THE Smalltalk REPORT
PROVIDING SOLUTIONS TO THE SMALLTALK COMMUNITY

SIGS PUBLICATIONS

Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, THE SMALLTALK REPORT, THE X JOURNAL, REPORT ON OBJECT ANALYSIS & DESIGN, OBJECT EXPERT (UK), and OBJEKT SPEKTRUM (GERMANY)

Cover illustration by Sandra Dionisi/SIS

Table of Contents

November–December 1995 Vol 5 No 3

Feature

Columns

Departments

Individual Subscription rates 1 year (9 issues): domestic $89; Mexico and Canada $114, Foreign $129; Institutional/Library rates: domestic $199, Canada & Mexico $224, Foreign $239. 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 domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sigs.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

Editors' Corner

John Pugh

Paul White

So, JUST WHAT is the proper way to test Smalltalk systems? This is a question we get asked often, and indeed are still trying to formulate an answer to it after all our years of building systems using Smalltalk. Of course, there won't be any one answer, since the potential schemes are as diverse as the types of systems being developed. And, for the most part, the issues aren't that different for Smalltalk systems than any other type of systems. But nonetheless, testing Smalltalk applications should not be as "hit or miss" as it has proven to be. And even those of you whose organizations have succeeded at fully testing Smalltalk applications in a systematic way are often using home-grown, customized systems that cannot be adapted to test other similar systems. This is not to say an excellent job of testing isn't being done, it's just that as an industry, software engineering needs to find more reusable solutions to proper testing. If tools such as browsers and window constructors can be created to work across multiple applications, then why can't testing tools be created as well?

At OOPSLA this year, we had the opportunity to query a number of "experts" on their strategies for testing. There are some obvious things one can do. For example, it seems clear that we need to develop a test suite that will exercise each individual method defined for a class. This is really the old style unit test, only with Smalltalk the methods being tested tend to be much finer grained than traditional systems, since methods are generally simpler than traditional style functions. And there have been some notable new tools brought to the market recently that address this type of testing (as well as other types).

Beyond individual methods, how does one define test suites? The first thought is to next test the behavior of a class. But how realistic is this type of test? This is certainly feasible if the class performs some well-defined function, and is well decoupled from other classes. But typically a given class relies heavily on at least a small subset of related classes, and therefore must be tested with those other classes as a single unit. This can often prove difficult because of the different behaviors that might be exhibited by the instances of a given class. Moreover, when we start introducing inheritance into the picture, this approach of testing a class becomes even more difficult.

In fact, a more significant test is probably not of the class per se, but rather of objects. If the goal is to build well-defined components, then it should be possible to define test suites for objects independent of the class to which they belong. This might help us avoid some of the issues introduced by inheritance, although not entirely.

The other style often employed when developing test suites is to generate all our test cases as code. While on the surface this seems reasonable, and is certainly the fastest method to test, it does seem to have some serious shortcomings. In an object-oriented world, it seems obvious that we should be developing "testing objects" rather than representing our tests through code. What responsibilities/behavior does a testing object have? It should probably, as a minimum, keep track of the tests it is to perform, the objects on which it is performing these tests, and the expected results. There are a number of different ways to implement such a test object, but it should be possible to build a standard protocol for all tester objects. Once this is done, and a widely accepted protocol gets adopted in the industry, we will be able to begin simply testing our code without having to build all the infrastructure for it, as is the case for most of us. Also, it would be nice if a series of "testing patterns" could be generated by those who have the experience to make life simpler for the rest of us who are struggling with this problem.

Like many of the outstanding issues in software development, there is a great deal of effort being expended on trying to address this issue. At OOPSLA, there was a full-day workshop dealing with this very topic. We will try to have someone who attended the workshop write an article to bring us all up to speed on the current state of the technology.

Meanwhile, we wish to draw your attention to the review by Jan Steinman and Barbara Yates in this month's issue. They review the new book SMALLTALK WITH STYLE (Skublics, Klimas, and Thomas), and we concur with them that titles such as this are long overdue. As pointed out in the review, this book should not be used as "the Bible" for style in development, but should certainly be used as a foundation for developing your own style guidelines. The importance of a consistent style across a project is often overlooked by development groups. It is no different than architects using a standard notation with their diagrams or accountants using standard accounting principles. Properly utilized, a standard style will help streamline the software development process within your teams, and to this end we highly recommend that you formalize your own style guidelines (along with guidelines for design principles, testing principles, etc.).

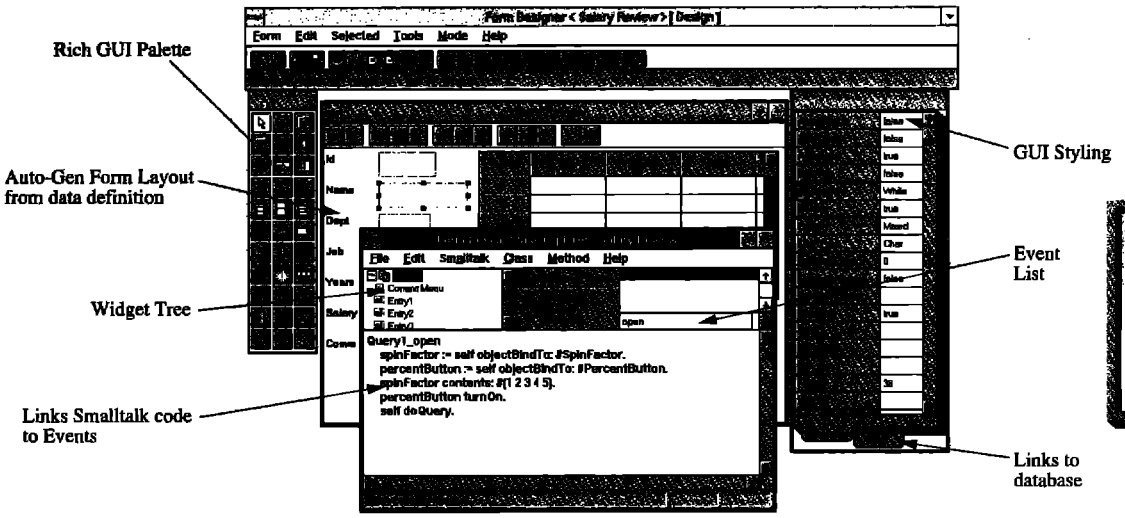
Enjoy the issue.

Visual TriO™ for Smalltalk version 2.1

Simplify Smalltalk Programming

Let Visual TriO handle GUI, OOUI and database access details
So you can focus on application design and delivery

Special Offer
Until Feb. 29, 1996
US$250.00 includes US$20 rebate



> *"I think that the exciting thing about Visual TriO is that you can lower the entry barrier to Smalltalk"*
> - Bruce Gilham,
> President, Dataforce
> Chairman, Los Angeles Smalltalk User's Group

Works like Visual Basic with the Object Power of Smalltalk

Makes Smalltalk Easier

Visual TriO is a comprehensive tool suite add-on to Visual Smalltalk. Its streamlined work-bench, automated database access and GUI support helps veterans deliver applications faster and novices get started quicker.

- Powerful Form Designer visually creates advanced GUI
- Class Explorer visually links Smalltalk code with GUI events
- Visual subclassing
- Controls Library facilitates reuse of custom controls
- Utility to manage global name space
- Incremental loading of classes into the Smalltalk image
- Incremental save of design work outside of the Smalltalk image
- Packaging utility aids application deployment
- Tutoring tool takes you step by step from simple GUI to advanced database and OOUI applications
- Team Support: Visual check-in / check-out via interfaces to Intersolv's PVCS

Automates Database Access

Visual TriO's data-smart and SQL-smart Form Designer help you create the visual parts of your application *in the context of the data you work on:*

- Links GUI controls to database fields via point and click
- Visual TriO generates SQL, manages the unit-of-work, concurrent access, commits and rollbacks across multiple tables
- Auto-generates GUI layouts from database
- Built-in ODBC or native data wrappers for popular databases:

  Access, FoxPro, dBase, Paradox, SQL Server, Sybase, Oracle, DB2/2, DB2 family via DDCS/2

- Embedded Btrieve SQL engine facilitates rapid prototyping (except Windows NT)
- Extendible to more complex data structures via master/detail, visual joining of tables, user defined data attributes and DDE

Creates Advanced GUI

Visual TriO gives you all the basic controls plus many more:

Custom Subpane, Status Bar, Table, Toolbar with Tip, Notebook, Business Graph, Timer, Hot Point, Gauge, Dial, Hierarchical List, Spin Button, Context Menu, Picture (ICO, BMP, GIF, TIFF, PCX, WMF, JPEG, EPS, IMG, WPG, DIB and Targa)

Creates Enticing OOUI

With Visual TriO, it's easy to visually program OOUI effects like: drag-and-drop, context menus, and conditional icons.

Give your end-users the same expressive flexibility and freedom-of-action as the Windows 95 or OS/2 Warp OOUI desktops!

• 30 Days Money Back Guarantee
• Royalty Free Runtime

TechBridge Technology Corp.

