation. His speciality is custom graphical user interfaces using Smalltalk (various dialects) and C. The authors may be contacted at Knowledge Systems Corporation, 114 MacKenan Drive, Cary, NC 27511, 919.481.4000.

TO PLACE A RECRUITMENT AD, CONTACT HELEN NEWLING AT 212.274.0640.

BOOK REVIEW

by Dan Lesage

SMALLTALK PROGRAMMING FOR WINDOWS

by Dan Shafer with Scott Herndon and Laurence Rozier

Prima Publishing
Roclin, CA
phone: 916.786.0426
fax: 916.786.0488
$39.95
ISBN 1-55958-237-5 1993

I am waiting for the day of the truly paperless book. The day when reading on an electro-luminescent or photo-polarized device provides me with as little eye strain as reading flat paper. I am sure that Dan Shafer is waiting for this day as well. On that day, the problem of publishing a timely technical book about rapidly changing technology will no longer exist.

Eighteen months ago, I reviewed Shafer's original Smalltalk book, entitled PRACTICAL SMALLTALK (THE SMALLTALK REPORT, October 1991). One of the issues I raised in that review was that the book presented examples in Smalltalk/V 286, just when Digitalk was moving toward PC desktop integration with Windows and OS/2 Presentation Manager. The paradigm used for modeling these new user interfaces had changed drastically from Model-Pane-Dispatcher. MPD lost its sex appeal for solving UI problems, although the fundamentals of Smalltalk were the same. Real-world Smalltalk development had moved on to a different paradigm.

Shafer's new book, which uses V Windows 2.0 as its base, is more timely than its predecessor. However, it is interesting that Digitalk's focus has moved onto Parts, once again leaving Shafer to play catch-up. What we need is the ability to publish a book directly from a Smalltalk image!

Once again the focus of the new book is a practical introductory guide for novice Smalltalk users. It acts as a supplement to the material provided by Digitalk. The format of the book is similar to the previous one. After two introductory chapters, it leads the reader through chapter pairs. The first chapter of each pair introduces important Smalltalk classes. The second of the pair highlights the use of these classes within a working example application.

The book describes seven detailed projects. The first is a List Prioritizer that prompts the user to prioritize text entries. The second consists of a Counter widget that introduces interaction between subpanes. The third project is a Calendar application that displays monthly pages, allowing you to navigate dates, highlighting holidays and the current date. The fourth application is an Appointment Book built by extending the calendar application in the third project. The Appointment Book introduces the ViewManager class. The fourth project also demonstrates how to manage multiple window interaction by adding a text based appointment window to the calendar. The fifth project is a Bar Graph Editor and Viewer. The sixth consists of a Form Designer that demonstrates how to create a user interface layout from a Smalltalk outline. The last project consists of a Clock that also hooks into the Calendar application. The clock is responsible for displaying the time and sounding alarms and chimes. The Clock project demonstrates the multiprocessing capability built into Smalltalk and how to use it in combination with ViewManager.

I found that the example projects contained within the book had greater relevancy to developing real applications than the ones presented in PRACTICAL SMALLTALK. Only the List Prioritizer, Counter, and Bar Graph Viewer appear as upgraded versions of examples used in the previous book. The remaining projects simulate the process of building real applications. They require the developer to add new functions to existing software rather than create designs from scratch. Changing the Calendar viewer into a time-based Appointment Book typifies how Smalltalk developers must constantly reorganize their code to accomodate new requirements. The Clock project, which is the cumulative effect of these requirements, provides new Smalltalk programmers with insight into the power of classes such as Time, Processor and Context (blocks). This last project demonstrates how to make these classes collaborate to simulate the behavior being modeled. The result of completing the last project is a sense of satisfaction and confidence. Developers should feel comfortable browsing the class hierarchy as they develop more complex applications.

The book includes a 3.5-inch diskette that contains V Windows 2.0 code, so browsing the examples is easy. Just remember to remove the diskette immediately when you buy the book or you will find that after a while, the soft back cover will look like it has been run over by an office chair!

There appear to be some errors within the printed Smalltalk code that do not appear on the diskettes. Pages 184 through 186 contain numerous syntax errors and erroneously repeated code. Unless you are a masochist, you should browse the code from your image rather than read the book to ensure correctness. Of course, that means you need your *paperless* book again, as you fly from Boston to Ottawa. Hmmm...

Highlights

Excerpts from industry publications

SOM

In practice, [IBM's] SOM (System Object Model) will allow programmers to "package" objects into blocks of code, of class libraries, that can be readily accessed from a C++ or Smalltalk program. Next month, IBM will extend SOM with a full CORBA (Common Object Request Broker Architecture) model. This Distributed SOM, or DSOM, spec will let objects be transparently accessed either locally or across a network.

