THE Smalltalk REPORT

Table of Contents

November–December 1994 **Vol4 No3**

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar–Apr, July–Aug, and Nov–Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1994 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is Flatly prohibited. Material may be reproduced with express permission from the publisher. Mailed First Class. Canada Post International Publications Mail Product Sales Agreement No. 290386.
Subscription rates 1 year (9 issues): domestic, $79; Canada and Mexico, $104, and Foreign, $119. To submit articles, please send electronic files on disk to the Editors at 885 Meadowlands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).
POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, P.O. Box 2027, Langhorne, PA 19047. For service on current subscriptions call 215.785.5996, 215.785.6073 (fax), P00976@psilink.com (email). PRINTED IN THE UNITED STATES.

Editors' Corner

Well, we're just back from Digitalk's Developers Conference in Irvine, CA. This is their third such gathering, and had quite a different feel to it than their previous two. Most notable was the lack of marketing effort on their part. They seemed more intent on allowing their developers to describe what's coming up in their long-awaited 3.0 product. This release represents a major shift for Digitalk, as they are planning on releasing their Parts Enterprise product on all their supported platforms at once. Digitalk users will find in this release some new features that should prove useful, as well as the cleaning up of many things from their earlier products. A few comments made that might be of interest to you from their senior staff:

- they state they will go public in 1995
- they are bundling their Parts, Smalltalk/V, and their Team/V products
- they plan to stop selling these as separate products sometime in the near future
- they don't plan to support their 16-bit version of the product in the future
- they are trying to put together a third-party Partners program

As we saw in early August at ParcPlace's Users Conference, and again this week with Digitalk, both these companies are targeting almost exclusively the Fortune 500 companies, and setting their price point for their products at the high end of the scale. This, coupled with IBM's Smalltalk and VisualAge targeting a similar market, seems to leave a major void in the marketplace for a low-end Smalltalk product. Neither QKS' Smalltalk nor Enfin's product offerings seem to be heading in this direction either. It would be nice to see a Smalltalk that might lack all the connectivity and glitz of these mainstream Smalltalks, but nonetheless fully supports the Smalltalk language and library. This would certainly allow for students and basement hackers to try Smalltalk, which in the long run can only help everyone in the Smalltalk marketplace. Time will tell if someone steps forward to fill this niche.

It's fascinating to see the interest in design patterns becoming more and more prevalent each month. The upcoming OOPSLA conference features a number of different presentations that will deal with this topic from a variety of viewpoints. Also, new books are coming to the market shortly that will describe the use of patterns, and the list of people discussing the topic on the Internet is also growing. Certainly, Kent Beck has been using this publication to further all of our understanding on the topic, and your feedback attests the popularity of his columns.

It seems to us that the approach offered by patterns offers the promise of group members to be able to articulate succinctly their ideas with one another, and to raise the level of understanding of everyone in the group. I know we are very interested in how we might be able to exploit patterns, both with the work being done by ourselves internally, as well as in our teaching. In fact, it is in this area that we might see a real win with this approach. We as teachers might be able to explain, through patterns, approaches to software development which have traditionally been difficult to articulate. We think it represents a very interesting area that we would like to see many of you explore for yourselves.

We hope you'll find this month's line-up of articles to be useful and stimulating. Certainly Bobby Woolf's article this month on the evolution of MVC provides some insight into the manner in which this much talked about but not well understood architecture has come about.

Enjoy the issue.

JOHN PUGH

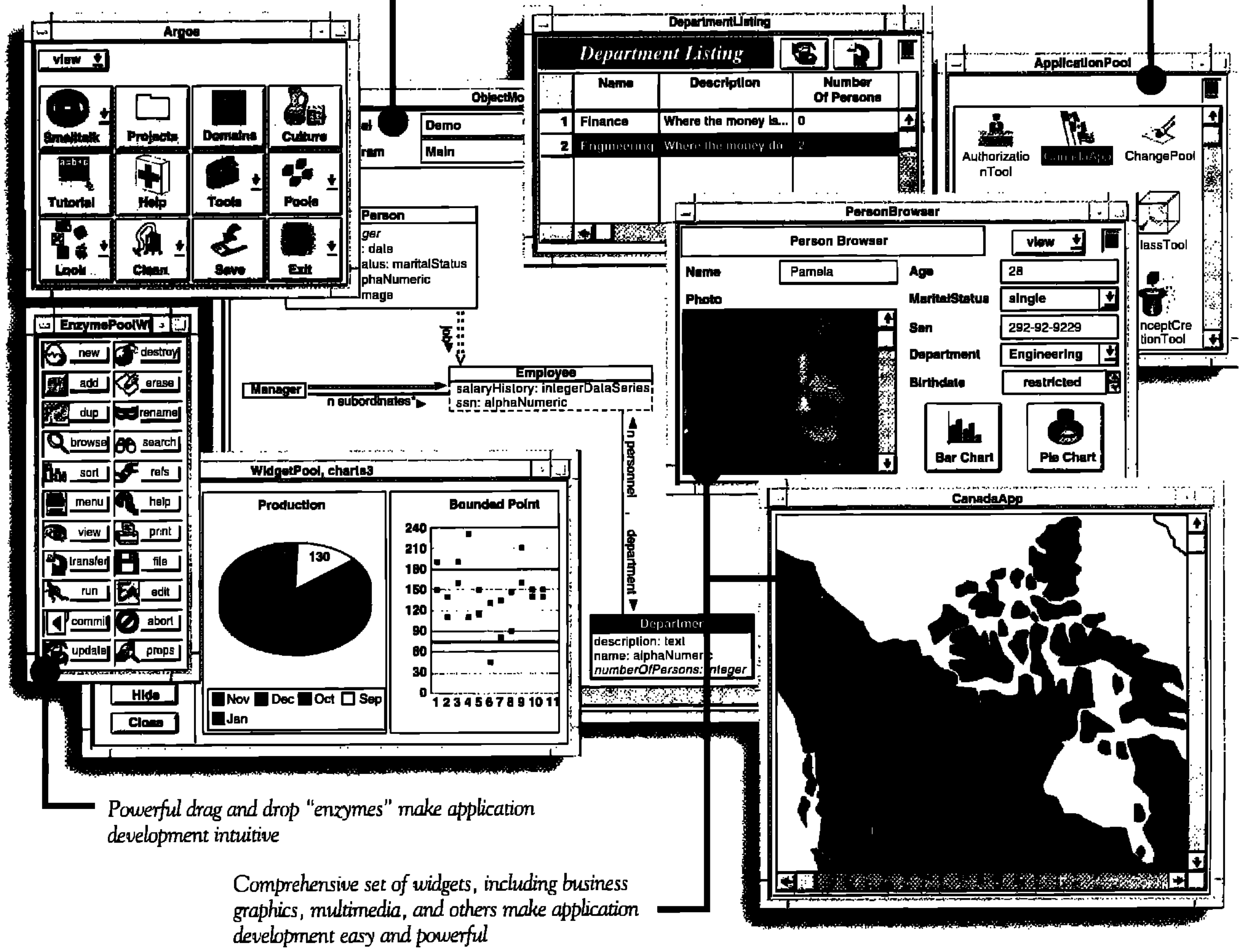
PAUL WHITE

Introducing Argos

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

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

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



	Name	Description	Number Of Persons
1	Finance	Where the money is...	0
2	Engineering	Where the money do...	2

Name: Pamela
Age: 28
MaritalStatus: single
Ssn: 292-92-9229
Department: Engineering
Birthdate: restricted

Employee
salaryHistory: integerDataSeries
ssn: alphaNumeric

Department
description: text
name: alphaNumeric
numberOfPersons: integer

Powerful drag and drop "enzymes" make application development intuitive

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

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

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

Find out why 3 of every 4 customers who've seen Argos purchase it to deliver their business-critical applications in weeks rather than years. If your organization needs objects today, let us prove it to you with Argos.

Contact us today at 1-800-VERSANT, ext. 416 or via e-mail at info@versant.com

VERSANT
The Database For Objects™

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

Improving dependency notification

Bobby Woolf

The changed:/update: protocol in Smalltalk allows a parent object to transparently notify its dependents of changes in the parent's state. However, the notification process is inexact, for all notifications are processed by all dependents. Although the framework itself is highly object oriented, the case statements the dependents must implement are not.

VisualWorks has significantly enhanced the framework with DependencyTransformers and ValueModels, objects that make dependency notification more efficient to execute and easier to maintain. Even programmers who are not aware of these extensions are using them through the code that VisualWorks automatically generates. By understanding how these extensions work, developers can improve much of the code they write and make better use of the code that VisualWorks generates.

THE CHANGED:/UPDATE: PROTOCOL

The changed:/update: protocol is a mechanism all objects have for notifying each other of changes in themselves. It is a weak style of collaboration that the source object hardly even realizes it's participating in. When two objects are defined as having a strong collaboration, each must explicitly know about the other and must support the other through a relatively complex interface. As an alternative, the changed:/update: protocol provides a simple interface through which all objects can collaborate by exchanging simple information in one direction only.

The information conveyed is that the source in the collaboration is notifying its targets that its state has changed; the targets in this relationship are called *dependents*, and the source is called (for lack of a better word) a *parent*. As Smalltalk's implementation has matured, the understanding of the role of the dependent has changed somewhat. Increasingly, a target in a dependency relationship is thought of as registering its interest in the source, an interest that may or may not be a dependency. In any event, the targets are still called dependents, although the phrases "register dependency on" and "express interest in" are used interchangeably. Thus far, VisualWorks does not appear to recognize any other kind of interest than dependency for a target to have.

Because the dependents are hidden from the parent by its changed protocol, it knows neither how many dependents it has nor what their types are; all a parent knows is that at any time, it has some number of dependents and they are each some type of Object. This forms a very weak collaboration that essentially hides dependents from their parents. For example, in the model-view-controller framework (MVC), the view and controller know what their model is but not vise versa. This is accomplished by making the view and controller dependents of the model; the model informs its dependents when it changes (in case it has any dependents), then the view and controller receive this update, and then they reretrieve their values as they wish. The responsibility for keeping the view in sync with the model lies completely with the view; the model doesn't even know the view exists, it just knows that periodically it's being queried for its state.

This framework is ideal for modeling the dependencies between any group of objects, not just model-view-controller clusters. Real world objects often require that other objects react when one changes. To encapsulate the one, it should not be aware of the effects it has on the others. The dependency framework does this well, which is one of the primary reasons Smalltalk is such a good language for modeling real world objects and their interdependencies.

All objects handle dependents, but subclasses of Model handle them more efficiently. In fact, the only difference between a Model and an Object is how efficiently it is able to handle its dependents. So if an object's class would normally be a direct subclass of Object, but it notifies its dependents often, it will do so more efficiently if its class is a subclass of Model. The reverse is also true: If an object rarely notifies its dependents and its class is a direct subclass of Model, its class can just as easily be a direct subclass of Object with no significant decline in performance.

Figure 1 shows the messages in the changed:/update: protocol.

SPECIFYING WHAT CHANGED

Ordinarily it is not enough for a parent to simply notify its dependents that it changed. The dependent usually needs to know not only that the parent changed somehow but also exactly what part of the parent changed. That way, if the part is something the dependent is interested in, it can react; otherwise, the dependent can ignore the notification.

For convenience, the parts of an object's state are often organized into aspects. An *aspect* is a single value (an object) stored within the container object. It is retrieved and set via getter and setter messages sent to the container, and it notifies its dependents (via its container) when it changes. For an aspect named aspect, the convention for its getter selector, setter selector, and update value is aspect, aspect:, and #aspect.

An aspect usually corresponds to an instance variable of the same name in the container, but this is not a requirement. Some aspects are virtual in that they act like the public access for an inst var, but their implementation actually delegates to other aspects and/or behaviors within the container. Also, not all aspects have setters; these provide access to values that may be

OBJECT TECHNOLOGY TRAINING IS A MOVING TARGET. KSC WILL MAKE SURE YOU HIT IT.

With software design taking on new dimensions almost daily, you can't afford to make your transition to object technology a hit or miss operation. Choose the training specialists who can help you keep up with the changes as they happen — Knowledge Systems Corporation.

Knowledge Systems Corporation offers the most comprehensive classroom education program available, including Smalltalk courses for ParcPlace, IBM and Digitalk dialects. Our materials are based on over eight years experience in applying object technology. All KSC instructors undergo a rigorous certification process and have practical experience in real-world application development.

KSC CLASSROOM EDUCATION COURSES

- *Manager's Introduction to Object-Oriented Technology*
- *Object-Oriented Analysis and Design*
- *Introduction to Object-Oriented Programming in Smalltalk/V*
- *Introduction to Object-Oriented Programming in VisualWorks*
- *Introduction to Object-Oriented Programming in IBM Smalltalk*
- *Advanced Programming in Smalltalk/V*
- *Advanced Programming in IBM Smalltalk*
- *Bridge Course from Smalltalk/V to VisualWorks*

All KSC courses are offered at the client site or our new training center in Cary, North Carolina. Courses are structured so that students spend more than half their classroom time in hands-on programming activities. To maximize actual programming time, all classes have a ratio of one student per machine.

KSC has a proven history with companies such as American Management Systems, GE Capital Corporation, IBM, Northern Telecom, The Prudential, Southern California Edison, and Sprint. Our classroom education, apprentice programs and development services offer our customers a total solution to real business problems.

To put your object technology transition on target, call the Smalltalk training specialists at Knowledge Systems Corporation, 919-481-4000. Or email: salesinfo@ksccary.com. 4001 Weston Parkway, Cary, North Carolina 27513.

KNOWLEDGE SYSTEMS CORPORATION

919-481-4000

Dependency Notification

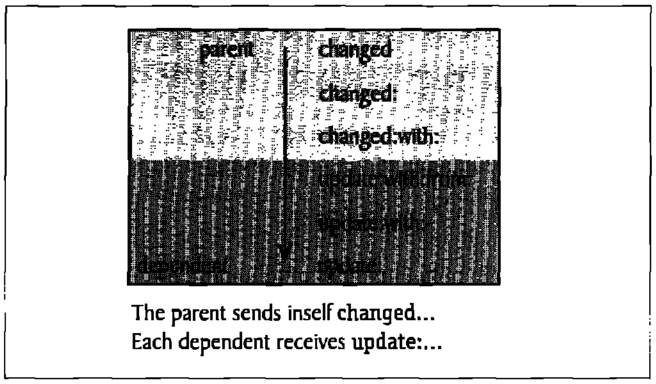


Figure 1. Messages in the **changed:/update:** protocol.

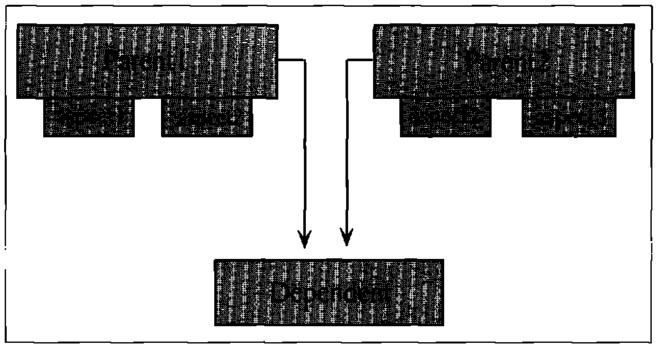


Figure 2: An example of dependency relationships.

queried but not altered. If an aspect's value cannot be changed, whether and how it notifies its dependents is irrelevant.