STIC

Ready! for OS/2

MICROSOFT® WINDOWS™ COMPATIBLE 32-Bit Application

To order call 1-800-463-8998
73532.456@compuserve.com

MasterCard VISA

TechBridge Technology Corp.
5001 Yonge Street, Suite 1301, North York, Ontario, Canada M2N 6P6
Phone: (416) 222-8998 Fax: (416) 222-0168

System Requirements - Visual Smalltalk v3.0.1 from ParcPlace-Digitalk / Windows 3.1, 3.11, NT; OS/2 2.1, Warp v3/ 486 33 MHz or higher with 12M RAM and 12M disk space.

Understanding inter-layer communication with the SASE pattern

Kyle Brown

In a previous article,[1] I described how Smalltalk applications are best built in layers, with each layer having a well-defined interface and well-defined communication paths to the layer beneath it. A layered architecture promotes

- looser coupling between objects
- better factoring of responsibility

both of which encourage reuse and ease maintenance.

However, once you have chosen a layered architecture how do you set up the communication paths between the layers? Specifically, how do you decouple a view from any specific model? To discover how this communication occurs in modern Smalltalk systems, let's take a look back at the old days of Smalltalk and the MVC model.

In the good old days of "classic" MVC, models, views, and controllers came in a triad. When you developed a view, it was generally hardwired to work with a specific class of model. Later, the notion of "pluggability" mitigated this hardwiring by allowing the creator of a view to specify the selectors of the messages that would be sent by the view to the model. However, this still had the drawback of restricting that all messages from a view must be sent directly to the instance of a model that the view held in its "model" instance variable.

In part, this was due to the assumptions implicit in the Observer[2] pattern that was used (in the form of change/update) in most Smalltalk implementations. Observer assumes that it is okay for an observer (a view) to know a little bit about the subject (a model), but that the reverse is not true. To gain real flexibility, however, even this assumption had to be relaxed.

Each of the major Smalltalk vendors have addressed these particular problems in their recent releases. VisualWorks 2.0, Visual Smalltalk 3.0 (VST 3.0), and IBM Smalltalk 2.0 share the same general solution to this problem. This common solution can be described by a new design pattern I call Self-Addressed Stamped Envelope (SASE).

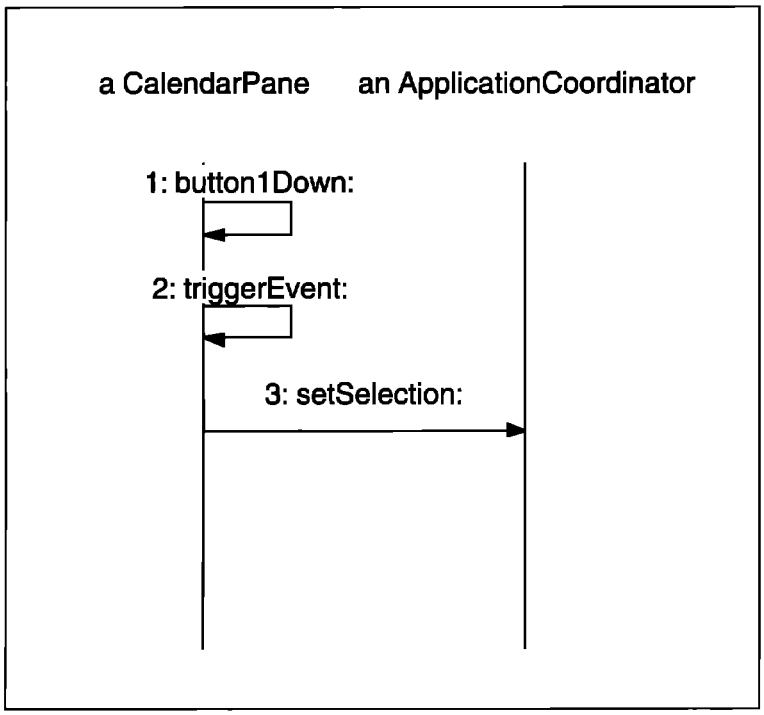


Figure 1. #setSelection: message flow.

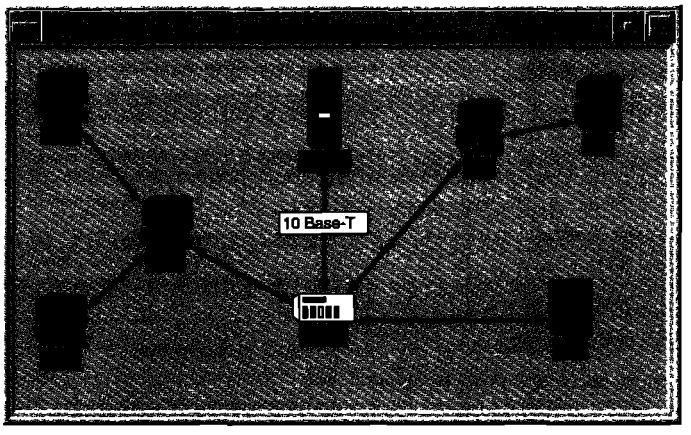
Problem: How do you define a context-free way to notify an object of the occurrence of an event? In particular, how does a view notify an object somewhere in the application layer that an event (e.g., a button press or a selection change) has occurred without specifically needing to have knowledge of what object to notify, and what message to send?

Solution: Define a mapping in advance from an event to a set of receivers and messages to be sent to these receivers. Use Smalltalk's #perform: facility to send that message when the event occurs at the "sender."

The pattern is called SASE because of the analogy to sending a self-addressed, stamped envelope to a recipient with the understanding that the recipient will send back the envelope whenever an event occurs. For example, you may want to send

Now it's Easy to Build Interactive Diagrams

Quickly create advanced interfaces that convey information better than lists... with DDF

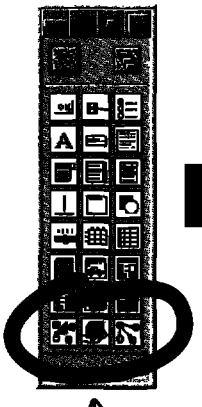


Example interface built with the Dynamic Diagram Framework

DDF™ is an easy-to-use tool that dramatically reduces the time needed to build interfaces:

- makes building diagrams simple
- provides a new VisualWorks widget
- pre-configured for immediate use
- written completely in Smalltalk
- refineable and extendable
- includes ARS's Parcels & Structured Graphics for building "dynamic" nodes

DDF - Dynamic Diagram Framework



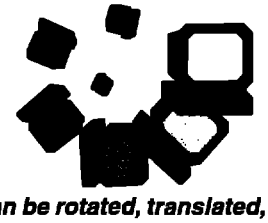
With DDF™ you can quickly and easily ...

- create customized node icons with shapes
- add and remove nodes from a diagram
- connect and disconnect nodes within a diagram
- customize lines and line decorations
- select and move nodes
- format diagrams and hide nodes
- print and store diagrams
- dynamically update diagrams

P&SG - Parcels & Structured Graphics

Parcels & Structured Graphics (P&SG™)

- high precision 2-d object-oriented graphics for VisualWorks
- structured graphic shape objects
- drag-and-drop with parcels
- shapes recognize "hot-spots"
- provides 2 new VisualWorks widgets



Shapes can be rotated, translated, scaled and combined to form new shapes.

Call (800) 260-2772 today to order or e-mail info@arscorp.com for more information. Ask for a free copy of the white-paper "Building Diagram-Based Applications with DDF"

Also Available: **MI - Multiple Inheritance**

Applied Reasoning Systems Corporation (ARS) is an innovative developer of high quality Smalltalk development tools, application frameworks, intelligent software systems, and related services that provide advanced solutions to complex problems.

Smalltalk Products • Consulting • Education • Mentoring

Applied Reasoning Systems

2840 Plaza Place • Suite 325 • Raleigh NC • 27612

Phone: (919) 781-7997 • E-mail: info@arscorp.com

an SASE to contest promoters with the understanding they will return it (with a list of the winners) when the contest is over.

As an example, let's look at the Event interface from Visual Smalltalk 3.0. In VST 3.0, each class can define a set of "events" that it can trigger. While all objects have this capability, it is used most often in the SubPane hierarchy, whose subclasses define events like:

- #clicked
- #needsContents
- #textChanged

When a particular object is interested in receiving notifications about an event from a SubPane (something an ApplicationCoordinator might do) it registers itself with that SubPane using the #when:send:to:with: method.

```
MyAppCoordinator (class) >> buildView: forModel:

...
aPane when: #clicked send: #setSelection:
   to: aModel with: aPane.
```

Now, at some point in the future, the SubPane will (in response to a mouse action) send itself the #triggerEvent: message, with #clicked as the argument. This will result in the message #setSelection: being sent to the ApplicationCoordinator, with the SubPane being the argument. This message flow is shown in Figure 1.