IBM reveals its new software 'object'-ive, Alexander Wolfe, ELECTRONIC ENGINEERING TIMES, 5/17/93

POINTER-SAFE

At least triggers are specified in an SQL variant. SQL has no pointers and there is no need to worry about wild stores. Even if the application is written in a language that is not pointer safe (e.g., C) a wild pointer or running off the end of an array will not corrupt the database. However, most object database vendors and at least one relational vendor allow behavior specified in C or C++ to be optionally linked into a server process, and server processes contain very large caches of data. The problem in the relational environment is that the rows in the cache are assumed to satisfy all integrity constraints and that the cache is often shared amongst multiple clients. A seemingly experienced application developer once told me, in all seriousness, that mature C code doesn't produce any wild stores (and you wonder why DBAs sometimes seem paranoid). A wild store in this scenario can result in corrupted data being committed to the database. And the corrupted data might not have been read by the offending application program. Many object databases have the same problem with behavior specified in C or C++. These databases tend to bulk copy their caches to disk at transaction commit. This is one of the major reasons why I have always believed that a pointer-safe language such as Smalltalk is a much better data-manipulation language that C or even C++.

ODBMS: Tear down the walls, Jacob Stein, OBJECT MAGAZINE, 7 8/93

Shafer's style of writing in this book is down to earth. This should appeal to new programmers, but there are instances where I found the style to be a little subterranean. On page 141, for example, Shafer writes:

(It's amazing to think one can actually get paid for doing this kind of work, isn't it?)

I hope I never accidentally put that into my application comments!

On the plus side, this book has really made strides in the area of integrating an application into its surroundings. Appendix B discusses DDE and DLL interfaces and provides an example of adding a DDE link to the Calendar application from Microsoft Excel. The DLL example shows how to use multimedia extensions in combination with a sound board. The example demonstrates how to modify the Calendar project to play a sound file instead of beeping for alarm events.

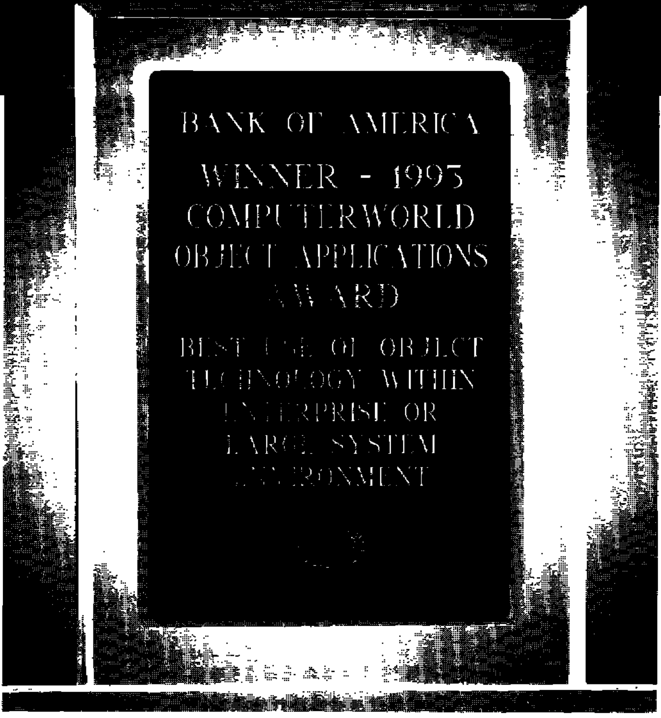
The end result of all these enhancements gives a Calendar application that is comparable in function to the Microsoft Windows desktop calendar. I believe that most programmers would classify this to be a true application, albeit a simple one.

Shafer demonstrates the use of fast prototyping a[illegible] [illegible]chanism for building applications. Throughout the book, he proposes designs that have minor flaws contained within them. He then leads the reader through the analysis required to correct the problem. This highlights an important point pertaining to the design of graphical applications. Most of the *discovered* problems have to do with event handling, sequencing, bad initialization and proper notification of change. To further complicate the analysis, these problems occur within a multi-window, multi-pane, multi-widget environment. This is also true in the real world: The hard part is not defining the visual aspects, it is getting the glue right. This book does an excellent job in highlighting these kinds of problems and demonstrating the type of analysis is required to correct them.

Once you overcome the silly book cover, the cartoons on the back and the fact that the publisher's name is about 3 times the size of the author's, the content of this book will be very useful to new Smalltalk programmers. The calendar application can form the basis of an introductory Smalltalk course. I know of one company that has modeled part of its internal training examples on those presented in the book. This book is a colossal improvement over its predecessor and it demonstrates what it takes to start building applications under Windows using Smalltalk. I recommend this book to new Smalltalk programmers who wish to quickly develop small scale applications within the Windows environment. ■

Dan Lesage is responsible for Distributed Systems Frameworks at Object Technology International Inc. This means that he gets to act as trial arbiter between very unlike pieces of hardware and software, protocol arbiter between collaborating classes in frameworks, personnel arbiter between team members and aqueous medium arbiter between aggressive piscatorial members of his aquaria. It occasionally means that he gets to develop software in Smalltalk. He can be reached at 613.820.1200 or dan@oti.on.ca.

LOOK WHAT HAPPENED WHEN DIGITALK BROKE INTO THE BANK.



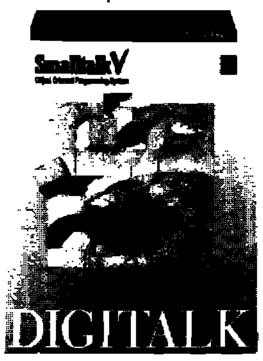
Congratulations to Bank of America on their new 11-state wide area network. A system they call "the most sophisticated distributed network in the world."

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