An aspect's update value refers to how the parent can specify to its dependent what aspect has changed. If the parent has a couple of aspects, one of which is area, dependents who are only interested in that particular change will listen for the update value #area and ignore the others.

THE ORIGINAL PROTOCOL

Back in Objectworks 2.5, the usefulness of the dependency framework and its changed:/update: protocol was almost equaled by its implementation difficulties. The difficulties included the following:

- Every method that changed an aspect in a parent object ought to send out an update, just in case the parent had a dependent on this aspect. Because any object can have dependents, every object should be implemented as though it might be a parent.
- Every dependent object had to implement one or more variations of update:. This method had to test for every aspect value its object is interested in. Unfactorable, it quickly became a lengthy case statement. The list format usually obscured which aspect values were expected to be sent by which parent(s).
- Every dependent was notified of every change, even though each often ignored the majority of the updates its parents sent.
- The traceability between senders of an update and its listeners was very low. Assuming the update values were symbols, the Smalltalk standard, the *browse senders* command would show most (but not necessarily all) the senders and listeners but would also mix into the list any methods that sent messages with the same name as the update (such as the getter for the update's aspect). Even with this list, matching a particular sender to its listeners was difficult at best.

In a small application implemented in a few months by one person, this lack of traceability between a sender of an update and its listeners was usually not a problem; the code didn't show the relationship, but the developer already had it in his head.

However, if an application was large and required a team of developers a year or more to create and still more developers to maintain, this lack of traceability would cause a maintenance disaster. Over time, such a system would become filled with parents sending out updates that no one ever listened for and dependents testing for updates that were never sent—at least, not to them. This extraneous code brought no value to the system and in fact added to its code bulk and slowed its execution efficiency. Yet, such code was difficult to isolate and remove because a developer had difficulty "proving" that the code was unnecessary. Therefore, it would be better to leave it in than to risk adding subtle but significant bugs to the system.

An example

Having said that the traceability between an update's sender and listener(s) is low, here's an example.

Say part of a system has two parent objects, the first with aspect1 and aspect2, the other with aspect2 and aspect3 (they duplicate the name of one of the aspects). The system also has a dependent object that is dependent on both parents and which listens for both parents' aspects.

The four setter methods for the two parents look like this example for aspect1:

```
aspect1: newValue
    "Set the value of aspect1 to newValue."
    aspect1 := newValue.
    self changed: #aspect1
```

The dependent object will make itself a dependent of both parents:

```
initializeParent1: parent1 parent2: parent2
    "Initialize the receiver to use its parents
    ...
    parent1 addDependent: self.
    parent2 addDependent: self.
    ...
```

The dependent object's update:... implementor will look like this:

```
update: anAspect
    "Standard."

    anAspect == #aspect1 ifTrue: [<a little code>].
    anAspect == #aspect2 ifTrue: [<a lot of code>].
```

Figure 2 shows the objects in this example.

There are several problems with this implementation:

- Changed notification for aspect3 is being sent, even though no one is listening for it.
- The dependent will receive notification of changes to aspect3 and process them, only to finally ignore them.
- It is not clear from the update: method what objects are expected to be sending these changed aspects; the developer must locate initializeParent1:parent2: to find this information. Using update:with:from: might help, but is usually not required to make the mechanism work, and so it is often not used.

“The Difference Between Success and Failure in IBM Smalltalk”

New!

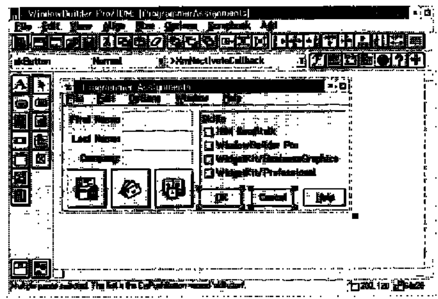
WindowBuilder™ Pro is an interactive tool that lets you build polished user interfaces fast in Smalltalk from Digitalk *and* IBM. WindowBuilder Pro (WBPro) saves you from the job of building UIs in code. It helps simplify maintenance and increase consistency.

Like VB, with Real Objects

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

OBJECT SHARE SYSTEMS INC.

Objectshare Systems, Inc.
5 Town & Country Village
Suite 735
San Jose, CA 95128-2026
Fax 408-970-7282
CompuServe 76436,1063



Building user interfaces is easy

High-Level Controls for WBPro

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

No runtime fees are required for applications developed with WBPro. Free support for the first 90 days.

All products include complete documentation. Support subscription available. WindowBuilder Pro/V is compatible with Team/V. Code generation in IBM Smalltalk is totally Motif compliant. © Objectshare Systems, Inc. 1994

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

— Milan Sremac, President, Medical Software Systems

WINDOWBUILDER PRO/V

For Digitalk Smalltalk/V	Win	$295
	Win32	$395
	OS/2	$495

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

— Gordon Sheppard, Senior Technologist American Management Systems

WINDOWBUILDER PRO

For IBM Smalltalk	OS/2 std.	$495
	Team	$695
	Windows std.	$495
	Team	$695

Call to order (408) 970-7280

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

VisualAge

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

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

WidgetKit/Professional for VisualAge Windows . . . **$795.**

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

WidgetKit/ Business Graphics
for VisualAge Windows . . . **$795.** for VisualAge OS/2 . . . **$795.**

No run-time fees for applications, 30-day money-back guarantee, 90 days free support.

Call (408) 970-7280 *9 AM to 5 PM PST, M-F*

OBJECT SHARE SYSTEMS INC.

Objectshare Systems Inc.
5 Town & Country Village, Suite 735
San Jose, CA 95128-2026
fax 408-970-7282, cis 76436,1063

Smalltalk/V

Programming tools for Smalltalk/V®	Win	Win32	OS/2
WindowBuilder™ Pro/V Interactive UI builder for Smalltalk/V, Save time, simplify maintenance.	$295	$395	$495
Add high-level controls to WindowBuilder Pro/V			
Subpanes/V for WBPro/V Table panes, columnar list box, drag and drop, bitmap pane, and more	$235	$235	$235
WidgetKit™/CUA'91 for WBPro/V Full CUA'91 controls	–	–	$295
WidgetKit/Professional for WBPro/V Spreadsheets, tables, validation, graphic viewers, virtual spreadsheet, and more	$395	$395	–
WidgetKit/Business Graphics for WBPro/V Pie, line, area, gantt, high-low-close, 2-D and 3-D graphs, printing and more	$495	$495	–

No run-time fees for applications, 30-day money-back guarantee, 90 days free support.

Call (408) 970-7280 *9 AM to 5 PM PST, M-F*

OBJECT SHARE SYSTEMS INC.

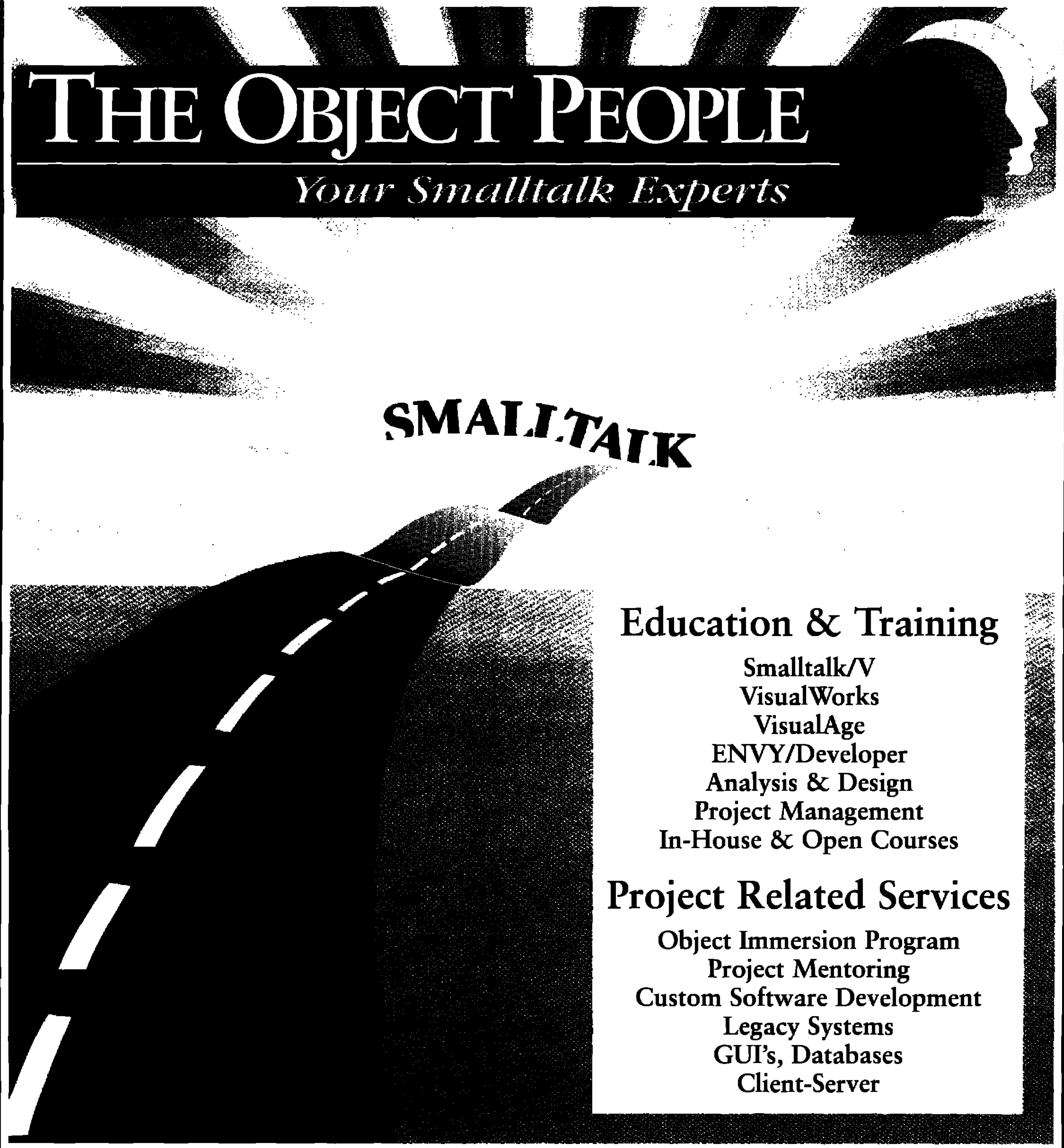
Objectshare Systems Inc.
5 Town & Country Village, Suite 735
San Jose, CA 95128-2026
fax 408-970-7282, cis 76436,1063

TRANSITIONING TO SMALLTALK TECHNOLOGY?
INTRODUCING SMALLTALK TO YOUR ORGANIZATION?

Travel with the team that knows the way...

THE OBJECT PEOPLE

Your Smalltalk Experts



Education & Training

Smalltalk/V
VisualWorks
VisualAge
ENVY/Developer
Analysis & Design
Project Management
In-House & Open Courses

Project Related Services

Object Immersion Program
Project Mentoring
Custom Software Development
Legacy Systems
GUI's, Databases
Client-Server

The Object People Inc. 509-885 Meadowlands Dr., Ottawa, Ontario, K2C 3N2
Telephone: (613) 225-8812 FAX: (613) 225-5943

- The update: implementor is effectively a case statement; this is a procedural structure that is discouraged in Smalltalk, not to mention the fact that it is inefficient to execute.
- The update blocks are buried inside the update: method and are unavailable for reuse. (However, some developers have discovered a kludgy backdoor for accessing these blocks: The object can *send* update: to *itself* (which is very unconventional!), providing the appropriate parameter to activate the desired block. Then the poor maintenance person must look for senders of a changed aspect not only in the parent objects but in the dependent as well!)

Perhaps the worst problem, one that makes this kind of collaboration especially difficult to maintain in legacy code, is the following:

- It is unclear which sender of aspect2 the dependent is listening for; the source is ambiguous. As implemented, the dependent is listening for that aspect from either parent (and in fact any parent that sends it)—but was that by design? The original developer could have clarified this by testing the from: parameter (in update:with:from:), but that wasn't necessary to make the code work, so he or she didn't do that test.

As the system's requirements change, the parent objects will be redesigned and refactored and the dependents' interests will have to be reevaluated. When making these enhancements, maintaining the dependency relationships will require inordinate time and effort as well as significant guesswork.

Analysis

The basic problem with implementing dependencies in this fashion is that one large complex object is made a dependent directly of another large complex object. One cannot easily tell how many of the parent's aspects the dependent requires. If the two are redesigned such that the dependent no longer requires one (or two, or 10) of the parent's aspects, does it still need to be a dependent of the parent? This kind of guesswork when maintaining code is fertile ground for introducing bugs, ones that in this case are subtle and difficult to isolate.

A BETTER WAY: DEPENDENCYTRANSFORMERS

VisualWorks 1.0 introduced a new object, DependencyTransformer, which is a substantial improvement to the dependency framework. A DependencyTransformer acts as a bridge between the parent object and the dependent object. It listens for exactly one update value from exactly one parent; when it receives that update, it runs a specified method in its sole target, the dependent. In this way, the update being listened for is factored out of the dependent object and a one-to-one relationship is established between the update that a parent sends out and the method that should subsequently be run in the dependent.

This solves many of the problems (although not all) that the old framework had:

- Dependents no longer have to implement update:...; it is already implemented in DependencyTransformer to listen for the specified update value. Instead, the dependent implements the method to be run when the update occurs, a method often named *aspect*Changed. This is not only more intuitive but is reusable as well.
- The update: method does not contain a case statement, just a single test.
- The dependent is only notified of the change for which it wants to listen. The DependencyTransformer must still process every update the parent issues, but because it is only listening for one aspect, it can quickly determine whether the aspect received is the correct one.
- When the dependent has multiple parents, it is clear which one is expected to be sending the update: It's the one at the source of the DependencyTransformer doing the listening. If the dependent wants to listen for the same update from two parents, it will specifically use two DependencyTransformers.

Thus, the use of DependencyTransformers makes dependents updating more encapsulated and the code used to implement it more traceable and easier to maintain. This comes at the expense of a layer of indirection between the parent and dependent—a small price for a better factored set of collaborating objects.

An example

Here's an example of the increased traceability DependencyTransformers provide.

This will revisit the example above, where a system has two parent objects with aspects 1-2 and 2-3, respectively, and a dependent object that is dependent on both parents and that listens for two of its parents' aspects.

The four setter methods for the two parents look like the example above.

The dependent object will make itself a dependent of the parents with this code:

```
initializeParent1: parent1 parent2: parent2
    "Initialize the receiver to use its parents."
    ...
    parent1
        expressInterestIn: #aspect1
        for: self
        sendBack: #aspect1Changed;
        expressInterestIn: #aspect2
        for: self
        sendBack: #aspect2OfParent1Changed.
    parent2
        expressInterestIn: #aspect2
        for: self
        sendBack: #aspect2OfParent2Changed.
    ...
```

Each send of expressInterestIn:for:sendBack: creates a DependencyTransformer whose source is the receiver and whose target is the for: parameter (usually self). It listens for the update value specified by expressInterestIn: and sends the message specified by sendBack: to the target.