As you can see from the previous example, this implementation gives us several properties we were looking for:

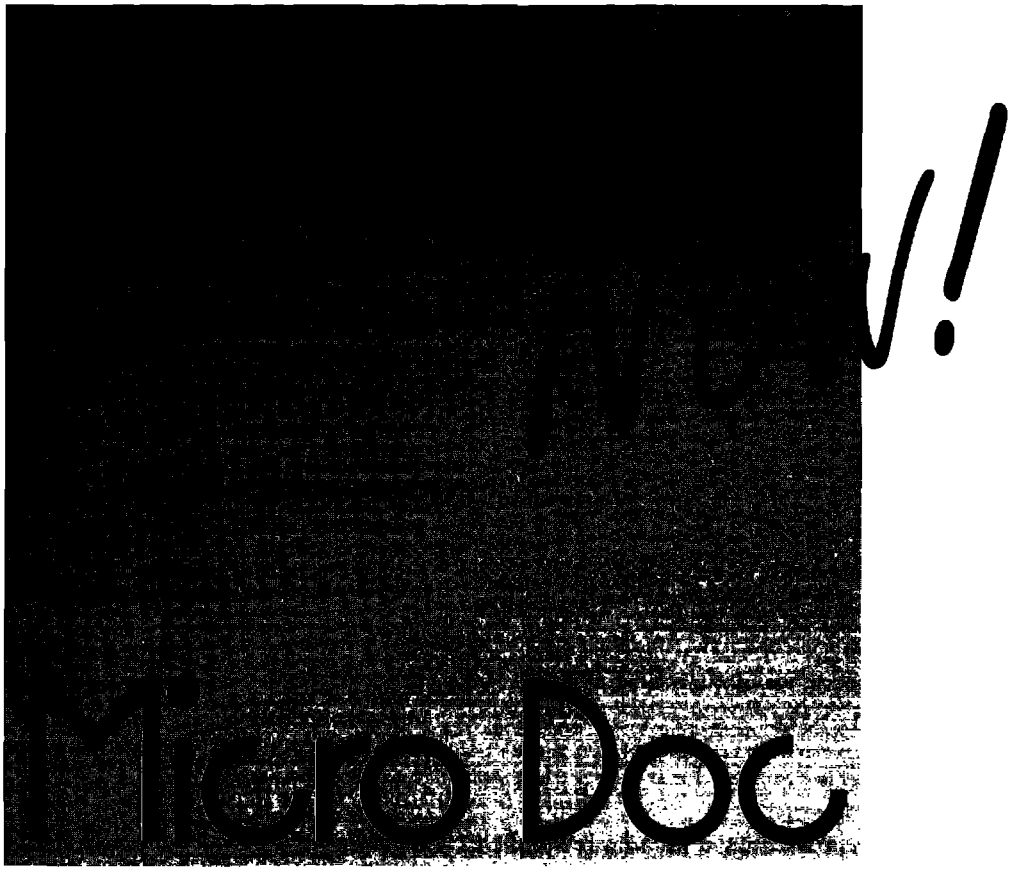
- Because the "to" argument can be any arbitrary object (and not just the View's model) we can send a message to any object in the ApplicationModel layer.
- Because the "send" argument can be any message, we are not forced to make the receiver of the message conform to any particular protocol. We can instead use whatever message is appropriate.
- This method more loosely couples the view and the application layer objects, since the view does not hard-code any methods that it sends to these objects.

The same pattern is used in IBM Smalltalk and VisualAge in a slightly different implementation, but for the same purpose. In IBM Smalltalk, a CwWidget (the closest equivalent to a VST 3.0 SubPane) implements two messages

- #addCallback:receiver:selector:clientData:
- #callCallbacks:callData:

The first method is equivalent to the #when:send:to:with: method in VST 3.0, in that it specifies an event (a Callback Constant), the receiver of a message, the message, and the arguments to the message to be sent when the event

Smalltalk and Lotus Notes™ Integration



Objects for Notes

Smalltalk Objects to access Lotus Notes
available for
IBM® and Visual Smalltalk™
OS/2® and Windows™

MicroDoc Computersysteme GmbH, Sternstraße 21, D-80538 München, Germany
Tel: +49-89-2908 5171, Fax: +49-89-22 28 67, CIS 100015,3007

occurs. A CwWidget "triggers events" by sending itself the #callCallbacks:callData: message.

In addition, IBM Smalltalk utilizes an almost identical set of methods for communication between objects within the view layer. CwWidgets respond to the method #addEventHandler:receiver:selector:clientData:, which adds an "event handler" to a CwWidget that is called when an event (a mouse movement, expose, or keyboard event) comes in from the underlying window system. Commonly, one of the last things that happens in an Event Handler is to send a callback to notify an application layer object that, say, a mouse click has been interpreted as a list selection.

Coming back full circle to VisualWorks, which descended from the original Smalltalk-80 that gave us change/update, we see that SASE is used here too, but in a slightly different way. In VisualWorks, instances of ValueHolder understand the message #onChangeSend:to:. For example, in an ApplicationModel you might see,

```
MyApplicationModel>>postBuildWith: aBuilder
...
listSelectionHolder onChangeSend: #changedSelection
  to: self
...
```

Whenever a ValueHolder receives a #changed: message, it will (actually, a DependencyTransformer will) send the message selector specified in the #onChangeSend:to: message to the object specified in the message. This implementation differs slightly from that of VisualSmalltalk and IBM Smalltalk in that it does not also specify an "event" or "callback" symbol that specifies under what specific circumstances the message is to be sent. Instead, in VisualWorks several ValueHolders are used, one for each particular circumstance. For instance, if a VisualWorks ApplicationModel wanted to know when the contents of a ListView changed, and when the user changed the selection, the ApplicationModel would have to register with two different ValueHolders—one representing the state of the selection, and another representing the state of the list itself.

Now, one benefit that this pattern gives you is the ability to decouple objects in different layers that need notifications of changes, but that may have varying protocols. For instance, let's consider the common case in which a change to one object will affect many objects in a different layer. As an example, consider a class LoginMonitor whose responsibility it is to know if a user is logged in to the system. Let's say a requirement exists that if a user does not use the system for a fixed period of time (say, 10 minutes) then the LoginMonitor would have to notify each of the open windows in the system to log themselves out, and would have to log out of any open databases or mainframe connections.

Using the SASE pattern, each interested object (be it an application layer object or an infrastructure object) could register itself on the LoginMonitor (which would probably be a singleton).[2] Whenever the LoginMonitor "went off" it would then automatically notify each registered object in the specific way that each requested at registration time. The LoginMonitor is unaware of either the existence of its registrants or their protocol, keeping the system very loosely coupled. The registrants only need to know:

- that the LoginMonitor exists
- that it complies with the standard SASE protocol for registration
- the name of the event/callback or message that returns the proper ValueHolder

So, you can see that SASE can permit a system to be even more loosely coupled than the Observer pattern, and that implementations of this pattern are extremely similar in each of the major dialects. Seeing the commonalities allows you to think in more abstract terms than the specific implementation, and also allows you to think about cross-dialect portability from design time.

References

1. Brown, K. Remembrance of things past: Layered Architectures for Smalltalk applications, The Smalltalk Report 4(9):4–7, 1995.
2. Gamma, E. et.al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

Kyle Brown is a Senior Member of Technical Staff at Knowledge Systems Corp. and a frequent contributor to The Smalltalk Report. He has over six years' experience working with Smalltalk. He can be reached at kbrown@ksccary.com.

A methodology mix

Mark Lorenz

FOR ANY OF YOU THAT HAVE WORKED on many object-oriented projects, it has been obvious for some time that the best methodology is a mixture of methodologies. Virtually all the commercial O-O projects I have been involved with over the last few years have used multiple methodologies to develop their O-O systems. In fact, IBM has standardized on a methodology called Visual Modeling Technique (VMT), that combines the best of Responsibility Driven Design (RDD), the Object Modeling Technique (OMT), and Object Oriented Software Engineering (OOSE). This is essentially the same methodology we use at Hatteras and the methodology we will focus on in this article.

A METHODOLOGY OVERVIEW

The basic steps to the methodology are shown in Figure 1 and briefly discussed in the following sections. Of course, this is not a monolithic waterfall approach, but rather a systematic process that results in requirement traceability using techniques that are natural and easy to learn.

Write use cases from requirements

There should be a use case written for each public service required of the system. The use cases focus on what is to be provided (see Fig 2). This is the first step in the development threads that trace back to the system requirements, as shown in Figure 3. These use cases have the added benefit of being good inputs for test cases.

Write scenario scripts from use cases

Each use case will typically need multiple scenario scripts to support it. Scripts focus on *how* the object model under development will support the use case. They are composed of time-ordered sequences of public message sends, documented in steps detailing the *initiator*, *action*, and *participant*. An example partial script is shown in Figure 4.

Fill in the object model from scenario steps

The scripts focus on the basic concepts in the business domain, resulting in classes and their relationships. As these efforts are taking place, class clustering into subsystems occurs based on coupling due to high-level services provided. This results in details required to satisfy the use cases being added to the object model under construction, as shown in Figure 5.

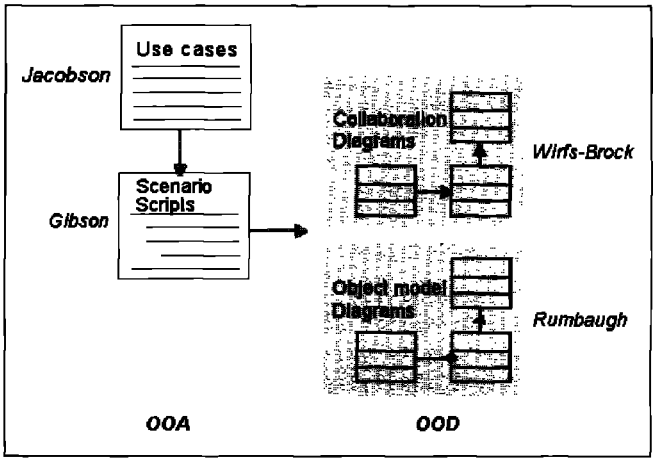


Figure 1. Methodology mix overview.