The dependent object will implement methods aspect1Changed, aspect2OfParent1Changed, and aspect2OfParent2Changed to be the code that was in the update blocks:

```
aspect1Changed
    "aspect1 changed. Update the receiver."
    <a little code>
```

Dependency Notification

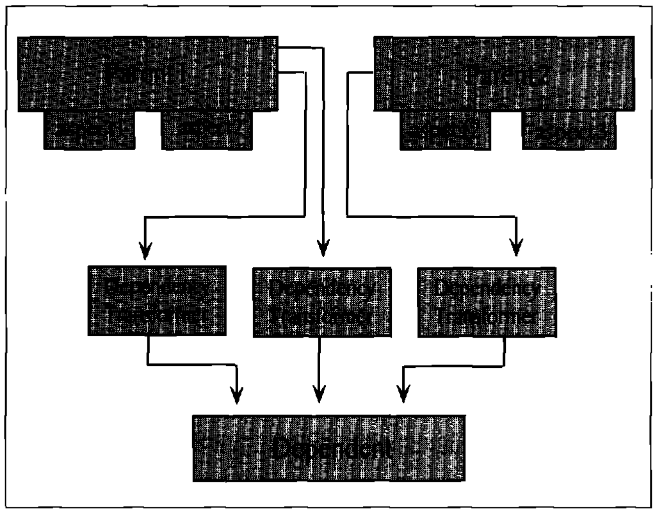


Figure 3. An example of dependency relationships using **DependencyTransformers.**

```
aspect2OfParent1Changed
    "aspect2 changed in Parent1.
    Update the receiver."
    self aspect2Changed.
    <code particular to aspect2 in Parent1>

aspect2OfParent2Changed
    "aspect2 changed in Parent2.
    Update the receiver."
    self aspect2Changed.
    <code particular to aspect2 in Parent2>

aspect2Changed
    "aspect2 changed. Update the receiver."
    <a lot of code, applicable to Parents 1 and 2>
```

Figure 3 shows the objects in this example.

Notice that there is no ambiguity as to which parent object the dependent object expects to send aspect2. Also, if the dependent no longer has to listen for aspect2 in Parent2, the code to register that interest will be removed. When the last interest in a parent is removed, the object will automatically no longer be dependent on that parent.

However, even with DependencyTransformers, certain problems with the dependency framework still remain:

- Every method that changes a parent object's state is still obligated to send out an update.
- Every dependent is still notified of every change, although DependencyTransformers factor the necessary testing for better encapsulation.
- The traceability between senders of an aspect and its listeners is better but still not perfect. Using "browse senders" still produces a bewildering list of possibilities. Each of the parent's update values still act as a vague conduit from the part of the parent's state that is changing to the DependencyTransformer which is listening for the change.

Analysis

DependencyTransformer is a major step in the right direction, reducing or eliminating many of the shortcomings of the original dependency framework, but others still remain. The problem is that the parent object is still one large complex object with numerous aspects; yet, each DependencyTransformer is only interested in a single aspect. Thus, a DependencyTransformer needs a way to be a dependent of just one single aspect in a parent, *not* the entire parent object.

AN EVEN BETTER WAY: VALUEMODELS AND DEPENDENCYTRANSFORMERS

The ValueModel framework was introduced in Objectworks 4.*x* primarily to implement a structure through which any solitary value in the system could be isolated and given the generic aspect value. However, a helpful bonus of ValueModels is that they provide a convenient place to register all of the dependencies of that value. In fact, since the ValueModel controls the access to setting the value, it is ideally positioned to inform dependents when the value is changed. Thus, when a parent object uses ValueModels to store its aspects, whenever those aspects' values change, the parent object does not have to take responsibility for notifying its dependents; the ValueModels will do that. Thus, DependencyTransformers factor the responsibility of listening for updates out of the dependent, and ValueModels factor the responsibility of issuing updates out of the parent.

ValueModels and DependencyTransformers work hand in hand. Because the only update value a ValueModel ever sends is #value, that's the only one its DependencyTransformers ever have to listen for. Thus, the difference between two DependencyTransformers on a single ValueModel is the dependents they update and the messages they send to do so.

To register dependency on a ValueModel's value, a dependent sends it onChangeSend:to:, specifying itself as the target. This creates a DependencyTransformer between the ValueModel and the dependent. This DependencyTransformer is not connected to the parent object with its many aspects; it's connected directly to the aspect itself. Thus, the only time that it receives an update is when the value changes, so checking to make sure the aspect is the correct one is merely a formality (it's always #value), and it tells the dependent to update.

Thus, using ValueModels and DependencyTransformers together eliminates the problems with simply using a DependencyTransformer alone:

- The methods that change a parent object's state are not obligated to send out updates if those parts of state are stored in ValueModels.
- The only dependents that are notified when a value changes are those of that particular value; thus, no one is informed who is not interested.
- The traceability between an object's aspect and one of its dependents is a very clear bridge between the two objects. Look at the sender of onChangeSend:to: (usually in the dependent's implementor of #initialize...) and it will explain everything: what value is being listed to, what method will be run when the value changes, and what object is the dependent that contains that method.
- Even at run time, the traceability is clear: One can inspect the ValueModel and its list of dependents (which are DependencyTransformers) and see what the targets of the

DependencyTransformers are. This is even easier to do using First Class Software's ObjectExplorer. No guesswork is required.

Thus, a combination of ValueModels and DependencyTransformers eliminate all the problems of the original dependency framework. The dependencies are set up in one place and execute efficiently. The setup shows direct traceability between the aspect being monitored and the result in the dependent, taking the guesswork out of maintenance. The dependency implementation reflects the design exactly.

An example

Here's an example showing the increased traceability ValueModels and DependencyTransformers provide. Revisiting the standard example, there is a system that has two parent objects with aspects 1-2 and 2-3, respectively, and a dependent object that is dependent on both parents and which listens for two of its parents' aspects.

The four setter methods for the two parents look a bit different now because the parents store their aspects in ValueModels. Using aspect1: as an example gives the following:

```
aspect1: newValue
    "Set the value of aspect1 to newValue."
    self aspect1Holder value: newValue


aspect1Holder
    "This method was generated by UIDefiner."
    ^ aspect1Holder isNil
        ifTrue: [aspect1Holder := nil asValue]
        ifFalse: [aspect1Holder]
```

The dependent object will make itself a dependent of the parents with this code:

```
initializeParent1: parent1 parent2: parent2
    "Initialize the receiver to use parentObject."
    ...
    parent1 aspect1Holder
        onChangeSend: #aspect1Changed
        to: self.
    parent1 aspect2Holder
        onChangeSend: #aspect2OfParent1Changed
        to: self.
    parent2 aspect2Holder
        onChangeSend: #aspect2OfParent2Changed
        to: self.
    ...
```

Each send of onChangeSend:to: creates a DependencyTransformer whose source is the receiver (a ValueModel) and whose target is the to: parameter (In fact, this example shows that the to: parameter is pretty redundant, because it seems to always be self). Because the source is always a ValueModel, the update value being listened for is always #value. The message to be sent to the target is specified by the parameter of onChangeSend:.

The dependent object will implement methods aspect1Changed, aspect2OfParent1Changed, aspect2OfParent1Changed, and aspect2Changed as before.

Figure 4 shows the objects in this example.

Notice that the parent objects never send themselves changed:; the ValueModels (accessed via the messages *aspect*Holder) do that for them. And the senders of onChangeSend:to: in initializeParent1:parent2: tell the developer everything he needs to know about what the dependencies are.

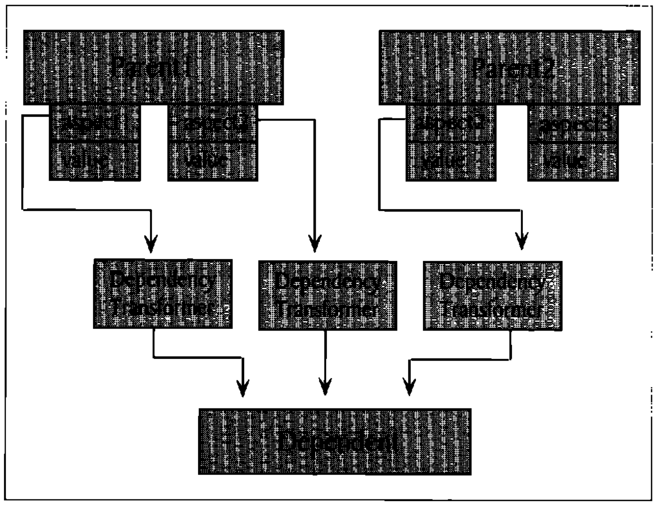


Figure 4. An example of dependency relationships using **ValueModels** and **DependencyTransformers.**

Analysis

All the problems of the original dependency framework have been solved:

- Setter methods aren't required to send out updates.
- Dependents don't have to implement update:....
- Only dependents who care about a particular change are notified.
- The traceability between parent aspects and their dependents is quite clear when inspecting either the static code or the run time objects.

Systems using this extended dependency framework run more efficiently and are easier (and safer) to maintain.

CONCLUSIONS

The dependency framework from Objectworks 2.5 is good, and it still works for small one person projects. But VisualWorks has extended it with ValueModel and DependencyTransformer to make it much better, especially for the large commercial applications for which Smalltalk is increasingly being used. Because the new framework is a superset of the old one, developers can continue to use the old one when desired; they can also use just parts of the new one (just ValueModel or just DependencyTransformer). The best results, however, are produced by using both parts of the new framework. It will factor the changed:/update: code more effectively, execute updates more efficiently, and be easier and safer to maintain for the life cycle of the code. Not bad for a couple of classes that many developers don't even realize they're using. ♀

Bobby Woolf is a Member of Technical Staff at Knowledge Systems Corp., where he is developing techniques for applying Smalltalk to large software projects. He has also been a Software Engineer at Ascent Logic Corp., where he gained considerable experience maintaining the complex dependency relationships in its large scale Smalltalk application, RDD-100. Comments are welcome at woolf@acm.org.

A quick peek under the covers of IBM Smalltalk

Steven G. Harris and Hal Hildebrand

IBM released VisualAge some time ago, entering this product into the Smalltalk marketplace with somewhat less than a roar. The initial release was available only under OS/2, which has limited its appeal to many potential end users. Over the last several months, we have been actively involved in using and evaluating the upcoming release of IBM Smalltalk, the foundation on which VisualAge is built. As we write this article, the product is still in Beta testing. It runs under Windows and OS/2, and we feel that, from many points of view, it is going to cause the roar that was missing before.

So what's the big deal? Actually, it's not just one *big* deal, it's a lot of *little* deals that add up to something that makes die-hard Smalltalkers like us sit up and take notice. First, IBM has separated the VisualAge visual programming environment from the base IBM Smalltalk product. It is a separate application, which can be loaded into the base image—use it if and when you want it. Second, IBM has established a formal program that actively encourages third-party suppliers—the first off the mark will be ObjectShare's WindowBuilder, coming soon to an IBM Smalltalk platform near you. Last, IBM Smalltalk is pretty hot stuff technically, and this characteristic of the product is the subject of this article.

As we are not doing a product review, we'll limit our recommendations to the following statement (sure to warm the hearts of IBM marketers everywhere): IBM Smalltalk is definitely worth taking a serious look at. Now, on to the show.

What's in this thing? How is it different from the Smalltalk products we've all come to know and love? We'll concentrate on the two most important areas: the class library and the development environment. We will provide comparisons with ParcPlace's VisualWorks product and Digitalk's Smalltalk/V, because we are intimately familiar with them.

THE CLASS LIBRARY

Wherever possible, IBM has chosen to use industry standards for developing their class library. Because the ANSI standard process (already in progress for Smalltalk) moves at glacial speed, they have made heavy use of the Blue Book (Goldberg and Robson) as well as of IBM's own proposed standard for Smalltalk, often called the Red Book. Where things get more interesting, however, is IBM's usage of POSIX.1 for their file interface classes, X-Windows for the graphics subsystem, and OSF/Motif for widgets. As shown in Table 1, they define a number of major subsystems and attempt to adhere to a standard in each.

Of course, programmers who are Smalltalk-fluent in essentially any dialect have very little learning curve to get started. Those that are familiar with POSIX.1, X-Windows, and OSF/Motif get to leverage their existing expertise, cast in a nice Smalltalk framework. People who know nothing about POSIX.1, X-Windows, or OSF/Motif certainly aren't any worse off than they would be learning the VisualWorks or Smalltalk/V protocols. On the other hand, if you are already familiar with Windows and OS/2 programming to the degree you need to be to do low level Smalltalk/V graphics work, or if you already know everything there is to know about ValueModels and the UIBuilder framework in VisualWorks, you won't find a great deal of commonality in these areas with IBM Smalltalk. The way that IBM Smalltalk has implemented the interface to the windowing and graphics subsystems makes it easy to pick up a book on programming X and Motif interfaces and apply it directly to programming the IBM implementation. Finally, documentation that exists before the release of the system.

It is also worth mentioning that many of the classes in these subsystems have been prefixed with an abbreviation of the subsystem name. For example, all the widget classes begin with Cw, which stands for Common Widgets. This approach helps you keep the modularity of the image in mind, but it does take some getting used to. It's also nice to see Smalltalk vendors practicing what they preach in regard to prefixing class names to avoid name-space conflicts.

Base classes

What we would think of as the Smalltalk base classes, IBM groups into the Common Language Data Types and Common Language Implementation subsystems. There should be very few surprises for any Smalltalk programmers in these areas. IBM's class hierarchy approaches the VisualWorks class library in terms of sheer numbers of classes and built-in richness.

Perhaps ironically, IBM has chosen to adhere to the Blue Book and X Windows convention of the Point 0@0 being at the upper left corner of the screen, *x* increasing to the right, and *y* increasing downward. This choice was made to facilitate portability in spite of the fact that the OS/2 Presentation Manager usage is different. Digitalk's current releases match the convention of the host windowing system, requiring usage of messages like rightAndDown: instead of simply +, as + is not consistent between the Windows and OS/2 implementations. ParcPlace goes with the Blue Book. From a cross-platform compatibility point of view, the impact of this decision goes a long way to keeping hair out of your hands and on your head, where it belongs.

IBM Smalltalk uses block contexts as does Digitalk Smalltalk, while ParcPlace uses block closures. This means that blocks that should be "clean" (i.e., no references to the outer context) are not treated as a special case, and thus, IBM Smalltalk does not provide a speed advantage for using them.

Following Object Technology International's approach in ENVY/Developer, IBM Smalltalk uses a single instance of SystemConfiguration class (called System) as a central point for information on how the image is configured and for funneling startup and shutdown messages. The original ENVY/Developer approach has been extended to include information on what subsystem types are installed, opening up the potential for integration with other products.

IBM's process model essentially follows the Blue Book approach. If you're comfortable with forking lightweight Smalltalk processes, you should feel at home here. They also provide an implementation of the Delay class (like ParcPlace) that allows you to impose delays based on the system real time clock. In Smalltalk/V, it is extremely difficult to implement delays that don't lock up the system. In IBM Smalltalk, as in VisualWorks, you can delay the entire user interface for as long as you like and still have the system behave as it should. The value of this subtle point goes a long way when programming with multiple Smalltalk processes.

Graphics subsystem

The graphics subsystem is complete and well implemented, covering all the necessary graphics operations used by the host graphics system. The IBM Smalltalk common graphics subsystems let you:

- define drawing attributes such as line widths or styles through the use of graphics contexts
- perform drawing operations such as drawing lines, arcs, text strings, and polygons
- manipulate colors
- manipulate fonts
- manipulate two-color bitmaps and multicolor pixmaps

IBM Smalltalk's graphic subsystem is based on X Windows, and it provides the same functions as the XLib C calls in that standard. This strategy results in a standard documented interface for dealing with the graphics capability of the host system regardless of whether you are on a Windows, OS/2, or AIX platform. IBM Smalltalk uses a well-documented strategy for converting the various types of objects and function calls used by XLib into the equivalent Smalltalk objects and methods. The upshot of this approach is that you can pick up any book that deals with X Windows and start using IBM Smalltalk graphics right away.