Selling products

The salesperson answers the phone and asks the customer for her phone number. The customer's information appears on the screen and the salesperson verifies the name and address.

The salesperson asks the customer for an item number. The salesperson sees the item information and verifies the type of product being requested. The salesperson asks for a quality and enters it. The salesperson asks for more item numbers until the customer is done ordering.

The salesperson verifies the last credit card used or gets new credit card information. The salesperson tells the customer the total amount and when to expect the shipment.

The inventory is updated once the order is committed by the customer. The invoice and picking slip are printed in the warehouse. The picker collects the items and puts them in a box for the shipment. He includes the picking slip in the box and puts the invoice on the outside as a shipping label. Once the shipment is completely satisfied, the order is archived, until then, the order is outstanding. The picker takes the shipment to the shipping dock for pickup.

Figure 2. Example use case.

Mark Lorenz is Founder and President of Hatteras Software Inc., which offers education, modeling, mentoring, and products to help other companies successfully use object technology, as evidenced by commercial products such as IBM's StorePlace and Hatteras' OOMetric. He welcomes questions and comments via email at mark@hatteras.com or phonemail at 919.319.3816.

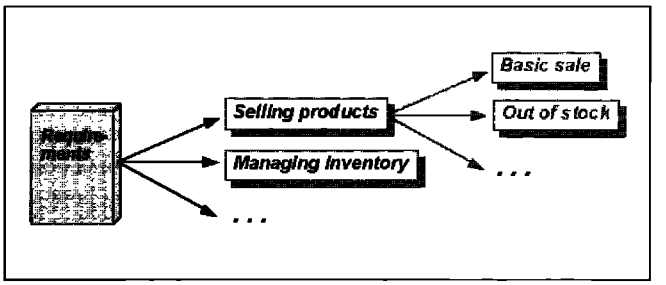


Figure 3. Requirement traceability.

Develop collaborations from scenario steps

Similarly, the scripts focus on the public behaviors exhibited by the identified classes to service the requirements. Groups of related public services called *contracts* are created, both for the classes and subsystems. Figure 6 shows an example diagram, with an indication of the amount of detail needed to capture the key relationships and behaviors in the business model under development.

TOOLS TO SUPPORT A METHODOLOGY MIX

Using a mixture of methodologies may require the use of multiple CASE tools, but there are also tools that handle the mixture. Paradigm Plus supports a number of methodologies, but it doesn't make mixing techniques across methodologies easy. It also doesn't support RDD as well as I'd like to see. HOMSuite supports a mixture like we've been discussing and has been used on successful commercial product development, such as IBM's StorePlace.

As methodologies evolve and people move around the industry, we will hopefully see some convergence toward an effective mix. For example, Booch's changes in techniques in the last year has moved him closer to the mix I propose in this article.

Basic sale

-This script details our phone order-taking procedures from customers
OrderWindow** requests customerFor: a PhoneNumber from **Company
  Company** asks hasPhoneNumber:aPhoneNumber from **Person
  ***script:** New Customer*
  ***script:** Customer information updates*
  ***branch:** Bad credit record*
OrderWindow** sends for: aPerson to **OrderTransaction
-Iterate across the following steps for each product the customer orders.
OrderWindow** requests productNumbered: aNumber from **Inventory
  Inventory** asks isProductNumber: aNumber for each **Product
  ***script:** Product not found*
  ***script:** Product search*
OrderWindow** asks name, description, and price from **Product
OrderWindow** sends sellQuantity: aNumber of: aProduct to **OrderTransaction
  OrderTransaction** sends sellQuantity: aNumber of: aProduct to **LineItem
    LineItem** sends deplete: aNumber to **Product
OrderWindow** asks total from **OrderTransaction
-End of line item sale.
OrderWindow** asks creditCard from **Person
OrderWindow** asks number, expirationDate from **CreditCard
***script:** New credit card*
***script:** Out-of-stock lineItems*
***script:** Order cancelled*
-We're now in the Warehouse
OrderWindow** requests submit to **OrderTransaction
  OrderTransaction** requests printFor: self to **Invoice

Figure 4. Example partial scenario script

A PROJECT ARCHITECTURE

I have previously written about the essential components of an architecture for your O-O systems and how to grow your teams around this architecture. This architecture revolves around the grouping of classes into subsystems and identifying and controlling public interface contracts between those subsystems. This methodology leads directly to the development of this architecture, which is essential for your project's success.

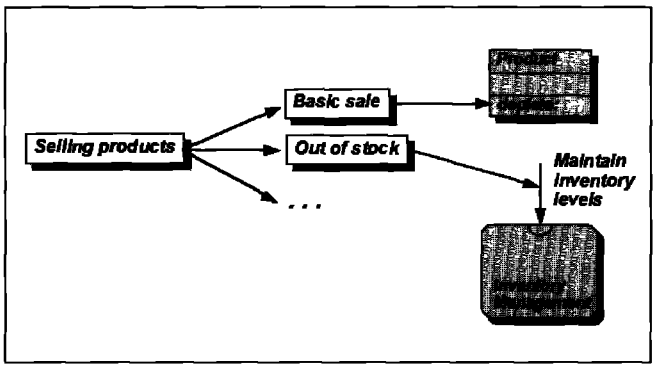


Figure 5. Filling in model details from scripts.

SUMMARY

We have examined the steps used in a methodology that is composed of important elements of multiple other methodologies. This mix optimizes the development of an object model and architecture. It has been effective in numerous O-O commercial projects and is the methodology most commonly found on the O-O projects I've been involved with over the last few years. The techniques are relatively easy to learn and provide good requirements traceability.

Terminology

architecture	The subsystems and their contractual interfaces for an O-O system.
collaboration diagram	A graphical view of an O-O system design that shows subsystem groupings, classes, and contractual usages.
contract	A logical grouping of related public responsibilities.
object diagram	A static model of an O-O system design that shows classes, methods, state data, and relationships between classes.
script	A time-ordered sequence of messages between public interfaces of key

continued on page 20

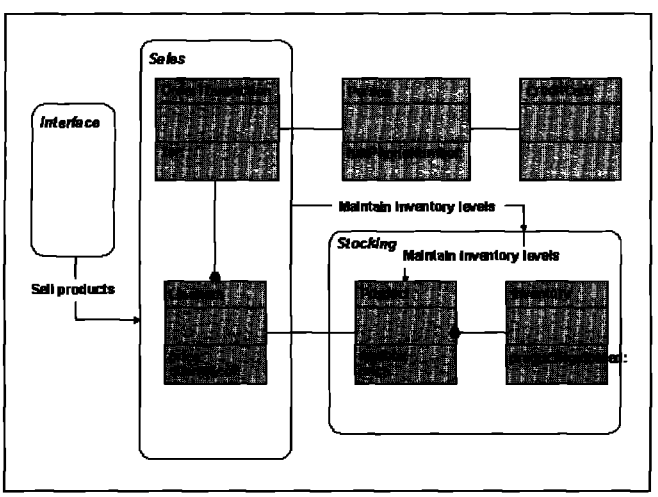
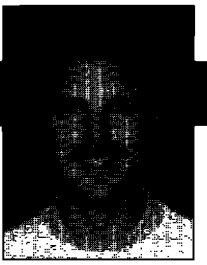


Figure 6. Object model and collaboration diagram example.



Variables of the world

Kent Beck

In the last issue, I presented the four ways temporary variables are commonly used. This time, I'll talk about how instance variables are used. The results for instance variables are nowhere near as tidy as those for temps. I'll speculate as to why after I've presented the information.

SOAPBOX

But first, I'd like to whine and complain a little. Here's the essence of my beef—it's getting harder, not easier, to write Smalltalk applications. This is not what I expected. Smalltalk had already raised the level of programming so much from what I was used to that I figured the trend would continue. Today's clichés would become tomorrow's abstractions and the day after that we would forget we ever had to program that stuff. Onward and upward.

Instead, I see my clients programming the same stuff time after time. Here are some examples:

- **Unit values**—If I want an object representing five days, I shouldn't have to create "January 5, 1900" or fall back on plain old "5." Five days ought to be five days. Decent unit values would catch lots of nasty semantic errors and eliminate code that is currently scattered through lots of domain models.
- **Time and date intervals**—"Every Thursday this month," "1 AM every night," "every month this year." Each of these expressions, used in almost all calculations, should be represented by an object.
- **Multi-currency calculations**—There is no reason Smalltalk applications should have to flinch at dealing with multiple currencies. Application developers should use a Money object to represent monetary values. Once the application knows it is dealing with money, supporting multiple currencies is a snap.
- **Drawing editors**—Interfaces where the connections between things are as important as the things themselves aren't effectively represented as lists, text, tables, or notebooks. A good framework for direct manipulation interfaces would go a long way toward distinguishing Smalltalk applications.
- **Active objects**—Time marches on, but not if you look at most of the Smalltalk library. I can't count how many times I've written an object that keeps hold of a Process and answers messages like "start" and "stop.". Doing a completely preemptive thread safe library is a lot of work. That's overkill for most applications. A little help writing and debugging active objects would go a long way.