Windowing subsystem

As in the graphics subsystem, IBM has chosen to base their window interface on another industry standard, OSF/Motif. The windowing subsystem is known as the common widgets subsystem in IBM Smalltalk. This subsystem allows the developer to

- create individual widgets (or controls), including buttons, lists, text menus, and dialog boxes
- create compound widget structures by combining individual widgets
- specify the positioning of widgets relative to each other
- program actions to occur in response to user actions

IBM Smalltalk again uses a standard translation strategy for converting OSF/Motif C types and functions to Smalltalk classes and methods. Like the common graphics, you can easily find documentation of these interfaces at your local bookstore.

The widget interface is event driven, and the hierarchy is designed in two pieces. The CwWidget hierarchy defines the common interface to the system. The implementation hierarchy, under OSWidget, is the low-level interface to the host operating system. Thus, unlike VisualWorks, IBM Smalltalk actually uses the underlying host resources for windows, menus, fonts, and such, while maintaining a platform-independent interface. IBM Smalltalk maintains the look and feel of the host system, as well as the speed inherent in host operating system supported functions. The clean implementation helps make IBM Smalltalk's user interface feel snappy in spite of a relatively slow virtual machine.

"It's also nice to see Smalltalk vendors practicing what they preach in regard to prefixing class names to avoid name-space conflicts."

The implementation includes some wonderful goodies for the developer, including a full implementation of form widgets, which provide extremely powerful positioning mechanisms. Row and column widgets simplify some common operations dealing with rows and columns of other widgets. Of course, the usual list of suspects is also available: pop-up menus, list boxes, buttons, and labels. The event mechanism and callback system implemented in the widgets also follows the OSF/Motif approach.

Table 1

IBM Smalltalk Subsystem	Industry Standard
Common Language Data Types	Smalltalk-80 Blue Book and IBM Red Book
Common Language Implementation	Smalltalk-80 Blue Book and IBM Red Book
Common Process Model	Smalltalk-80 Blue Book
Common File System	POSIX.1 and Smalltalk-80 Blue Book
Common Graphics	XWindows
Common Widgets	OSF/Motif

File system

IBM Smalltalk's Common File System provides low-level protocols based on the POSIX.1 standard. They have included the necessary framework for dealing with files in a platform independent manner. The approach has to deal effectively with the usual killer minutiae such as line endings varying between platforms, platforms that have no concept of *drive* versus those that do, and so on. ParcPlace has been doing this stuff for some time, and the IBM Smalltalk approach in this area is no great surprise. In VisualWorks, for example, you would send the named: message to Filename class to instantiate a platform-specific kind of Filename object that can then be used to obtain a Stream on the file. In IBM Smalltalk, you would use a CfsFileDescriptor to write platform-independent code that handles platform-specifics for you automatically.

IBM Smalltalk also provides a variety of file sharing and region-locking facilities based on the POSIX.1 style. Since these types of operations are not supported uniformly across platforms, they offer the ability to determine whether the locking you want is supported. For example, note that

```
CfsFileDescriptor supportsLockType: FMDLOCK
```

will return true or false depending on whether the platform supports exclusive mandatory locks. Platform-specific file errors are available to you if you need them, but most people will use the subset of POSIX.1 error constants available via the CfsError class to handle problems in a platform-independent manner.

Exception handling system

We happen to be big believers in the value of a robust exception handling system, or EHS. As long as you use an EHS to handle exceptions and not to handle the kind of things that happen all the time, it is an absolutely invaluable part of a Smalltalk programmer's tool kit. IBM provides an EHS with IBM Smalltalk that is similar to the VisualWorks EHS in style. Smalltalk/V follows the approach described by Christophe Dony in ECOOP '88, where exceptions are treated as first class objects; i.e., each exception is an instance of some subclass of ExceptionalEvent. ParcPlace and IBM take a slightly different approach to achieve the same result: each exception is an instance of a single class. Where IBM Smalltalk differs from VisualWorks is that IBM Smalltalk provides a default handler, which is a block of code that is evaluated if the exception is not handled. We provide a thumbnail comparison in Table 2.

The asterisk in Table 2 that indicates the lack of Unwind Protection for IBM is there because this feature is not present in the Beta release (it may be added in the final release). Unwind protection requires nonlocal returns to be properly handled by the virtual machine. For example, critical sections implemented by Semaphores usually require unwind protection. No matter how the critical section terminates, you want to signal the semaphore. The syntax for critical sections in both VisualWorks and IBM Smalltalk is aSemaphore critical: aBlock. If you happen to return from the block (i.e., [... some code ... ^self]), the semaphore will never be signaled without unwind protection. Both Smalltalk/V and VisualWorks implement this behavior correctly.

Interface to other languages

IBM Smalltalk provides two interfaces to other languages: platform functions and primitives. Platform functions are similar to Smalltalk/V's ExternalInterface class (for interfacing to DLL's) and VisualWorks' ExternalLibrary interface (used for both DLL's and statically bound code for platforms that do not support DLL's). The C calling convention is the only one currently supported by IBM Smalltalk, and it supports the standard C types you would expect.

Platform functions do not require you to write any code other than the Smalltalk code used to access the external routines. IBM Smalltalk provides two approaches for defining platform functions. In the first approach, you create an instance of a platform function using protocol in the PlatformFunction class. The instance of PlatformFunction that you obtain can be sent messages and treated as any other Smalltalk object. You use the instance of PlatformFunction by sending it messages appropriate to the number of arguments used (i.e., call, callWith:, callWith:with:, etc.). The number of arguments in the call must match the number of arguments in the platform function.

The alternate method of calling a platform function allows you to inline external function calls directly in methods. The syntax, which is the same syntax as PlatformFunction, is embedded in the body of a method, similar to primitive functions. An example is the following:

```
send: msg to: id
    <c: int32 'message' :sendTo int32 int32>
    ^self primitiveFailed
```

This example invokes the function sendTo in the library message. This approach differs from the VisualWorks and Smalltalk/V approaches, since platform functions can reside in any class of the system, not in subclasses of a particular class. Quite useful and extremely slick.

Primitives

Like VisualWorks, IBM Smalltalk supports a rich *user primitive* interface. A user primitive is code that is written in C, typically for performance reasons. This code, unlike a platform or external function, directly manipulates objects and is written for exactly that purpose. Like external functions, primitives can be inlined in methods on any class in the system, and IBM Smalltalk provides syntax to access primitives in any shared library (DLL) by name or number. From the C side of things, you have access to the context of the virtual machine, who the receiver of the message was, and to the objects passed to the primitive method.

Functions that are available inside a user primitive are impressive. Like VisualWorks, but unlike Smalltalk/V, IBM Smalltalk provides functions for object allocation, sending messages to Smalltalk objects from inside of primitives, and pro-

Table 2.

Feature	IBM	Digitalk	ParcPlace
Exceptions as 1st Class Objects	No	Yes	No
Unwind Protection	No*	Yes	Yes
General Usage	[block] when: exception do: [handler]	[block] on: exception do: [handler]	exception handle: [handler] do: [block]

New! Release 2.0 Builds Help Files

Automatic Documentation - Easier Than Ever

*With **Synopsis** for Smalltalk/V Development Teams*

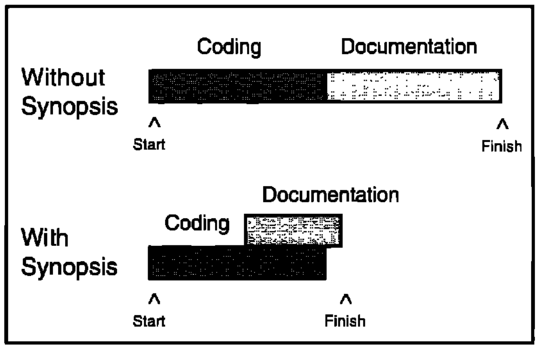
Synopsis produces high quality class documentation automatically. With the combination of Synopsis and Smalltalk/V, you *cut development time* and *eliminate the lag between the production of code and the availability of documentation.*

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 Documentation as Encyclopedia Files or as Help Files for Distribution
- Supports Personalized Documentation and Coding Conventions

Working with Synopsis is *easy.* Install Synopsis and see *immediate results* --- without changing a thing about the way you write Smalltalk code!

Development Time Savings



Products:	Synopsis for Smalltalk/V and Team/V Synopsis for ENVY/Developer
Environments:	Windows, Win32, OS/2
Pricing:	Smalltalk/V $295, ENVY $395 Site licenses available.

Synopsis Software
8912 Oxbridge Court, Raleigh NC 27613
Phone 919-847-2221 Fax 919-847-0650

tecting objects from garbage collection. In addition, IBM Smalltalk provides several other useful functions, including determining the version of the virtual machine and explicitly invoking both the scavenger and garbage collector from inside of primitives.

IBM Smalltalk also provides a nice interface for dealing with asynchronous messages (i.e., interrupts). Because interrupts can happen at any time, the Smalltalk virtual machine must be protected from this behavior and interrupted only at certain check points when the virtual machine is prepared to process interrupts. IBM Smalltalk provides a queue of interrupts that is polled by the interpreter at various times.

Because the interrupt handler could become active at any time (i.e., when the virtual machine is garbage collecting), normal objects cannot be used in asynchronous messages. To handle this situation, IBM Smalltalk allows objects to become *fixed,* meaning that they do not move during garbage collection. This approach means that direct references to fixed objects remain valid and can be passed off to external functions or primitives without worrying about their being moved around during garbage collection.

Finalization

One of the difficulties with dealing with host system resources in Smalltalk has been the finalization of these resources. Finalization is the process of determining when an object is no longer referenced in Smalltalk and is receiving a notification of that fact. Current releases of Smalltalk/V leave finalization to the programmer. VisualWorks has a rich finalization model, providing the programmer with such things as WeakArrays and WeakDictionaries. IBM Smalltalk provides a simpler approach to finalization. If you need to be notified when a particular object is about to go to the great heap in the sky, simply send it the message onFinalizeDo: selectorOrNilOrMessage. When the receiver is about to become garbage, the action supplied by the selectorOrNilOrMessage is performed. If the argument is nil, the message finalize is sent to the receiver. If the argument is a Symbol, then the receiver is sent the message corresponding to the symbol. If the argument is a Message, then the receiver is sent the message (a selector and arguments).

THE DEVELOPMENT ENVIRONMENT

It is often difficult for Smalltalk programmers to mentally separate the differences between the Smalltalk language, the class library, and the development environment. Let's face it, though: if all you had was the language and a workspace, you would not be a happy Smalltalker. Rather than talk about specific tools, we will concentrate on what IBM is supplying for application development, cross-platform development, and deployment of Smalltalk applications. We'll take a little side trip to discuss a new gizmo, TrailBlazer, IBM's shot at improving the basic Smalltalk browsers.

Application management architecture and team development

OTI's ENVY/Developer application management architecture is provided with IBM Smalltalk. Interestingly enough, they provide a separation between the single-user version and the Team version, while maintaining the same ENVY concepts in both. The primary difference between them is that the single-user version uses the standard Smalltalk change log/list approach, while the team version uses a central ENVY repository. In fact, if you have ENVY/Developer, you can connect your Smalltalk/V, VisualWorks, and IBM Smalltalk images *all to the same source code repository.*

Let's do a brief discussion on the application management architecture. ENVY users will find nothing new. For a more complete discussion of ENVY/Developer, see the October 1992 issue of THE SMALLTALK REPORT. We will simply present it as a well-known fact that you cannot do serious Smalltalk development, or have a hope of reuse, without some way to group together the classes and methods that comprise your application. In other words, you need some way to cut across the class hierarchy to identify and maintain the parts that make up your application. ParcPlace uses class categories in their base VisualWorks product to accomplish this goal to a limited degree, and in Smalltalk/V, you can purchase TEAM/V. You can also buy ENVY/Developer as an add-on to either ParcPlace's or Digitalk's base image environment.

Applications and subapplications (classes in the image) are the basis for application management in IBM Smalltalk. All classes must be defined as part of some application. There are many applications provided in the base IBM Smalltalk image, a number of which correspond directly to the subsystems we've been talking about. In addition, classes that are defined in one application can be extended in another, so that the application-specific extensions you might add to something like String are localized to your application without affecting the Kernel application definition and methods. Related applications can also be grouped together into a configuration map, which often corresponds to a single deliverable software product.

Prerequisites are a central part of application management in the ENVY environment. Prerequisites can be somewhat of a religious issue, so we'll try to stay out of it. It would probably be fair to say that nobody likes prerequisites, but if you feel they are necessary, you better have tools to deal effectively with them. In ENVY, they are used to enforce the existence in your image of one application before another that depends on it can be loaded from the library. IBM Smalltalk also extends the use of prerequisites to configuration maps, a welcome addition. Load order of classes within an application is all handled automatically—prerequisites have nothing to do with things at a class level.

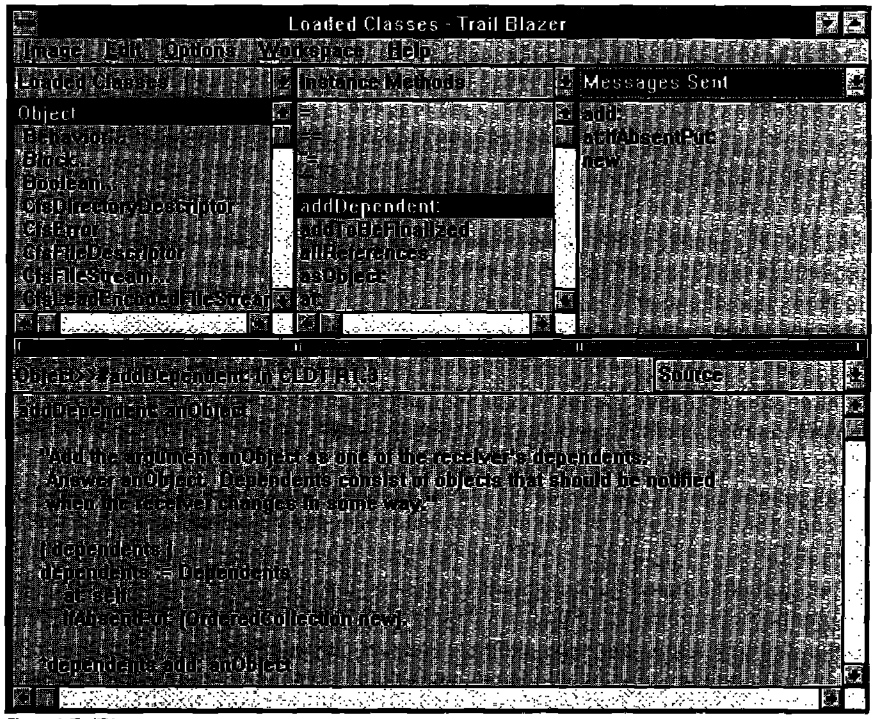
The Team version of IBM Smalltalk comes with the full arsenal of configuration management tools of ENVY/Developer. These tools enforce the concept that all software components, from classes, to applications and subapplications, to configuration maps, are subject to strict version control. Once a component has been versioned, you must create a new edition to be able to change it or any of the components that it contains. The development environment and use of the ENVY repository then let you load configuration map, application, and class versions without wondering whether that bug you fixed last week is still fixed. Of *course* it is, because you versioned it!

Cross-platform development

IBM has done a nice job of providing a Smalltalk with completely portable code between platforms, but which uses host widgets. ParcPlace goes one step further in portability but at the expense of not using host widgets currently. In VisualWorks, you do your development under Windows, save image, put your image on an OS/2 or UNIX machine, and you are up and running. Smalltalk/V in its current release (2.0) is not quite source-level compatible between platforms (Windows, OS/2, and Mac). Some porting is required in almost all cases, although it is usually not too tough. Upcoming Smalltalk/V releases should eliminate the problem.

In IBM Smalltalk, it doesn't matter much which platform you do your development on. If you keep to the common classes and don't go diving into places you're not meant to go, you will be able to move simply to another platform.

Here's how you do it: Because you must organize your code into applications, you simply have to start up a virgin image for the other platform and load your application from the library into the image. ENVY provides ways to handle platform-specific code loading within your application if you require it, too. Then, you're up and running on that platform. Eric Claiberg at ObjectShare, who has been working on the WindowBuilder version for IBM Smalltalk, reports having



```
addDependent: anObject

    "Add the argument anObject as one of the receiver's dependents.
    Answer anObject. Dependents consist of objects that should be notified
    when the receiver changes in some way."

    | dependents |
    dependents := Dependents
        at: self
        ifAbsentPut: [OrderedCollection new].

    ^dependents add: anObject
```

Figure 1. TrailBlazer.

never to change a single line of code between the Windows and OS/2 versions—and ObjectShare does some pretty obscure low-level graphics operations in their work.

TrailBlazer—an improved browser?

IBM provides a new browser called TrailBlazer with their Smalltalk . Despite the fact that there is no documentation for it in the Beta product, we still found it to be a useful tool; this says something significant about it. The idea is to combine a number of the normal browsers into one window, as shown below:

Figure 1 shows something like the Smalltalk/V class hierarchy browser or a VisualWorks hierarchy browser. The drop-downs at the top let you choose what you want to appear in the list. By default, if you open the IBM Smalltalk classes browser, the list on the far right would be blank. We chose to view

"The choice of POSIX.1, X Windows, and OSF/Motif standards for the file system, graphics, and widgets in the system will appeal both to developers and to corporate decision makers"

Messages Sent in it by selecting it from the drop-down menu. As you might expect, the list shows all messages that are sent in Object's instance method **addDependent:**. When you select a message in the list, the TrailBlazer shifts all the lists over one slot to the left and shows Local Implementors on the right. Senders are usually what you're after when you look at the messages in a method, but you don't get another window on your screen. The thick bar under the lists shows (using a blue color) which three slots you are viewing out of the trail you've gone down. To go back to the original three lists (with classes on the left), you just have to click in the bar.

Okay, it's not perfect. It takes some getting used to. The Beta version has some bugs. However, we found we could get our work done at least as effectively with it, but without having to track through 10 different windows at any given time. If you don't like it, IBM lets you use the standard ENVY-style browsers, which most Smalltalkers feel pretty comfortable with, ENVY experience or not.

Application deployment

It often feels as though application deployment strategy is one of Smalltalk's dirty—and certainly well kept—secrets. The usual ParcPlace approach is simply to carve away, or strip, all the classes and methods you do not need from the base image before deploying. Digitalk nicely separates development classes from base classes via DLLs, which helps somewhat, and they allow you to build object libraries to which you can bind in runtime. ENVY/Developer users under Digitalk Smalltalk/V Windows and OS/2 (but not WIN32) have had the luxury of using the ENVY Packager. The concept here is that because the image is built from applications, you should be able to build one from scratch by specifying the applications you want in it and leaving out the ones you don't want. To build a really small image, you need some more refined tools, too, and these are part of the Packager.

Not too surprisingly, you get the Packager as part of IBM Smalltalk. The result is that you have a tool to help you deploy a real Smalltalk application that uses as little footprint as possible, and you can track via ENVY's configuration-management facilities to see just how you did it.

What else is interesting?

The virtual machine is a byte code interpreter and does not use dynamic compilation as do ParcPlace and Digitalk (in their WIN32 and OS/2 releases). Consequently, IBM Smalltalk is slower than either of these other products on almost any of the standard Smalltalk benchmarks you might use. However, because of the clean architecture and implementation of the graphics and the use of host widgets, the user interface doesn't feel slow at all. IBM says, "We're working on it." As the adage recommends, make it *work* first, then make it work *right*, then make it work *fast*. It already works right, so we'll have to wait a little while for it to work faster.

The Beta release under OS/2 and Windows comes with a 50 Mb ENVY library, chock full of things. No wonder it's distributed on CD! The library includes many examples that will be a valuable aid to those for whom X Windows and OSF/Motif are new, even if they are expert Smalltalkers. Apparently, IBM is considering providing the full documentation set on the CD as the standard way to distribute it.

With such a large class library, it seems a shame that so much source code is missing. In the Beta release, no source code is provided for the compiler, common widget, OS widget, and the common graphics subsystems. The lack of source could limit the usefulness of these very useful areas of the class library and prevent further specialization and host operating system integration. It is possible that the situation may change for the final release.

CONCLUSIONS

IBM has entered the Smalltalk market with a product that compares well with the established vendors in most areas and is in some ways superior. The choice of POSIX.1, X Windows, and OSF/Motif standards as the basis for the file system, graphics, and widgets in the system will no doubt have appeal both to developers and to corporate decision makers. The primitive and C language interfaces are innovative and provide extremely useful functions for developers. The system uses host widgets on all platforms and shows good user-interface speed in spite of a relatively slow virtual machine. In addition, IBM bundles the team version of the product with the application management and team development tools of ENVY/Developer. All these factors combined show that IBM has placed itself in a position to be reckoned with, not just from a marketing point of view, but from a technical point of view as well. ♀

Steve Harris and Hal Hildebrand founded Polymorphic Software, Inc., in March 1993. Since that time, Polymorphic has released Tensegrity, an object-oriented database for Smalltalk, as well as other Smalltalk tools and frameworks aimed at increasing developer productivity and reducing time-to-market of Smalltalk applications. They can be reached at 75010,3075 on CompuServe or at loki@polymorf.win.net on the Internet.

A trace logger

Alec Sharp & Dave Farmer

This article describes our implementation of a TraceLog mechanism that we have found useful in the development and support of our product. We describe various features of the TraceLog and show some of the code. Embedded in the code in *italics* are explanations of some of the more interesting aspects of the code. We also take detours on occasion to go over things that we found interesting, and that may be useful to the reader. This is rather like the way we developed the TraceLog, following a basically straight path, but taking little detours now and then as something aroused our curiosity.

Our current product consists of a lot of UNIX processes running in the background, each one with a specialized task to perform. Each is forked by a single parent process and communicates with other processes via standard UNIX IPC mechanisms (sockets and shared memory). Once running they continue until sent a signal from the parent. We are writing new functionality for the product in Smalltalk, using ObjectWorks 4.1 from ParcPlace.

Our initial model of the TraceLog was that sending a SIGUSR1 signal to our Smalltalk UNIX process would toggle "packet tracing." When packet tracing is enabled, we print to a log file all data that is sent to a serial device and received from the device. This ability is very important to our support people in helping track down problems at customer sites.

We immediately decided to extend the model to allow us to trace entry into methods, and to support the logging of debug statements from within a method. We extended the model further as we needed new capabilities for our own debugging, as we thought of fun enhancements, and as we came across ideas and features that we thought would make for a slick demo of the TraceLog.

Here's a listing of the parameter file that we now use to control logging; anything after a # is a comment. If the file does not exist, then at startup, logging is off, and when a user sends a SIGUSR1 signal, packet tracing is toggled:

```
# TraceLog parameter file.
#
#resetLogFile              # Open a new, empty log file

stackDepth:      1         # 0 means show whole stack for
                           # each method
traceDebug:      true      # Trace Debug statements?
traceEnter:      true      # Trace Entry to methods?
tracePackets:    true      # Trace packets to and from device?
showTimestamp:   true      # Show time of each logged message?
showSource:      false     # Show source code?

# Include or exclude specified classes. Format is:
#            includeClass: <classname>
#            excludeClass: <classname>
# An include list takes precedence over an exclude list.
# If there are no lists, all classes are logged.

exit                       # Exit from parameter file processing

excludeClass: LmSocket
excludeClass: LmOutput
excludeClass: ChannelManager
```

When we read this file, we ignore comments and blank lines. All other lines are treated as Smalltalk code and executed using the perform:withArguments: method, shown later. The exit statement in the file causes the exit message to be sent and this stops us reading any more of the file. So, in this example, none of the excludeClass: messages will be sent. If the exit was missing or commented out, the excludeClass: method adds the specified class to a Set. We have two Sets, classesToInclude, and classesToExclude, both initialized to nil. If classesToInclude is not nil, classesToExclude will be ignored.

Rather than show all the code, we will just show some of the key points, and leave it to the reader to fill in the gaps. To start, let's go over how we set up our signal handler and check to see if we've received a signal. Our signal handler is written in C, and so we use the CPOK extensions; our C interface object is stored in a pool variable called CInterface.

We are limited by the ability of C and Smalltalk to communicate with each other—Smalltalk can call C, and if you want to, the C function could immediately call back into Smalltalk. However, a C function cannot call Smalltalk asynchronously; i.e., a C function can only call Smalltalk if it was first called from Smalltalk. The problem is one of memory locations shifting around under you. So the way we do this is to have our signal handler modify a Smalltalk heap variable, then check this variable every time our product gets a new request to process. We set up the signal handler in the initialize method of one of our application objects.

```
initialize
        ...
    self ptrTraceReset: 0 gcCopyToHeap.
    CInterface initLogSignalHdlr: ptrTraceReset.
checkTraceReset
    "Check to see if the signal handler has been invoked. If so, reset
    the trace mechanism."
```

```
        self ptrTraceReset contents = 1
          ifTrue:
            [ptrTraceReset contents: 0.
```

Note that for performance, after sending reset, we set the a pool dictionary variable, Trace, to true if tracing is on. Our instance of the TraceLog has been stored in the pool variable Log.

```
        Log reset.
        Trace := Log traceEnter or: [Log traceDebug]]
```

Here's how we open and read the parameter file. Once the file is open, we loop through it. As we loop through, if we come across a line saying exit, the exit method returns true, terminating the loop and preventing the rest of the file being read.

```
  readParameterFile
      | stream file |
      file := self parameterFileName.
      file isReadable
          ifTrue:
              [self tracePackets: false.
              stream := file readStream.
              self loopThroughFile: stream.
              stream close]
          ifFalse:
              ["If there is no readable parameter file,
              we should simply toggle packet tracing"
              self tracePackets: self tracePackets not].

  loopThroughFile: aStream
      | line exit |
      exit := false.
      [aStream atEnd or: [exit == true]]
          whileFalse:
```

Why do we explicitly test exit against true, *rather than simply saying* or: [exit]? *Because it's easier to allow the methods we perform to return* self *rather than requiring them to return* false, *and you can't say* or: [exit] *if* exit *is not a Boolean.*

```
          [line := aStream upTo: Character cr.
              exit := self processLine: line].
```

For each line we read, we strip off any comments: comments start with a #. Then we break the line into white space separated words. The key part is that each line is in valid Smalltalk syntax so we can simply perform the line, using the first word as the selector and the rest of the words as the arguments. We wrap the perform in a general exception handler so that we trap any errors—specifically: selector is unknown, and the number of parameters is incorrect for the selector. Here's the code that processes each line.

```
  processLine: aLine
      | array |
      array := (aLine copyUpTo: $#) asArrayOfSubstrings.
      array size > 0
          ifTrue:
              [Object errorSignal
                  handle: [:ex | self logMsg:
                                  'Invalid line: ' , aLine]
              do: [^self perform: (array at: 1) asSymbol
                  withArguments: (array copyFrom: 2
                                          to: array size)]]
```

Objects Everywhere!

Why settle for hybrid implementations when you can have the real thing? JumpStart is the leading provider of solutions and training programs for pure object systems using Smalltalk and the GemStone(tm) ODBMS. We also specialize in deploying IBM Smalltalk(tm) and VisualAge(tm) applications.

Ask about our Corporate Educators Program.

Manufacturing
Process Control
Network Management
Pharmaceutical
Client-Server IS Systems

JumpStart

Certified Service Partners with:

SERVIO

919.460.1583

Copyright 1994, © JumpStart Systems, Inc.

Let's take a couple of detours here. The perform: family of selectors (perform:, perform:with:, ..., perform:withArguments:) allow you to dynamically create messages in your code. The perform: family are messages to which Object responds. Regardless of which one you use, eventually perform:withArguments: is sent, with an array of size zero or greater, and a primitive is invoked.

The perform family can help eliminate the need for a switch/case statement or the need for multiple [conditional] ifTrue: [block] statements. For example, in our application, we get back status codes from a serial device. We could check each possible value and do something like:

```
  statusCode = '0027' ifTrue: [^do something].
  statusCode = '0028' ifTrue: [^do something].
  statusCode = '0029' ifTrue: [^do something].
```

However, we can eliminate this checking for matches very simply. Instead, we create methods called message0027, message0028, etc., then we do:

```
  Object messageNotUnderstoodSignal
      handle: [:ex | handle invalid statusCode]
      do: [self perform: ('message', statusCode printString) asSymbol].
```

An alternative way to do this last piece of code is to see if we understand the message.

```
  selector := ('message', statusCode printString) asSymbol.
  (self respondsTo: selector)
      ifTrue: [self perform: selector]
      ifFalse: [handle invalid statusCode].
```

In terms of performance, handle:do: is marginally faster when the message is understood, but in our timings it was about seven times slower than the respondsTo: technique when the message was not understood. We feel that the handle:do: technique is more elegant though.