One interesting question is why such obvious objects aren't part of the shared language of Smalltalkers. The boring answer is that buyers don't have these objects on their check lists, so the vendors don't produce them.

The more interesting answer is that the Smalltalk culture has shifted from producers of abstractions to consumers of abstractions. We have in our hands the best tool I've ever seen for creating reusable stuff, but we're all so busy writing apps that as a community we don't step back and make things that everyone can use.

Of course there is an economic rejoinder to this—it isn't possible to make money making reusable software. So what! Good abstractions are the product of experience and inspiration, not economics.

We need to change our culture. Application developers need to demand higher and higher levels of abstraction from their vendors. Framework developers need to create and publish abstractions, even if they don't make any money at first. Vendors need to aggressively search for, incorporate, and educate about the best new abstractions. In short, we have to start acting like a community, putting aside some short-term gain for the greater good.

I'm putting my time where my mouth is by putting my unit testing framework in the public domain. I'm also preparing my multi-currency framework for public consumption (it'll be a few months, but I'll get there).

INSTANCE VARIABLES

The temporary variables boiled down to a simple set of patterns. You can use a temp to:

- cache a value for performance
- hold a value of a side-effecting expression
- explain a complex expression
- collect results from a complex enumeration

I discovered these uses by looking at every method in the system that uses temporary variables and classifying them. Pretty soon the first three classifications became clear. After a while I had to add a fourth.

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

When I tried to do the same thing for instance variables all I got was a muddle. I came up with nine uses. Where temps were clear, however, these nine uses are not. You can classify one variable as two or three at once. I also invented three (mostly orthogonal) styles of usage of instance variables.

Ward Cunningham and I tried to figure out why instance variables are so much harder to pin down than temps. I wasn't satisfied with our answer, but here it is: Temporary variables are tactical. They are created to resolve a set of constraints that only exist within the scope of a single method. Instance variables are often created to solve much bigger problems, problems that may span many objects.

In the process of writing a handbook for software engineering, we've been much more successful at canonizing coding practice than design or analysis practice. The decision to create an instance variable goes back to design or even analysis. It shouldn't be surprising that the result isn't crystal clear.

Styles

Having successfully lowered your expectations, here are the three styles I've found so far:

1. Private
2. Public
3. Acquaintance

Private. These are instance variables that are a simple part of an object. They are used almost exclusively by the object itself within its own methods. A good example is the Visual Smalltalk version of OrderedCollection. It has variables startPosition, stopPosition, and contents. No object outside of the OrderedCollection has any need for the values of these variables.

Public. These are instance variables that are more complex parts of an object. They are often made available to the outside world for further processing. Frequently, they hold objects that are complex in their own right. However, if the referring object didn't exist, the object referred to by the variable wouldn't need to exist. Panes in Visual Smalltalk have an instance variable "pen" which holds a Pen. If you want to draw on a Pane, you need its pen. You can often improve your design by shifting responsibility into an object and making some of its public instance variables private.

Acquaintance. These are variables that are there for convenience, but don't imply the sort of ownership of a private or public instance variable. Stream's instance variable "collection" is an acquaintance. If you have an Array you need to stream over, you could send it along with every message to the Stream (nextPut:on:, nextFrom:). The protocol would be much uglier and there would be a greater chance of errors if you used different collections at different times. Thus, Streams get acquainted with one and only one collection.

Uses

Here are the nine uses I've found so far:

1. Parent
2. Child
3. Name
4. Properties
5. Map
6. Current state/strategy
7. Pluggable selector/block
8. Cache
9. Flag

Parent. Sometimes an owned object needs to acquaint itself with its owner. The owner provides context for calculations. VisualWorks' VisualPart has an instance variable "container" that points to the containing VisualComponent. You can improve your designs by passing more context into the owned object and eliminating parent variables. This allows one object to be "owned" by several others.

Child. In tree structures, interior nodes need a variable to hold a collection of children. VisualWorks' CompositePart has a variable "components" that contains an OrderedCollection of VisualComponents.

Name. If everyone who refers to an object must use the same key to identify it, the object needs a variable (probably public) to hold the key. You wouldn't want two clients to access the same Account with different numbers. Sometimes you can improve a design by replacing name variables with a Map (see below) in the owning object.

Properties. Every instance of a class has the same variables. What happens when every instance needs different variables? Visual Smalltalk Panes, for example, have a host of optional values that can be (but don't need to be) set. Such an object needs a variable to hold a Dictionary mapping names to values. Unlike a Map (see below), a Property Dictionary's values are heterogeneous. You can often improve a design by figuring out what the invariant state is, or finding distinct clusters of properties that can form their own objects (the pattern Whole Value addresses this issue).

Map. Objects hold all the state associated with them. That is, if the system has a number connected with a particular object, that object generally has an instance variable to hold the number. However, when an object is added to the system and it needs to associate new information with an existing object, adding a variable to the existing object would clutter it up. For example, Visual Smalltalk's ObjectFiler and VisualWorks' BOSS associate file offsets with objects. It wouldn't make sense to add a "fileOffset" instance variable to Object. Instead, each ObjectFiler keeps an IdentityDictionary mapping objects to file offsets. Unlike Properties, Maps have homogenous values. Sometimes you can improve designs by moving state out of an object and into a Map, or vice versa.

Current state/strategy. When you use the State Object or Strategy Object pattern, you need a place to put the current state or strategy. VisualWorks' UIBuilder has an instance variable "policy" that holds an object that will create user interface widgets.

Flag. When you have simple variable behavior where an object can be in one of two states, the object needs a variable to hold a Boolean to signify the state. VisualWorks' ParagraphEditor has a flag called "textHasChanged." It is true if the text has been edited. If you have lots of flags, or if a flag shows up in lots of methods, you can improve your designs by introducing a State Object or Strategy (see above).

Pluggable selector/block. Every instance of a class has the same behavior but different state. Sometimes you need a little variable in behavior, but not enough to warrant creating a whole new class. Objects with slightly variable behavior need an instance variable to hold either a Symbol to be performed or a Block to be evaluated. Visual Smalltalk's ListPane has an optional printSelector that is performed on the objects in the list to get the strings to display.

continued on page 20

Jay Almarode

Object security

In single-user Smalltalk systems, the entire image of objects is available to read and write. You can think of the image as your private universe of objects, and as long as you can get a reference to an object, you can send it any message, even one revealing internal state or causing unintended side-effects. As a client implementation technology, single-user Smalltalk provides enough security for most applications, since any object that is manifested in the client is inherently not secure in the first place. As a tool for server implementation, however, multi-user Smalltalk must provide much more security. A server Smalltalk must handle requests from many users running a variety of applications that require different accessibility of objects. This column describes different kinds of object security and examples of how to utilize these security mechanisms.

There are many different ways to achieve object security. The first defense is to prevent unauthorized users from entering the system in the first place. This is usually achieved by requiring users to log in to the server supplying a user name and password. The popular press is full of horror stories about hackers and password-guessing programs, but this mechanism is fairly effective in preventing casual intrusion into a system. To prevent intrusion by hardcore thieves, other barriers must be erected.

In GemStone Smalltalk, a user is represented by an instance of class UserProfile. Among other things, a UserProfile contains the user's unique username, password (encrypted, of course), privileges (that specify which system operations the user is allowed to execute), default segment (to be discussed later), and groups (a set of symbols indicating to which named groups the user belongs). The set of all UserProfiles is maintained in a global set called "AllUsers". One way for an administrator to add a new user to the system is to execute the following:

```
AllUsers addNewUserWithId: #User1
  password: 'pass5698word'
```

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@slc.com.

This operation can only be invoked by an existing user who has the explicit privilege to add new users (more on privileged operations later). In a vanilla GemStone system, there are three pre-existing users: SystemUser, DataCurator, and GcUser. SystemUser is omnipotent and can touch any object without restriction; this user account should only be used for system upgrades. DataCurator is the account used for general system administration functions, such as adding new users or performing a full backup of object memory. The GcUser account is used to control a background process that garbage collects objects during a recent series of transactions. Once a user has logged into GemStone, the user can access his/her UserProfile object by sending the message "System myUserProfile". This can be useful as a programmatic way to determine who is the current user. The class UserProfile defines a number of useful methods including ones to change the user password, add or remove a group name from the set of groups, and list the user's privileges.

Although a user can log into the system, it is still necessary to be able to restrict users from reading or writing particular objects. In GemStone, every object is assigned an instance of class Segment. A segment is an object used to specify authorizations for objects assigned to that segment. The only users who may read or write a particular object are those who are granted authorization to do so by the segment to which the object belongs. A segment has nothing to do with the physical layout or location of an object; it provides a way to group similar objects for security purposes.

A segment provides the means to exercise authorization control over all objects that reference that segment. A segment has an owner; typically the user who created the segment. The segment's owner can change the authorizations that the segment grants. There are three levels of authorization: #read, #write, and #none. As you might expect, #read authorization means that the objects that reference that segment may only be read, #write authorization means that the objects may be read or written, and #none means that the objects cannot be read or written.