The other detour is to take a look at the message asArrayOfSubstrings. This is a message that we added to the system class String. Here's the code; note that it uses findFirst:startingAt:, which we also added to String by an easy extension to findFirst:—

```
asArrayOfSubstrings
    | first last collection |
    collection := OrderedCollection new.
    last := 0.
    [first := self findFirst: [:ch | ch isSeparator not]
            startingAt: last + 1.
    first ~= 0]
        whileTrue:
            [last := (self findFirst: [:ch | ch isSeparator]
                        startingAt: first) - 1.
    last < 0 ifTrue: [last := self size].
    collection add: (self copyFrom: first to: last)].
^Array withAll: collection.
```

Back to the TraceLog. We have two functions that log entry into methods. Here are examples of them.

```
Trace ifTrue: [Log enter].
    Trace ifTrue: [Log enter: 'here is some value in string form'].
```

The enter method simply sends self enter: nil. So let's look at the enter: method:

```
enter: aString
    | context currentDepth |
    traceEnter ifFalse: [^self].
```

Next we pop the context until we are no longer in the TraceLog object. A context knows what object it is in (the receiver), what message was received, and what object sent the message (the sender). So we can migrate out through the senders of messages until we find a sender that is not our TraceLog instance; i.e., not self*. The variable* thisContext *is classified as a special variable, along with* self *and* super *and contains information about, surprise, the current context. It can be interesting to put a* self halt *in a method and inspect* thisContext*.*

```
context := thisContext sender.
[self = context receiver]
    whileTrue: [context := context sender].
```

The reason for popping the context is that we may be called from the enter *method, and we don't particularly want to record TraceLog as the class that is trying to log a message. What we end up with is the class from which the* enter *or* enter: *message was sent. Once we know this class, we can check to see if it is one of the classes we want to log. The code that checks whether to log a class is listed after this method.*

```
(self logThisClass: context)
    ifTrue:
        [currentDepth := 0.
        (self logFile) cr; cr; nextPutAll: 'Enter'.

        self timestamp.
        aString notNil ifTrue: [(self logFile) nextPutAll: aString].
```

We now stay in a loop, printing out the method we are in then popping the stack, until either we are at the end of the stack, or we have satisfied the stack depth condition in the parameter file.

```
[context notNil
    and: [context receiver notNil
        and: [currentDepth < self stackDepth
            or: [self stackDepth = 0]]]]
    whileTrue:
        [(self logFile) cr; nextPutAll: context printString.
```

Here, we print the source code of the method we are in, assuming the parameter file tells us to do so, i.e., we print the source code for the context. We haven't found it particularly useful, but it makes an impressive demo. Interestingly, it will show the source code even if the sources are not available. It does this by decomposing the byte-codes and using t1, t2, etc., for the variable names.

```
self showSource
    ifTrue: [(self logFile)
        cr;
        nextPutAll: context sourceCode].
```

Now get the context of the sender of the message that generated the context being printed.

```
context := context sender.
currentDepth := currentDepth + 1].
self logFile flush]
```

Here's the code that checks to see if we should print trace information for the current method. I.e., is the class of the object which sent the message one of the classes we should be logging?

```
logThisClass: aContext
    | thisClass |
    (self classesToInclude isNil
     and: [self classesToExclude isNil])
        ifTrue: [^true].
```

The next line looks rather strange. What we are doing is taking the context of the method that sent the enter or enter: message and getting its class. aContext receiver printString = 'a Thing' *if the message was sent by an instance of Thing, and "Thing" if it was sent from the class* Thing. aContex receiver class printString = 'Thing' *if the message was sent from an instance of* Thing, *and* Thing *class' if it was sent from the class* Thing. *So the line we have ensures that we end up with* Thing'*, whether the sender is a class or an instance method.*

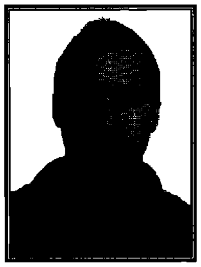
```
thisClass := aContext receiver class printString
                copyUpTo: Character space.
    self classesToInclude notNil
ifTrue: [^self classesToInclude includes: thisClass]
ifFalse: [^(self classesToExclude includes: thisClass) not]
```

What is the final outcome of all this code? Here's an example of the log file, showing output from enter, enter:, and debug:, with stackDepth set to 1, and with showTimestamp set to true. We don't print the date because if logging is on, the first thing logged is the date and the list of trace parameters in effect.

```
Enter(2:26:57 pm):
LmInput>>getRequestUsing:


Debug(2:26:57 pm): acs=0, vsn='RB1400'
LhMoveRequest(LhRequest)>>initialize:
```

continued on page 32



Bob Brodd

Collection protection

"To accessor or not to accessor?" A good question, but one that I answered for myself five years ago. My mentors used them exclusively and I have been addicted to them ever since. My question is not should I use them or not, but how can I use them safely?

THE PROS

I suppose at first I used accessor methods without a whole lot of thought. Now I have five years of Smalltalk under my belt and I have experienced the good, the bad, and the ugly sides of accessor methods. When I add up all the pluses and minuses though, the pluses win out. Here are the main reasons why I like the following accessor methods.

- **Robustness of the system under development.** Accessor methods allow me to use laissez-faire initialization techniques. This makes my objects more resilient since variables (instance, class, class instance) initialize as needed. The advantages of this technique include the following:
    1. Initial values for variables are defined in a single location. This removes the chances for (re)initializing variables improperly (that is, other methods of the object can set the variable to nil instead of to an object of a particular type). Additionally, if you decide to change the type of object held by a variable, you only have to make the change in a single location.
    2. Objects held by variables are not created until they are needed, if ever.
- **Facilitation of business rules and access control.** System requirements may include actions that must take place when object states change. These include security restrictions that may make variable access restricted in certain circumstances. Accessor methods can support these types of requirements.
- **Less coupling.** Changes to variable storage types should not affect the class' methods or its subclasses' methods.

THE CONS

I understand that there is a concern that providing and *blindly* using accessor methods violate the encapsulation benefits of object-oriented systems. One of the biggest mistakes I have made (and seen many others make) is to manipulate the contents of another objects variables instead of letting the object manage it on its own. This usually occurs with variables that are of the Collection types, but occurs with other object types as well.

A typical example is when an object gains access to another object's instance variable that contains a collection and simply adds an object to it, or removes one from it. Lets say we have a HierarchicalObject class that has an instance variable children that holds a collection of its immediate descendants. A typical get accessor method for children might look like the following:

```
HierarchicalObject Class
    children
        "Answer my collection of direct descendents"
        (children isNil)
            ifTrue: [ children:= OrderedCollection new].
        ^children
```

At this point, lets assume that the only task to perform when adding a child is to add it to the children collection. The Clients of this class might write the following code to do so:

```
...
aHierarchicalObject children add: child.
...
```

This may be acceptable for the current implementation, but it becomes a difficult situation if there are specific tasks that must occur when adding and removing children. For example, lets say HierarchicalObject's implementation changes so that it includes an instance variable parent to hold its immediate ancestor. Whenever a child is added to a parent, the parent of the child also needs to be set. Now Clients must change their code to look like:

```
...
aHierarchicalObject children add: child.
child parent: aHierarchicalObject
...
```

The behavior of the object ends up dispersed among other objects in the system. The object that owns the children has no control over its own variables. This is a complete violation of encapsulation and a potential maintenance nightmare as well. This of course could be avoided if the children collection is not accessible to Clients.

A SOLUTION

The optimal way to solve this problem is for the Smalltalk vendors to support the notion of public and private variables and methods in ways similar to C++. In the meantime, how do we gain the benefits of accessor methods without suffering the typical abuse that is associated with them? One solution is a technique I am developing for use in our designs at Hatteras Software, Inc. that we call *collection protection*. The name is a little misleading as the technique applies to protection of all state data for an object, but collections are the most abused.

The technique provides for the use of accessor methods but reduces the chances of accidental encapsulation abuse. It does so by using a combination of public and private methods. Here are the key points to collection protection:

Bob Brodd is Vice President of Products at Hatteras Software, Inc., a company that specializes in helping other companies use object technology effectively. He welcomes questions and comments via email at 73021.3235@compuserve.com or by phone at 919.319.3816.

Project Practicalities

The private method

Kent mentions in his article that simply making accessors private is not a sufficient solution to solve the encapsulation problem because "anyone can invoke any method (and will, given enough stress)." I agree with this statement. After all, making a method private is nothing more than placing the word private at the beginning of the method comment and when available, placing the method in a private protocol. I cannot keep up with my own classes private and public methods, much less anyone elses!

Once I realized this, I began thinking about ways to make private accessor methods obviously private. First I tried using the prefixes public- and private- for appropriate protocol names. This is very useful, but it does not make it obvious when referencing the method in code. I then pursued different ideas for naming the private accessor methods so that they were somehow different from others. I thought about using private as a prefix for each private accessor method name. This was certainly obvious, but is rather cumbersome and does not read well. Then I tried some other variations such as a prefix of *basic*, which is used in some of the base Smalltalk code. This is better, but as noted, it is being used for another purpose.

I was convinced that this was the best way to solve the problem, so I tried several other prefixes until I stumbled upon the word *my*. It seemed too simple at first, but the more I thought about it, the prefix my seemed like a reasonable solution. As it turns out, my works because it reads both naturally and unnaturally. Take a look at the following code that uses the example of a Store that has employees:.

```
Store class
    myEmployees
        "private - Answer my collection of employees"

    ( ^employees isNil )
        ifTrue: [employees := OrderedCollection new ].
        ^employees

    myEmployees: aCollection
        "private - Seth my collection of employees to aCollection"
        employees := aCollectionOfEmployees
```

My reads naturally when an object is accessing its own variables because the object does indeed own the variables. My is an appropriate term for this frame of reference. For example, an object wants to access its own collection of Employee objects through its accessor method named myEmployees. A method within the object doing this would look something like:

```
:...
list addAll: self myEmployees
...
```

On the other hand, my reads very unnaturally when code in another class tries to access the object's variables through its private accessor methods. A method within another class doing this would look something like:

```
...
list addAll: anotherObject myEmployees
...
```

In this case, the meaning of the statement becomes confusing.

continued on page 32

Database Solution for Smalltalk/V

A class library for ODBC Database Access

ODBTalk

- ODBC 2.0 support
- Automatic class generation
- Native data type support
- online help, source included, no runtime fees

Available for Win16, Win32s, Win-NT, OS/2 and PARTS

"... *simple but elegant* ..." - Australian Gilt Securities

Client Server Solution for Smalltalk/V

A class library for Windows Sockets Development

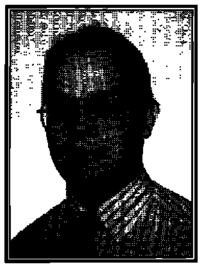
Socktalk

- UDP and TCP Sockets
- Synchronous and asynchronous support
- sample code for remote disk browser app
- online help, source included, no runtime fees

Available for Win16, Win32s, Win-NT

LPC Consulting Services
Tools for the Smalltalk developer

Tel: 416-787-5290
Fax: 416-797-9214
CompuServe: 73055,123
Internet: lucc@tor.hookup.net



ALAN KNIGHT

Literals

The discussion this time starts with a simple, commonly asked question, and takes it to extreme lengths. While many of the proposed solutions aren't things you would use in practice, they illustrate some interesting Smalltalk techniques and areas of confusion.

The question is how to create an array of points in a method. The obvious:

```
#(6@3 12@4 13@79)
```

won't work. It yields an array of integers and symbols, like this:

```
#(6 #@ 3 12 #@ 4 13 #@ 79 )
```

The numbers in the expression are properly interpreted, while the @ operators turn into symbols. Clearly, the compiler needs more detailed instructions. How about if we use brackets to tell the compiler how to group the numbers and operators:

```
#((6@3) (12@4) (13@79))
```

Unfortunately, this doesn't work either. It gives us an array containing three subarrays, each of which contains two integers and a symbol:

```
#(#(6 #@ 3) #(12 #@ 4) #(13 #@ 79))
```

POINTS AREN'T LITERALS

The real answer is that you can't do this directly. The Smalltalk compiler treats certain "literal" strings specially, and creates objects at compile-time which are embedded into the compiled code. This literal syntax is very important, since it would otherwise be awkward to generate many of these objects. Advanced readers can think about how to generate numbers if there were no literal syntax. It's awkward, but certainly possible, and I'll discuss it briefly at the end of the column.

Some examples of literals are

3	(an Integer)
3.14159	(a Float)
3.1415926535898d	(a Double)
$b	(a Character)
'hello world'	(a String)
#helloWorld	(a Symbol)
[ObjectMemory quit]	(a Block)
#(2 aSymbol #aSymbol 'a String')	(an Array containing other literal objects)

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

Of these, we are most interested in the last one, creating a literal array. Note that "words" (anything that would be a legal variable name) inside a literal array are treated as symbols, whether or not they have a # in front of them. That's why the @ sign was turned into a symbol in the previous examples.

The other important thing to note is that points are not one of these literal constructs. When I write 4@10 in my Smalltalk code, it's not interpreted as a literal point, but as two literal integers and a message send. The operator @ is treated exactly the same as the operator + in 2+2. Both are messages sent (at runtime) to numbers which generate a new object.

WHAT CAN WE DO?

If we can't make a literal array of points, then what's the easiest way to create a non-literal array of points. There are quite a few, but for small arrays the easiest one is obvious.

```
Array with: 6@3 with: 12@4 with: 13@79
```

Unfortunately, that only works on arrays with four or fewer elements. For larger arrays we could make smaller arrays and concatenate them:

```
(Array with: 6@3 with: 12@4 with: 13@79 with: 14@7) ,
    (Array with: 7@33 with: 9@13).
```

but concatenation is expensive if the arrays start getting large. We could create an OrderedCollection instead, since that's easy, and we can always convert it to an array if we really require an array and are willing to pay the cost of conversion:

```
(OrderedCollection new: 3)
    add: 6@3;
    add: 12@4;
    add: 13@79;
    yourself "or asArray"
```

If we want to avoid the cost of conversion, we could always use a stream. Streams are most commonly used for strings, but they work with fine arrays:

```
WriteStream on: (Array new: 3)
    nextPut: 6@3;
    nextPut: 12@4;
    nextPut: 13@79;
    contents
```

Transforming the literals

All these mechanisms are pretty simple, but we're getting further and further away from the clear, simple literal syntax. We can get closer to that by using valid literal syntax and then transforming the results. **Alan Reider (alanreider@sensenet.com)** suggests two possible forms:

```
#(10 78 90) with: #(45 10 34) collect: [ :x :y | x @ y ]
```

This requires us to define a with:collect: method, similar to the common with:do: iterator. The with:do: method iterates over matching collections, executing its argument block with corresponding elements from both collections. The with:collect: method does the same thing, but also accumulates the results. While this is an interesting technique, for this case it doesn't seem as clear as the alternative using the normal collect: method:

```
#((10 45) (78 10) (90 34)) collect: [ :each | each first @ each last]
```

Rather than creating a literal array of points, this creates a literal array of two-element arrays of integers and then at runtime, builds a collection of points based on the arrays. This is quite readable, since the *x* and *y* coordinates are kept close together.

Performing literals

If we start making use of some of Smalltalk's meta-facilities, we can get very close to the literal form. Björn Eiderbäck (bjorne@cyklop.nada.kth.se) suggests making use of the perform: facility. Since the "@" operator is interpreted as a symbol, we can get Smalltalk to send it as a message. In fact, this will

"I like going around the image changing the class of objects just to see who objects."

let us write an almost literal array using any expression that has only binary selectors and literal constants:

```
| s w |
s := #( 10@45 78@10 90@34 67 @ 12 98 *2 34 + 44) readStream.
w := WriteStream on: Array new.
[s atEnd] whileFalse:
    [w nextPut: (s next perform: s next with: s next)].
w contents
```

By converting this code into appropriate instance methods, and adding a facility for handling nested arrays, the syntax can be made very simple:

```
#(10@45 (90@34) (67@12 (1@1 2@3) 99@100) 98*2 34+44 (1+2))
convert
```

Use the compiler

Of course the ultimate in metafacilities and general solutions is to invoke the compiler. Alan Lovejoy (lovejoya@netcom.com) suggests:

```
(#('44@122' '18@54') collect: [:string | Compiler evaluate: string])
```

which can also be given a more convenient syntax by defining appropriate evaluation methods for strings and collections. While it's certainly clear that we're creating points, this is extremely inefficient and is also not allowed in a normal application (Most Smalltalk vendors don't allow you to include the compiler in a run-time system). This can be made somewhat more efficient and less general, as suggested by Björn Eiderbäck:

```
#('10@45' '78@10' '90@34' ) collect: [:x | Point readFromString: x]
```

(Note that readFromString: is ParcPlace-specific.)

Change the compiler

One step beyond using the compiler is changing the compiler so that it does generate the literals you want. **Eliot Miranda (eliot@ircam.fr)** writes:

I modified the 2.3 compiler to evaluate expressions in { }s at compile time. So you could write for example:

```
{Array with: 10@14 with: 12@16 with: 14@16} at: index
```

Within { }s self is bound to the class in which the code is compiled. Alas, the ParcPlace 4.0 compiler doesn't remember the class, so I never bothered to implement it in 4.0. It took about four hours to do. The major tweak is to include all the literals in all the compile-time expressions in the resulting method so that e.g., browsing senders still works even though they don't occur in the compiled code.