A segment can grant authorizations to three designations of users: the owner of the segment, named groups of

users, and all users (commonly called the world). Thus, a segment could be created that grants read and write authorization to the owner, read authorization to a group named #friends, read and write authorization to a group named #trustedFriends, and no authorization to the rest of the world. The following code illustrates how to create a segment with these characteristics:

```
| newSegment |
"create the segment"
newSegment := Segment newInRepository: SystemRepository.
"by default, the owner of the new segment is the user
who created it"

newSegment ownerAuthorization: #write.
newSegment group: #friends authorization: #read.
newSegment group: #trustedFriends authorization: #write.

newSegment worldAuthorization: #none.

"put the segment in a global variable for later use"
UserGlobals at: #trustedFriendsSegment put: newSegment.
```

In this example, the group names must exist in the global set of named groups called AllGroups. AllGroups is a set of symbols that resides in a segment owned by DataCurator; thus, only DataCurator and SystemUser may add or remove named groups.

To assign the segment of an object, simply send the message "assignToSegment: aSegment" to the object. For this operation to succeed, the user must have write authorization for the object's current segment and the new segment. This operation only affects the receiver of the message, not any nested subobjects. To change the segment of the receiver and any logical subobjects, send the message "changeToSegment: aSegment." Any new classes you implement should follow this same convention and reimplement "changeToSegment:" if necessary. In general, you should use "changeToSegment:" to assign a new segment for an object.

When you create new objects, the system automatically assigns them to the system's current default segment. When you initially log in, this is the same as the default segment maintained in your UserProfile. You can change the system's current segment by executing "System currentSegment: aSegment." Any new objects that you create will be placed in the new segment. If given the privilege to do so by your system administrator, you can change your default segment by executing "System myUserProfile defaultSegment: aSegment." This does not take effect until you commit your transaction, log out, and log back in again.

Segments are a flexible way to prevent unauthorized users from accessing specific objects. Support for authorization enforcement must be implemented in the Smalltalk virtual machine for two reasons: It must be implemented at the lowest level of basic object access to prevent users from circumventing the authorization checking, and it must be implemented as efficiently as possible for performance reasons. After all, authorization checking will occur the first time each object is accessed in a transaction.

In some cases, a developer wants to prevent certain objects from being accessed; in other cases, a developer

Table 1. Method access accorded by privileges in GemStone.

Type of Privilege	Privileged Methods
SystemControl	System | shutDown, stopOtherSessions, suspendLogins, resumeLogins
SessionAccess	System | currentSessions, currentSessionNames, stopOtherSessions, descriptionOfSession:
UserPassword	UserProfile | oldPassword:newPassword:
DefaultSegment	UserProfile | defaultSegment:
OtherPassword	UserProfile | password:
SegmentCreation	Segment | newInRepository:
SegmentProtection	Segment | group:authorization:, ownerAuthorization:, worldAuthorization:
Statistics	"methods to gather system-wide statistics"
FileControl	"methods to backup object memory, replicate object memory, create additional file extents, etc."
GarbageCollection	"methods to initiate and control garbage collection"

wants to prevent certain behavior from being executed by specific users. This is easily implemented, either programmatically or by using segments. To implement this capability programmatically, you can explicitly check for a specific user or group of users in the method you wish to restrict. For example, the following method only allows members of the group #trustedFriends to execute the given method:

```
method: Person
driveMyCar

"determine if the current user belongs to the group of
trusted friends"
(System myUserProfile groups includesValue:
#trustedFriends)
    ifFalse: [ self errorNotAllowedToDriveMyCar ].

"perform the act of driving the receiver's car"
```

This technique has the advantage of being visibly explicit in the source code, and you can compose a specific error message when the error condition occurs.

The other technique is to use segments to restrict behavior execution. Since a CompiledMethod must be read before its bytecodes can be executed, placing the CompiledMethod in a restrictive segment can prevent unauthorized users from executing the behavior. The following code shows how to do this.

```
(Person compiledMethodAt: #driveMyCar)
    changeToSegment: trustedFriendsSegment
```

This technique has the advantage of better performance and cannot be circumvented by interrupting program execution and executing internal code by hand.

Previously I mentioned that each UserProfile maintains the privileges for the user. The privileges for a user determine whether the user is allowed to execute certain system functions typically performed by the system administrator. Privileges are more powerful than segment authorization because a user with the appropriate privilege can always change the segment authorization scheme by sending a privileged message. Table 1 lists the types of privileges in GemStone and some of the specific methods that can be invoked when the privilege is granted.

To achieve object security in production systems, the system must support authorization control at the lowest levels. Segments with underlying virtual machine support provide one flexible way to exercise control over access to objects. In a client/server environment, this is one component in overall system security. In addition to exercising authorization control (enforcing a policy of user access), a server also must reliably identify principals attempting to use a system resource. This is called *authentication*. A server Smalltalk can support authentication by utilizing existing schemes, such as kerberos. Together, authentication and authorization are necessary components to building truly secure systems.

SMALLTALK WITH STYLE

reviewed by Jan Steinman & Barbara Yates

SMALLTALK WITH STYLE
Skublics, S., E.J. Klimas, and D.A. Thomas
Prentice Hall Inc., Englewood Cliffs, NJ
ISBN 0-13-165549-3

IN OUR JUNE 1995 COLUMN (Managing project documents) we mentioned the importance of having a style guide within your organization, but we weren't able to suggest any that you could walk into a bookstore and take home. Until now, we have pointed our clients to a 1986 OOPSLA poster paper by Roxie Rochat and an internal Allen–Bradley Smalltalk style guide by Ed Klimas as starting points. This book fills the gap, and is long overdue. It belongs with every Smalltalk project team.

SMALLTALK WITH STYLE is not an "intro to Smalltalk" book. The authors provide references to several other texts for the novice to read to become acquainted with Smalltalk dialects, but this book is mostly dialect-independent, which deserves special applause.

HOW TO USE THE BOOK

The 126 guidelines are grouped into five chapters by topic: naming (guidelines 1–34), comments (guidelines 35–48), formatting (guidelines 49–69), reuse (guidelines 70–90), and tips (guidelines 91–126). A glossary includes basic terms, and the Preface includes some good stuff that should not be skipped.

This book should be read cover to cover the first time, rather than used strictly as a reference book with random access to guidelines via the index, because the prose leading up to the numbered guidelines is essential to grasping the authors' full meaning for the guideline. The examples following most guidelines are required reading too! Finally, there are boxed notes or "tips" sprinkled throughout the chapters that contain very helpful information.

A recent thread in comp.lang.smalltalk about Smalltalk code formatting shows how religious an issue such a topic is; the same is true about a lot of the guidelines in this book. You'll find experienced Smalltalkers have strong opinions both favoring and opposing some of these guidelines. This book's value is not in dictating a standard to be followed, but in collecting in one place the combined wisdom of many Smalltalk programmers who have been writing *and reading* Smalltalk code for over a decade.

Some guidelines may appear deceptively obvious. You may ask yourself "who would *think* of doing such a thing?" when reading some of the bad examples. Even simple guidelines that experienced Smalltalkers take for granted may be novel ideas to brand-new Smalltalkers, and our experience in helping organizations adopt Smalltalk shows that not a single guideline in this book is too obvious.

Some of the guidelines will be misapplied by novices without careful attention to their accompanying examples. One such case is the recommendation to use parentheses to improve readability. Some novices will use this as an excuse to avoid learning the simple Smalltalk precedence rules, but the examples clearly show use and abuse of parentheses. Don't expect this to be a Smalltalk programming instructional manual—it isn't intended to be.

DIPLOMACY VS. DOGMATISM

The writers are not overly dogmatic, recognizing that certain issues are largely a matter of individual taste. For example, on p. 41 they state "There is no absolute way to indent and align Smalltalk code. It is more important to be consistent within your code and, when changing someone else's code, to be consistent with their code."

One unfortunate effect of this diplomacy is that in places the book appears to bend over backwards to avoid stating a preference—there are *eight* code fragments to illustrate the alignment of brackets for Guideline 61, which then states "Choose one way to align brackets in blocks and use it consistently." This statement of a guideline backed up by alternate ways to apply it illustrates one of the points made in the introduction: "This book should be used as the first draft for your own guide to good Smalltalk style...The best guidelines are those that people want to follow because they appreciate the benefit. Blind enforcement of a matter that is of personal taste is not in the interest of the project as a whole."

Although the book is dialect-independent for the most part, we did notice guidelines that are unnecessary in many Smalltalk development environments. For example, a line length limit for source code is suggested, but is

a nuisance in the majority of Smalltalk environments that have resizeable, auto-wrapping code windows. Rather than attempting complete dialect neutrality, it would have been nice if the authors had pointed out the dialect implications of such guidelines.

This suggests the need for additional style rules in your organization that *are* dialect and toolset dependent, for example, specific naming conventions for ENVY applications and version names. Keep in mind (as the authors point out), that this book is a starting place for your organization, not a coding style bible to be adopted as is.

For us, the most enjoyable section of the book is chapter 5: "Tips, Tricks, and Traps." Unlike the guidelines in the preceding sections, which are open to argument and sometimes controversial, these guidelines are mostly universally accepted and of particular importance to beginning Smalltalkers. Some are meant to help you avoid "classic Smalltalk bugs" such as modifying a collection while iterating over it, or forgetting to send #yourself after a succession of cascaded #add: messages to a new collection.