This, however, is a little bit beyond the ability of the average Smalltalk programmer. I know it would take me more than four hours.

If you're lucky, though, someone else will have modified the compiler for you. Users of GemStone and SmalltalkAgents are in luck, as both of these dialects apparently have a construct like this, although they both build the array at runtime instead of compile-time (compiler optimizations may change that for specific cases).

Class changing

By this time, the discussion had become something of a contest. Jan Bottorf (janb@netcom.com) writes:

It seems amazing how many different ways we have come up with to do this simple thing. Maybe we should have categories: the clear source code people, the highest performance people, the most encapsulated people, the special compiler people.

He also contributed a particularly sneaky mechanism, using one of the lesser-known features of ParcPlace Smalltalk:

```
#((10 45) (78 10) (90 34)) collect:
    [ :each | each changeClassToThatOf: 0@0]
```

Yes, that's right, you can actually change the class of an object. It breaks encapsulation horribly, since you are responsible for knowing that the number of instance variables match, and understanding the mapping between them. If you're careful, it's a very powerful technique that can do all kinds of nifty stuff (q.v. THE SMALLTALK REPORT, 2(9):4 and THE SMALLTALK REPORT, 2(6):13). Used in an undisciplined way, it can cause some extremely confusing bugs. Anything that changes literals in a compiled method is likely to be confusing.

Nevertheless, this is a very powerful technique. Jeff McAffer (jeff@is.s.u-tokyo.ac.jp) proposes a fairly structured way of using it, the "self-initializing method." The first time it's called, it actually modifies its own internal storage so that it can return the computed result without extra computation the next time. Given the level of system hacking involved, I'm not sure this is really better in practice than caching the array of points in a class variable. It is, however, an extremely neat hack, and points the way to some of the amazing things that can be achieved by exploiting Smalltalk's reflective capabilities.

Unfortunately, <the previous solution> has two drawbacks. 1) #collect creates another array. Since you are already bashing the sub-arrays why even bother to #collect:. Just run through and change the class and voila, an Array of points. 2) If this is in a method it will only work once since it hacks and bashes the class of the literal elements. Well, OK, in this case the second run will just change a bunch of Points to be Points but that is wasteful.

To address these, I propose a self-initializing method as follows:

continued on page 30

CLIENTSERVER Tour

Register and receive a FREE copy of Borland's new Delphi product—a $495 value.

January 12-13	NEW YORK
January 17-18	WASHINGTON, DC
January 19-20	CHICAGO
January 23-24	SAN FRANCISCO

Two Days of Training—A New Era of Information Management

Schedule of Events

DAY I	
8:00-9:00	Registration/Breakfast
9:00-9:30	Welcome Address
9:45–11:45	Morning Classes: C/S Overview
	[illegible]
11:45–1:45	Luncheon & Borland Presentation
2:00–4:00	Afternoon Classes: Making the Change to Client/Server
	[illegible]
4:00–5:00	Q&A

DAY II	
8:00-9:30	Breakfast
9:45–11:45	Morning Classes: C/S in the Real World
	[illegible]
11:45–1:45	Luncheon & Borland Presentation
2:00–4:00	Afternoon Classes: Strategic Considerations in the Transition to C/S
	[illegible]
4:00–5:00	Q&A

Speakers

Management Track Speaker Jim Stikeleather is the Product Review Editor and C/S Workbench Columnist for *Client/Server Developer* Magazine. He is currently the Technical Partner of the Technical Resource Connection, a client/server and O-O technology implementation firm.

Jim Stikeleather

Charles Bowman

Technical Track Speaker Charles Bowman is an independent consultant, author and lecturer specializing in GUI, relational and object-oriented technologies. Mr. Bowman is a regular columnist for *Client/Server Developer* magazine and [illegible] Journal.

The **Client/Server Applications Tour** is a 2-day training and education event making stops in 4 cities across the U.S. All classes are objective and product neutral. The tour is designed to provide attendees with practical guidelines to assess and develop realistic goals for their own client/server projects.

Divided into a technical and management track, this seminar offers practical solutions for implementing a client/server architecture within corporate environments. Whether you are a software manager or part of the technical development team, you'll walk away from these two days of intense training with a complete framework for implementing and managing client/server solutions which meet your particular business needs.

Who Should Attend

Anyone exploring the corporate benefits of client/server technology, including:

- Programmers
- Developers
- Designers
- Software Engineers
- System Analysts
- Project Managers
- Project Leaders
- CIOs
- MIS

[illegible] Ma[illegible]

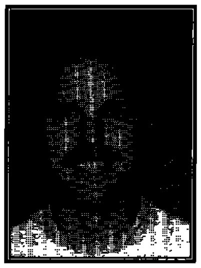
Presented by:

SIGS CONFERENCES

Co-sponsored by:

Borland

CLIENTSERVER DEVELOPER



KENT BECK

Architectural Prototype: Television Remote Control

Now, where was I? Oh, yes. Last issue I talked about my philosophy of testing and presented a framework that supported writing unit and integration tests. But before that, I was talking about how to use patterns. I have spent a couple of issues designing the software to run a television and remote control, using patterns to guide every design decision. Here are the CRC descriptions of the objects we found, and the patterns that created them:

By the time I have this many objects designed, especially with clients who aren't really familiar with objects all the way through implementation, I find that most people's understanding of the design is so vague as to be actively dangerous. Everyone is thinking of a different implementation (or trying hard not to think of any implementation at all). It is about at this point that I like to write a quick sketch of the architecture in Smalltalk to bring everyone's focus back to something concrete.

Sometimes I call this a "spike," because we are driving a spike through the entire design. We are not searching for completeness. Instead, we want to illustrate the kinds of responsibilities accepted by all the major objects in the system. Because people variously associate "spike" with volleyball, railroads, or dogs, I have begun using "architectural prototype" to describe this implementation.

What does all this architectural prototype stuff have to do with patterns? I have two answers. First, Architectural Prototype is a pattern, but at a completely different level than most of the patterns I have discussed in this column. See below for the pattern itself. The second answer is a bit more complicated. I never design objects without wanting to see them implemented. Especially with designs guided by patterns, I find that the translation of the design into code is both straightforward and enlightening.

PATTERN: ARCHITECTURAL PROTOTYPE

When do you put design ideas into code?

In the beginning, programmers just sat down and wrote their code. Any preparatory work was either entirely mental, or scratched on the back of sheets of line printer paper. Experience soon showed that while this approach worked for smart people working on small projects, larger team efforts required coordination before coding to avoid the enormous costs of revising already running code to match data structures, resolve names, and ensure reliable computations. Software engineering has a nearly unbroken record of pushing more and more work "up front," before coding begins.

The urge to resolve all possible issues before coding rests on good economics. The later in development a problem is discovered, the more it costs. A dogmatic adherence to "design first, then code" ignores two very important issues.

First, the goal of "up-front" development is to set up clear, effective, concise communications between the members of the team. A good design creates a shared vocabulary and mindset among the people who will have to implement, manage, and use the resulting system. Design notations both help and hinder this process. Because they are abstract, they encourage discussing essentials without descending into unimportant implementation details. That abstraction cuts both ways, though. The meaning of design documents are subject to human interpretation. It may be months before it is apparent that a team's understanding has diverged.

Second, code is no longer the boat anchor it used to be. Modular development supported by a modern, object-oriented language, and programming environment results in code that is less expensive to write and easier to modify, even late in development, than were the products of earlier generations of programming languages. The cost curve supporting "design first, then code," has changed.

Together, these two points demonstrate that early coding is both necessary, to overcome the vagueness of design notations, and practical, because doing so will not invoke inordinate costs.

What code should you write early? The same constraints apply to early code that apply to early design. You'd like to make decisions with far reaching effects. Big decisions are the ones that will are most important to communicate early. You'd like to avoid making decisions with small effects. Their presence in the code will obscure the important points you are exploring.

Implementation a system when you have enough objects that their interaction is no longer obvious to everyone. Use simple algorithms and data structures. Implement only a single variation where there will be many in the final system.

You may also need an Interface Prototype to aid in communication with users.

THE TELEVISION PROTOTYPE

The goal in an architectural prototype is to demonstrate and communicate the architecture as simply as possible. I was talking to Ward Cunningham the other day and he said, "My job is to write five method classes." That is exactly what I am talking about.

The user interface to an architectural prototype should be extremely simple. If you can run it from a Workspace, so much the better. The goal of the prototype is not to demonstrate a whizzy interface, but communicate from programmer to programmer. User interface code is some of the hardest to get and

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

February 21–24, 1995

Omni Park Central
New York, NY

SMALLTALK SOLUTIONS 95

Where All the Talk is Smalltalk

Corporate Sponsors

[illegible]

• Knowledge Systems Corp.

[illegible]

Finally, a vendor-independent conference dedicated to all Smalltalk users. Focusing on the practical application of Smalltalk in its dialects, **Smalltalk Solutions '95** is an opportunity for the entire Smalltalk community to network, share innovative strategies and programming tips, and stay up-to-date on the latest tools and techniques.

Learn from the Smalltalk Experts

The educational program has been designed in conjunction with the Technical Conference Chair John Pugh, editor of *The Smalltalk Report*. The 4-day conference offers over 30 intensive classes ranging from beginner to advanced, all taught by experienced and well-respected Smalltalk experts.

You'll come away with new insights on language advances, usage tips, project management, analysis and design techniques, and insightful, practical applications. Specific class tracks focus on Team Programming, Analysis and Design, User Experiences/Case Studies, Technical Features, and Management Issues.

The latest Smalltalk products will be displayed in the **Smalltalk Solutions '95** Exhibit Hall, where you'll have a chance to demo the leading Smalltalk products, and receive an up-close, hands-on comparison. Don't miss this chance to see Smalltalk in action.

Smalltalk Solutions '95 is presented by SIGS Conferences, sponsor of over 7 conferences world-wide, including **Object Expo**, **Object Expo Europe**, and **C++ World.**

SIGS CONFERENCES

For information on attending **Smalltalk Solutions '95**, please contact the SIGS Conferences Registrar:

PHONE: 212.242.7515

FAX: 212.242.7578

email: sigsconf@ix.netcom.com

DON'T MISS THIS UNIQUE SMALLTALK EVENT!

keep clean. Any interface you don't need will detract from the primary purpose of the prototype.

REMOTE CONTROL

For the television prototype, we will start on the remote control side. Recall that we have:

Object	*Responsibilities*
Keyboard	create Events from keystrokes
RemoteControl	read keyboard Events
EventStream	read and write Events
InfraredStream	read and write bytes

Figure 1 summarizes the Smalltalk objects I came up with and their relationships:

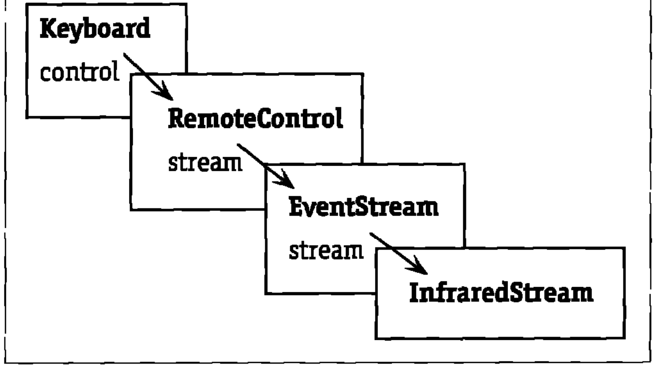


Figure 1. Objects in the Remote Control

We will start processing by sending character: aCharacter to the Keyboard to simulate the Keyboard noticing that a button has been pressed.

```
Keyboard>>character: aCharacter
    control event: (Event on: aCharacter)
```

Creating an Event sets its character and timeStamp.

```
Event class>>on: aCharacter
    ^self new setCharacter: aCharacter


Event>>setCharacter: aCharacter
    character := aCharacter.
    timeStamp := Time now
```

The RemoteControl doesn't try to process the Event, it just passes it along to its other half inside the television:

```
RemoteControl>>event:
    stream nextPut: anEvent
```

The EventStream lives to transform Events to and from bytes. I picked the simplest format I could think of, storeString. All objects can produce a storeString, which when compiled results in an equivalent object:

```
EventStream>>nextPut: anEvent
    anEvent storeOn: stream.
    ^anEvent
```

I'll defer the implementation of InfraredStream for a moment, since it is the trickiest piece of code and the least interesting architecturally.

TELEVISION

On the television side, we have the following objects:

Object	*Responsibilities*
TelevisionControl	map Events to commands
Television	change channels
EventStream	read and write Events
InfraredStream	read and write bytes

In the television, the EventStream and InfraredStream will be reading rather than writing as they were in the remote control.

Figure 2 is a picture of the objects in the Television.

We will start processing by sending the TelevisionControl the message "poll." This interface (and Keyboard>>character: used above) let us defer decisions about how control really works. Decisions like polling versus interrupts are important, but don't affect a well-factored architecture that much:

```
TelevisionControl>>poll
    stream atEnd
        ifFalse: [self event: stream next]
```

Getting the next Event from the EventStream is accomplished by compiling the characters in the InfraredStream.

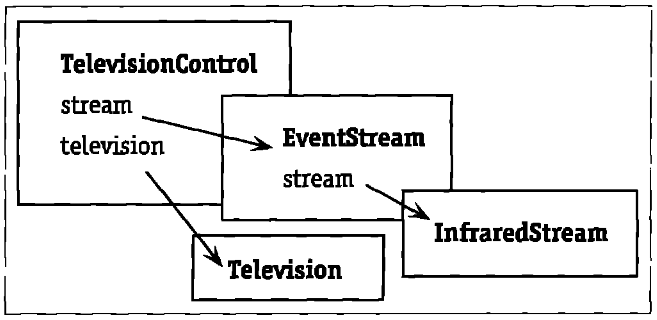


Figure 2. Objects in the Remote Control

Table 1.

Object	Pattern
Event	Event
Keyboard—Create events from keystrokes	Objectified Library
RemoteControl—read keyboard events	Objects from the User's World,Half Object
EventStream—Read and write Events	Formatting Stream
InfraredStream—Read and write bytes	Objectified Library
TelevisionControl—Map user input to commands	HalfObject
Television—Change channels	Objects from the User's World

```
EventStream>>next
    ^Compiler evaluate: stream upToEnd
```

Again, we will defer discussing InfraredStream until later. The atEnd test for EventStream delegates to the InfraredStream:

```
EventStream>>atEnd
    ^stream atEnd
```

TelevisionControls respond to an Event by sending the channel: anInteger message to the Television:

```
TelevisionControl>>event: anEvent
    television channel: anEvent digit
```

Events find their digit by getting the digitValue of their character:

```
Event>>digit
    ^character digitValue
```

Finally, Televisions just print the channel to the transcript to show that they have received the message:

```
Television>>channel: anInteger
    Transcript cr; show: 'Channel: ', anInteger printString
```

Figure 2 shows the effect of executing the prototype. The Keyboard has been sent character: 2. The TelevisionControl has been sent "poll." The new channel has been printed on the transcript.

InfraredStream

I promised to talk about how InfraredStream was implemented. We are trying to simulate two address spaces talking over an infrared beam. The implementation of the infrared protocol isn't interesting to the architecture, so we can simulate simply without worrying about how correct it is (although later we might want to take into account communication errors).

The trick is to have both InfraredStreams share a common OrderedCollection. The collection will contain the characters that have been written to one beam that haven't yet been read by the other. Figure 4 shows how the two streams look when they are connected.

Writing to an InfraredStream puts the character on the end of the collection:

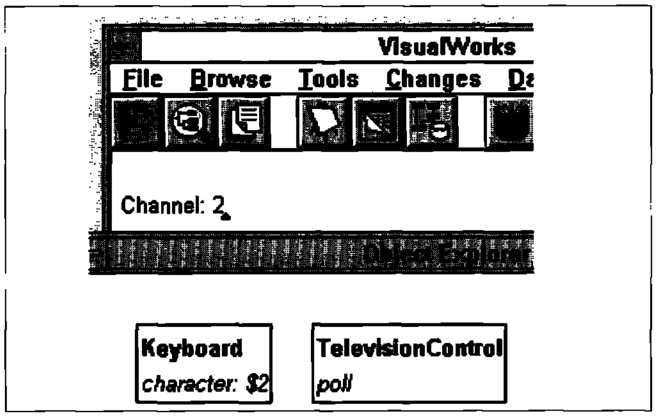
```
InfraredStream>>nextPut: aCharacter
    characters addLast: aCharacter.
    ^aCharacter
```

Reading takes the first character off of the collection:

```
InfraredStream>>next
    ^characters removeFirst
```

The atEnd test tests whether the collection is empty:

continued on page 32



Channel: 2

Figure 3. Executing the prototype.

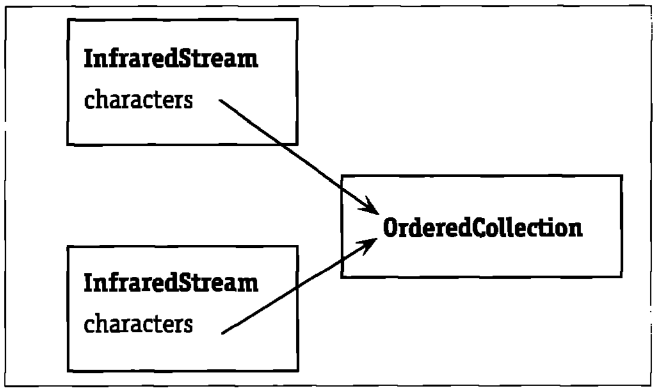
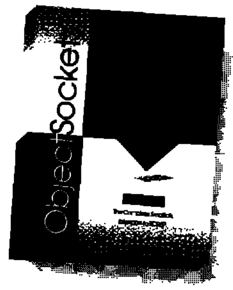


Figure 4. Streams sharing a Collection.

The complete Smalltalk interface to TCP/IP.



- Supports Smalltalk/V for Windows, Win32s, Windows NT
- Manages the TCP/IP asynchronous event notification system transparently
- More than 40 classes representing all aspects of TCP/IP programming

ObjectSpace™

SPECIALISTS IN OBJECT TECHNOLOGY

PRODUCTS · TRAINING · CONSULTING · MENTORING · AUDITING

For more information call 1-800-OBJECT-1, Email: info@objectspace.com

Recruitment

For information on advertising in the Recruitment Section, contact Michael W. Peck at 212.242.7447

Computer

SMALLTALK/OOP
Birmingham, AL

For more than 20 years, American Management Systems, Inc. (AMS) has led the way in the application of leading-edge information technology for our client base of Fortune 500 companies and public sector organizations, Today, our reputation as an industry innovator is stronger than ever.

Continued growth has created immediate opportunities in Birmingham, Alabama for qualified object oriented programmers, designers, and architects.

Requirements include 1+ years of experience in one or more of the following:

- Smalltalk
- Envy/Developer
- ODBMS
- C++
- OOA/OOD

Experience with Windows or UNIX is a plus.

We offer a competitive compensation and benefits package and career growth opportunities. For immediate consideration, please send your resume in confidence to: **Dept. SR-9411, AMS, Two Chase Corporate Drive, Suite 105, Birmingham, AL 35244.** An Equal Opportunity Employer M/F/D/V.

ams

AMERICAN MANAGEMENT SYSTEMS, INC.

Software Professionals

USAir, a recognized leader in the airline industry, seeks the following experienced software professionals for its Operations Research Department in **Arlington, Virginia** (just outside of Washington, DC). This organization develops high impact decision support systems for all divisions of the airline.

OBJECT-ORIENTED PROGRAMMER/ ANALYSTS
Yield Management Applications
Design and develop programs in Smalltalk. Experience developing client/server applications using relational databases under UNIX and MS-Windows strongly preferred.

OBJECT-ORIENTED PROGRAMMER/ ANALYST
Customer Services Applications
Design, develop and support systems to support a variety of projects. Experience with OS/2, communications software, GUI design, and object-oriented analysis required.

For consideration, please send a resume with cover letter, including salary history and academic GPA (recent graduates also send a copy of transcript), *making sure to clearly indicate position of interest, to:* **USAir, Employee Relations/IS Recruitment, 3800 N. Liberty St., Winston-Salem, N.C. 27105.** EEO Employer. Principals only, please.

USAir

The best of comp.lang.smalltalk

continued from page 24

```
someClass>>#someSelector
    | point array |
    point := 0@0.
    array := #((10 45) (78 10) (90 34)).
    array first class == Point ifFalse: [
        1 to: array size do: [:i |
            (array at: i) changeClassToThatOf: point]].
    ... more code here ...
```

Note that if you are really offended by self-modifying methods you can modify it yourself manually using the above code (slightly modified) as a **Doit** on the **CompiledMethod**. Personally, I like going around the image changing the class of objects just to see who objects. ;-)

NUMBERS WITHOUT LITERALS

In this same spirit of interesting intellectual exercises that may not be useful in practice, here is the promised discussion on creating numbers without using literals.

Normally, numbers are created either as literals, or as the result of operations on other numbers. It's not possible to just send new to a numeric class. For one thing, the result is not well-defined. I tried sending basicNew to Float anyway, just to see what would happen. On all the implementations I tried, this gave values that were unpredictable, but very, very small (less than 1.0e-20). So:

```
Float basicNew truncated
```

seemed like a plausible way to get zero. Unfortunately it isn't very reliable (since I have no idea where those numbers come from) and very platform-dependent.

It was only after I had tried this out on a couple of platforms that I thought of the much more obvious:

```
Object new size
```

which should give zero reliably on any platform. Given zero, the rest is easy. I can get one exploiting the simple mathematical fact that anything raised to the power zero is one.

```
| one zero |
zero := Object new size.
one := zero raisedToInteger: zero.
```

Given one and zero, I can use basic arithmetic operations, asFloat, and asDouble to get any other numbers I want. ♀

Product Announcements

Product Announcements are not reviews. They are abstracted from press releases provided by vendors, and no endorsement is implied.

Vendors interested in being included in this feature should send press releases to THE SMALLTALK REPORT, Product Announcements Dept., 885 Meadowlands Dr., #509 Ottawa, ON K2C 3N2, Canada, 613.225.8812 (v), 613.225.5943 (f).

Polymorphic Ships FastObjectFiler for Smalltalk/V

Polymorphic Software Inc. has announced the availability of FastObjectFiler, a tool that allows developers to save objects to disk. FastObjectFiler offers performance improvements of up to two orders of magnitude over Digitalk's native ObjectFiler utility, allowing Smalltalk programmers to use the convenience and ease-of-use of ObjectFiler to solve persistent object storage problems.

FastObjectFiler is implemented as a subclass of ObjectFiler, so it retains compatibility with existing files, and allows the developer to plug it right in to existing applications. Externally, the utility looks the same. However, internally, Polymorphic has re-engineered ObjectFiler's object traversal scheme, realizing substantial performance improvements in the process. The result is an ObjectFiler with performance that does not degrade when saving larger, more complicated objects.

FastObjectFiler is now shipping for Digitalk's Smalltalk/V 2.0 for Windows, Smalltalk/V 2.0 for OS/2, and Smalltalk/V 2.0 for Win32.

Polymorphic Software Inc., 1091 Insuctrial Rd., Ste. 220, San Carlos, CA 94070, 415.592.6301 (v), 415.592.6302 (f), 75010,3075 on CompuServe

ParcPlace introduces VisualWorks Business Graphics package

ParcPlace Systems, Inc. (Nasdaq: PARQ) announced the availability of VisualWorks Business Graphics, a new data presentation tool for building and incorporating charts and graphics into VisualWorks applications.

Fully integrated with VisualWorks, the Business Graphics package will bring a variety of presentation capabilities to corporate developers. Point-and-click editing makes the Business Graphics toolset easy to use and yields accurate data representation with minimal effort. Developers can choose from a wide variety of chart types, including: bar, band, line, and pie styles.

ParcPlace Systems, Inc., 999 E. Arques Avenue, Sunnyvale, CA 94086-4593, 408.481.9090 (v), 408.481.9095 (f)

Easel and Computer Systems Advisers form strategic partnership

Easel Corporation has announced a joint development agreement with Computer Systems Advisers, Inc. (CSA). Under the agreement, Easel and CSA will jointly develop a bidirectional bridge between Easel's Smalltalk-based Object Studio application development tools and CSA's SILVERRUN modeling hub.

The planned bridge will enable IS developers to share models and specifications between Object Studio applications and SILVERRUN, providing the necessary foundation for the design and development of both the client and server components of applications. The bridge will link Object Studio object-oriented applications with the installed base of relational databases modeled and maintained by SILVERRUN.

The bridge will also make it possible to develop an initial design of a data model in SILVERRUN, which could then be reflected in an Object Studio object model for client/server application development. Conversely, an object model created in Object Studio could be reflected in SILVERRUN's data model for generating databases. In addition, the bridge will allow changes in both the application and in the database to be reflected in the corresponding models. This will include existing databases that can be reflected in the models through SILVERRUN's reverse engineering facilities.

Easel Corp., 25 Corporate Drive, Burilington, MA 01803, 617.221.2100 (v), 617.221.6899 (f)

Hewlett-Packard Co. announces release 3.0 of HP Distributed Smalltalk

Hewlett-Packard Company announced version 3.0 of its HP Distributed Smalltalk development environment. This new release enables programmers who work with local or remote Smalltalk code to develop and deploy portable applications rapidly beyond the traditional client/server paradigm.

Distributed Smalltalk release 3.0 provides the speed and flexibility necessary to move beyond typical client/server architectures to true distributed enterprise application development. For example, it allows developers to encapsulate or surround existing data with a distributed object layer to provide communication between the existing data and applications developed with Smalltalk. This approach extends the life span of legacy systems while providing developers with the software reuse productivity gains of working with objects.

Release 3.0 is built on and extends ParcPlace's VisualWorks Smalltalk environment to create a distributed development environment. Distributed Smalltalk release 3.0 provides classes of objects that communicate over a network using an Object Request Broker (HP's implementation of the Object Management Group's CORBA 1.1 specification). It includes a number of distributed programming tools, such as a browser, debugger, interface repository, sample applications, and other utilities. HP's new release is available bundled with VisualWorks or on a standalone basis.

Hewlett-Packard Co., 3404 E. Harmony Road, mailstop 81, Ft. Collins, CO 80525, 408.447.4722 (v), 303.229.2180 (f)

MathPack: Mathematical classes for Smalltalk

MathPack from GSoft is a mathematical software package designed to assist engineers and scientists in mathematical computations using Digitalk's Smalltalk/V or ParcPlace's Smalltalk-80. With MathPack, a Smalltalk application is defined as a coherent set of Smalltalk classes and methods that solve specific mathematical problems. Mathpack contains over 120 mathematical classes and over 850 methods written entirely in Smalltalk.

Project Practicalities

continued from page 22

It looks as though I am asking anotherObject for my own list of employees. Because this statement is unnatural and confusing, developers will avoid using it improperly!

The public accessor methods

The public (get) accessor method name matches the name of the variable and answers a *copy* of the contents of the variable:

```
employees
    "public - Answer is a list of my employees"
    ^self myEmployees copy
```

Clients access the collection through this method. Of course, they can do what ever they want to the collection because it is only a copy and will not affect the original collection.

There is no public set accessor method (i.e., employees:). Instead, define methods that provide necessary behavior for the object such as addEmployee:. These methods can manipulate the object's variables as needed.

CONCLUSIONS

The use of the private variable protection technique described here retains the benefits of accessor methods while minimizing their drawbacks. It does not keep the Clients' code from invoking the private methods, since they are not truly private, but will help keep unintended changes to other objects' variables from occurring.

In a future article, I will explore a framework of methods that are useful to create to compliment the key collection protection methods described in this article. I will also look into uses of collection protection for variables that contain objects other than Collections. ♀

Reference

1. Beck, K. To accessor or not to accessor? The Smalltalk Report, 2(8).

Product Announcements

continued from page 31

MathPack provides classes for mathematical objects such as complex numbers and functions, radicals, decimal fractions, linear algebra, polynomials, rational functions, trigonometric, logarithmic, exponential, and special functions, with symbolic and numerical differentiation and integration, root finding, contour plots, splines, Bezier curves, and 2-D and 3-D plotting. In the statistical class, the following methods are available: average, standard deviation, variance, Chi-square test, F-test, Kolmogorov-Smirnov test, t-test, analysis of variance, Kendall-tau, regression analysis, general least-squares fitting, and random number generation. The Digital Signal processing class provides the basic functions for Fourier transform spectral methods, particularly the transformation of discretely sampled data, data filtering, and power spectrum estimation.
GSoft, 13918 Notley Road, Silver Spring, MD, 20904-1122, 301.384.8325 (v), 301.384.8325 (f)

Smalltalk Idioms

continued from page 29

```
InfraredStream>>atEnd
    ^characters isEmpty
```

Finally, upToEnd returns an OrderedCollection by default. We need it to return a String, because the result will be compiled by the EventStream. We can do this by overriding contentsSpecies.

```
InfraredStream>>contentsSpecies
    ^String
```

CONCLUSION

The architectural prototypes I've done for paying clients have been bigger than the television prototype presented here. Yours likely will be larger, too. The key point to remember is that you should write an architectural prototype to bring design discussions back down to earth. Whenever the abstractness of design is causing people to talk past each other, or fear of making concrete, "could-be-proven-wrong" decisions is slowing progress, a little bit of code goes a long way towards advancing the project.

In the next issue I will begin to discuss how patterns can be used to document reuse. ♀

A Trace Logger

continued from page20

```
Enter(2:26:57 pm): LhMoveRequest (acs=0, vsn='RB1400')
SharedQueuePacket>>queueYourselfUsing:

Enter(2:26:57 pm): acs: 0
ChannelManager>>sharedQueueForAcs:

Enter(2:26:57 pm):
Channel>>processPacket:
```

And how useful is this whole scheme? In our development, we keep the logger active whenever we are running the product and debugging our work. Since we don't have a user interface that gives us visual clues as to what is happening, we find it very convenient to scan through the log file to see what was going on in our product if we see a problem. The other option is to add self halt messages in judicious places, which we also do, but we've often found it quicker to see what was happening internally by looking at the log file. When the product is shipped to customer sites, this will be the only information that the support people have to help track down the causes of problems.

All in all, we've found our TraceLog class to be very useful. More than that, it was a lot of fun to create, and gave us an opportunity to learn some very interesting Smalltalk features. ♀

Alec Sharp is an Advisory Software Engineer at StorageTek. He is the author of Software Quality and Productivity. He can be reached at alec_sharp@stortek.com. Dave Farmer is a Senior Software Engineer at StorageTek. He can be reached at david_farmer@storetek.com. They both work on the Unix Storage Server software, which manages connections to networked hosts and drive the StorageTek family of Robotic Tape Libraries.