Regrettably, some guidelines in this section cover important topics such as testing in a superficial way. Perhaps it is more important that *something* be said on these topics, albeit very little, with further guidelines in these areas left up to the readers to develop. Once again, this book is a template for you to complete, not an all-inclusive tome.

IMPROVEMENTS

Although we unreservedly recommend this book, we have some wishes for a second edition. It could benefit from the eye of a professional book designer—the layout is a bit unapproachable and difficult to scan, with numerous widows and orphans and a fussy indentation scheme. (Contrast this to an absolutely gorgeous book that is similar in concept: 201 PRINCIPLES OF SOFTWARE DEVELOPMENT, A.M. Davis, McGraw-Hill, 1995, which should be on everyone's shelf.)

There were also a very few outright inaccuracies, such as the glossary definition of a class instance variable. (Class instance variables are *not* shared by instances of the class.)

Also strangely lacking (given two of the authors are at OTI, makers of ENVY) are guidelines regarding the temporal aspects of development, which is in many ways unique to Smalltalk. Certainly more could be said on this topic than "Guideline 120: Avoid modifying the existing behavior of base system classes," without turning into an ENVY ad!

Many of the guidelines lack context and background. This is probably necessary to keep it from ballooning into a general-purpose Smalltalk how-to book, but the lack of "why" behind some guidelines may cause unnecessary resistance to following them.

It should be taken as a tribute to this book that our criticism can be largely boiled down to "more, please!"

EXCERPTS

To give you an idea of what the book is like, here are some of our favorite guidelines, and some that we found most controversial.

- **Guideline 26: Do not use the same temporary variable name within a scope for more than one purpose.** This is one of those "motherhood" guidelines that should be obvious, but we see it violated every day, particularly by new converts to Smalltalk from C or FORTRAN.
- **Guideline 40: Maintain the method comments with as much care as the source code and keep them synchronized.** Amen! This is the "constant accuracy" principle of documentation we describe in our June 1995 column.
- **Guideline 104: Test classes as they are developed** and **Guideline 105: Test components as they are integrated.** Testing is one of the great lies like "The check is in the mail," and "It's done (except for testing)." Expect testing to take on renewed importance as many large, first-generation Smalltalk projects enter their second generation.

More controversial is the example under guideline 59 shown as a "bad choice" because "although the Blue Book...uses this style, most programmers [keep the left bracket on the line with whileTrue:]":

```
[number <= 100] whileTrue:
    ["more code here
    and here"].
```

This is an example of the hidden dialect bias we would have preferred was explicit. The built-in VisualWorks formatter makes your code look similar to this "bad" example, and "most programmers" with a VisualWorks, Objectworks, or Smalltalk-80 background are likely to code in the "bad" style—it should not be discouraged in such shops.

We also find many of the bracketing and indenting styles would be alien (or go missing) to someone whose only Smalltalk exposure was a Smalltalk-80-family image. For example, an important missing Smalltalk-80 guideline is to not line up columns of text with tabs, since the use of different fonts will obliterate the horizontal spacing.

- **Guideline 64: Separate cascaded long key word messages with a blank line or further indent subsequent keywords after the first if the message has multiple keywords.** Example [complies with guideline]:

```
anOrderedCollection
    replaceFrom: 2
    to: 3
    with: #(a b c d e f g)
    startingAt: 3;

    replaceFrom: 7
    to: 8
    with: #(a b c d e f g)
    startingAt: 5.
```

Vertical white space is usually a precious resource, and we

prefer guidelines that conserve it over guidelines that tend to make you scroll. This example also shows the "narrow, non-wrapping window" dialect bias, and a Smalltalk-80 programmer would probably keep each message on the same line, with no extra whitespace.

Guideline 119 regarding lazy initialization does not address its real value: providing state sequence independence. Better would have been a brief discussion of base and derived state and a guideline to "eagerly" initialize base state, while "lazily" initializing derived state, but this once again treads the fine line between a style guide and a design guide.

We look forward to reading the "revised and enlarged" edition that will surely come in the future.

CONCLUSION

Before looking at the book, we asked ourselves some questions that we've heard from novices. Happily, this book answered all that strictly concerned style, but missed some that sit on the line between style and design:

- Should I always write instance methods and create a single instance of a class, rather than treat the class as a global and write only class methods?
- What kind of behavior normally belongs on the class side?
- How do I decide which type of collection to use in my application—anArray, anOrderedCollection, something else?

Admittedly these are not really style questions, but this book does cross the line from style to programming and process guidelines at various times.

We don't envy the authors the task of choosing what to put in and what to leave out. All in all they did an admirable job, and it is with pleasure that we look forward to reading the "revised and enlarged" edition that will surely come in the future.

At least one copy of this book belongs with every Smalltalk project team!

Jan Steinman and Barbara Yates took a month off from their "Managing Objects" column to write this review. They are cofounders of Bytesmiths, a consulting company that has been helping companies adopt Smalltalk since 1987. Between them, they have over 20 years' Smalltalk experience. They can be reached at Barbara.Bytesmiths@acm.org or Jan.Bytesmiths@acm.org.

PROJECT PRACTICALITIES

continued from page 11

	objects that describe how the model will service the requirements specified in a use case.
subsystem	A group of relatively tightly coupled classes that provide some end user functionality, as documented in its contracts.
use case	An English language description of what the system is required to do upon receipt of one type of user request.

Suggested reading

1. Business Object Modeling course materials, Hatteras Software, 1995.
2. Gibson, E. Objects: Born and bred, BYTE MAGAZINE, October, 245–54, 1990.
3. Jacobson, I. et al. OBJECT-ORIENTED SOFTWARE ENGINEERING: A USE CASE DRIVEN APPROACH, Addison-Wesley, Reading, MA, 1992.
4. Lorenz, M. Architecting large projects, THE SMALLTALK REPORT, 4(5):28–29, 1995.
5. Lorenz, M. OBJECT-ORIENTED SOFTWARE DEVELOPMENT: A PRACTICAL GUIDE, Prentice Hall, Englewood Cliffs, NJ, 1993.
6. Lorenz, M. RAPID SOFTWARE DEVELOPMENT WITH SMALLTALK, SIGS Books, New York, 1995.
7. Rumbaugh, J. et al. OBJECT-ORIENTED MODELING AND DESIGN, Prentice Hall, Englewood Cliffs, NJ, 1991.
8. Wirfs-Brock, R., B. Wilkerson, and L. Wiener. DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice Hall, Englewood Cliffs, NJ, 1990.

SMALLTALK IDIOMS

continued from page 14

Cache. Sometimes an object returns the same answer over and over in response to a message. If computing the answer is expensive, you can improve the performance of the system by adding an instance variable to the object to hold the value of the expression. You will have to make sure the value is recomputed if the value of the expression ever changes, and you should only add a cache if a performance profile of the object running under realistic conditions shows that the expression is expensive. The variable "name" in Visual Smalltalk's Behavior (VisualWorks' ClassDescription) is an example. The message "name" could be implemented as:

```
Behavior>>name
    ^Smalltalk keyAtValue: self
```

But because it happens so often, the value of the expression is cached in an instance variable.

CONCLUSION

That's it so far. Looking back at the list, it's obvious that there's still a lot of ground to cover. For example, sometimes variables have their values set when the instance is created and never change. If you've got a favorite trick with instance variables that isn't covered above, drop me a line.

GF/ST—A Smalltalk framework for graphical objects

Jim Haungs

As SMALLTALK BECOMES MORE MAINSTREAM and competes with products like Visual Basic and Delphi, it's nice to see an important product that couldn't possibly be implemented in anything but Smalltalk. GF/ST from Polymorphic Software is such a product. It is designed to solve one of the hardest programming problems: designing and implementing intuitive graphical user interfaces (GUIs) with graphics technology. Menus, toolbars, and common controls notwithstanding, what to do with the client area of your application window is often frustrated by the complexity of the OS platform's graphics primitives. GF/ST is an excellent graphics toolkit that demonstrates the extreme power of Smalltalk in the right hands.

INTENDED AUDIENCE

This product is intended for Smalltalk programmers who wish to add real graphics to their applications. Real graphics go beyond menus and listboxes. Real graphics represent application objects as manipulable graphical objects that support drag-and-drop, and are perceived by the end user as visually representative of the application domain.

BACKGROUND

Manipulating graphics is hard work. Normally, drawing in a coordinate space just leaves a trail of dots. Creating meaningful figures and animating them is a nontrivial task. It is rendered (pun intended) even harder by the wildly differing graphics systems available in modern operating systems. Coordinate systems differ. Graphics primitives are neither standard nor universal. Scrolling and printing are nightmares to implement correctly. Even though all the major operating systems provide generalized driver-based printing, not one of the major Smalltalk platforms implement any kind of native printing support.

Not all platforms use the same coordinate system. For example, Windows considers the origin (0,0) to be the top left corner of the display, while OS/2 considers (0,0) to be at the lower left. In Windows, X increases to the right, and Y increases down, while in OS/2, X increases to the right, and Y increases up. Digitalk exposes these platform differences, implementing generic methods such as rightAndDown:, which are implemented differently on the two platforms. ParcPlace and IBM, on the other hand, completely hide the platform's graphical system by imposing a single, uniform layer of abstraction: SPIM for ParcPlace, and Motif for VisualAge. With so many divergent approaches, source code compatibility across platforms is nearly impossible.

GF/ST addresses the coordinate problem by imposing a lightweight coordinate system over the existing Smalltalk implementation. You are free to use or ignore this system depending on whether portability or platform compliance is more important to you. But much more importantly, GF/ST assists portability by eliminating most of the difficult drawing code, moving it instead into the framework itself.

PRODUCT DESCRIPTION

GF/ST is a framework. It supplies mechanisms for creating drawing spaces for graphical objects. The drawing spaces properly support scrolling and printing; multipage printing of large drawings is also supported. GF/ST is intended to make graphical representations of domain objects as easy as using a ListBox or a TextPane.

The Graphical Object (GO) is the basic behavioral unit in GF/ST. GOs are displayed in a drawing space, which

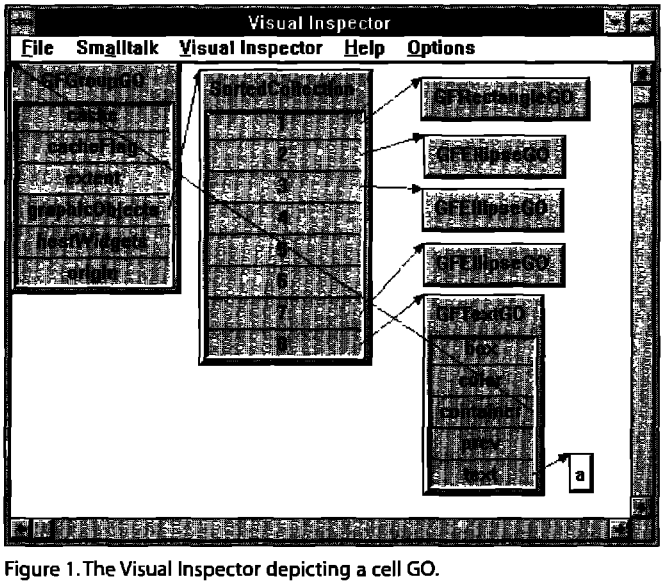


Figure 1. The Visual Inspector depicting a cell GO.

presents an abstract interface to the programmer. This interface disguises most of the underlying platform's graphics primitives, and standardizes the coordinate system across platforms and across Smalltalk products. Through the creation of GOs, you only need to code specialized renderings of your objects; you don't need to manage their display in the drawing, nor their interaction with the platform graphics primitives.

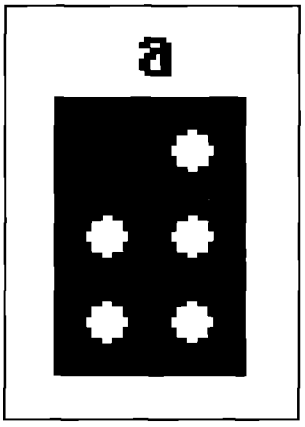


Figure 2. A BrailleCellGO object.

GF/ST supplies many ready-made GO classes: text, lines, ellipses, rectangles, paths, splines, and bezier curves. Lines and polylines can have arrowheads on either or both ends, and paths can be drawn as orthogonal (right-angles only) or straight lines. Group and composite GOs keep track of several objects at once. A group is treated as a single object, while a composite allows separate manipulation of the subobjects and of the composite as a whole. Host widget GOs allow the creation, display, and manipulation of host widgets; use of these constrains the portability of your application, but are invaluable when the widgets are necessary and portability is not a consideration.

A GO can keep track of a single metaobject, which is usually the domain object represented by the GO. Because it is simple to detect and traverse the selected object(s) in a drawing, the metaobject makes it simple to access the corresponding domain object(s).

Because GF/ST is a direct manipulation framework, a large number of classes are supplied to enable direct interaction with graphical objects. The most important notion is the "handle." The GFHandle class abstracts the general behavior of object handles. Subclasses provide more specialized behavior, such as selecting objects, moving them, changing their size and colors, and connecting objects to each other.

One of the nicest things about the GF/ST framework is its built-in support for undoing graphical manipulation operations. GF/ST implements a completely generic Memento framework[1] for remembering previous object states without breaking encapsulation. It also supplies a BoundedStack class that facilitates creation of a limited-size Undo stack; new elements are pushed on the stack until the bound is exceeded, then the oldest ones are discarded as new ones are added. GF/ST also makes use of the platform's object finalization mechanism for properly releasing bitmaps and other resources when a GO becomes garbage. On some platforms (Digitalk), the finalization mechanism must be explicitly loaded into the image. Polymorphic strongly recommends the use of finalization, and I agree with them. After creating the sample application for this article, I ran out of bitmap handles, and I could no longer save the image without causing a walkback. After recreating the image using finalization, I had no further trouble.

GF/ST makes extensive use of the dynamic messaging capabilities of Smalltalk. For example, a GFLocator object is a kind of Message that, when evaluated, yields a point. The GFGraphicObject class implements a default method called "locator" which returns a locator on the center of the object: [GFLocator on: self at: #center]. When the locator is evaluated, it sends the #center message to the receiver, and returns the point at the center of the object. Locators are used to find centers or edges of objects, establish connecting lines between objects, and evaluate constraints when objects are moved or interact with one another. Constraints are blocks of code that establish limits on or between objects. For example, a Position constraint keeps an object from moving outside an established boundary. Connection constraints keep objects and their connecting lines connected and in synch when connected objects are moved.

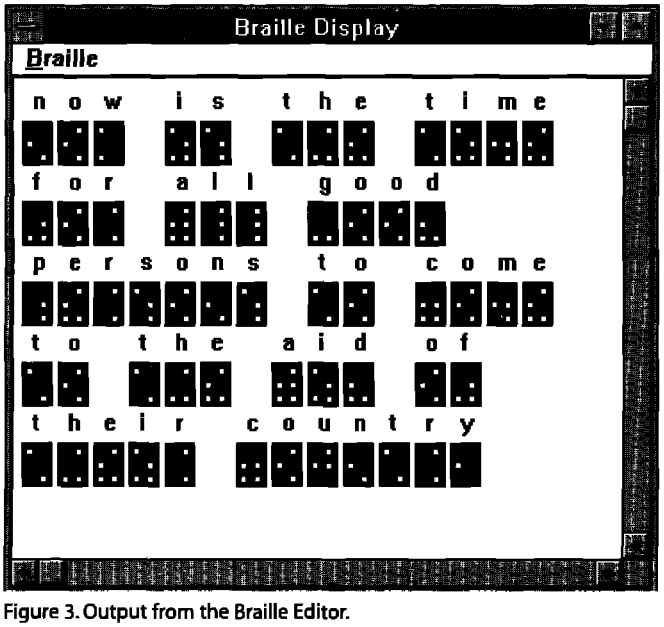


Figure 3. Output from the Braille Editor.

The drawing framework manages the interaction with the user through a set of input "tools" that translate input gestures into object manipulations. For example, selecting objects in a drawing is done with a GFSelectionTool. Tools are sent messages in response to input events, and respond by translating and forwarding messages to the selected objects. Drag/drop is handled simply and elegantly, and the creation and drawing of shapes and the moving and interconnecting of objects could not be simpler. User interactions can also be disabled for output-only applications.

An important aspect of the drawing interface is the internal representation of the objects in the drawing. Determining which of hundreds or thousands of objects in a drawing is under the cursor requires an efficient storage and display list traversal mechanism; a linear search is not sufficient. GF/ST maintains a quad-tree representation[2] of the display list coordinates; searching for an object under the cursor is basically a binary search in progressively smaller quadrants of the display, giving an $O(\log n)$ search time, which scales up nicely even to thousands of objects. An additional efficiency

consideration is the use of double-buffering to repaint a damaged window and eliminate flash and flicker when moving objects around. Objects know their Z-order, and can even be moved behind other objects without flicker. GF/ST supplies the source code for all these traditionally difficult but necessary display mechanisms, and makes them transparently available to your application.

SAMPLE APPLICATION

While visiting friends in Oregon, an interesting application emerged from discussions with my friends' 12-year old son, Nicholas. He had a small metal Braille slate he used to write letters to a blind friend. It has three rows of blank Braille cells and a metal stylus. You put a piece of paper over the cells, and poke the stylus into the paper over the cells to make indentations. Although this is incredibly tedious, (for any serious writing, you'd use a Braille typewriter) it sufficed for quick notes. What made it more difficult was working right to left, and having to look up each letter.

We imagined a little application that would display a complete Braille sentence in a window. We figured that drawing cells was easy, a little like dominoes, and if GF/ST lived up to its promise, displaying the cells should be simple. As it turned out, this was exactly the case. We defined a BrailleCellGO class with an instance variable "char," and a class variable "Alphabet," which contained a dictionary whose keys were letters of the alphabet and whose values were bit patterns representing the letters.

A cell consists of the letter, an enclosing rectangle, and six circles representing the cell pattern. A group GO is used to keep the subobjects together, simplifying tracking the composite object's bounding box. As each object is added to the group, the new boundaries are automatically updated. In real Braille, of course, the letter itself is superfluous, but it facilitates learning as you read the screen, and it looks nice. The following code creates a group GO with a bounding rectangle, six circles, and the text of the character:

```
cell
    | rectGO h v textGO gos |
    h := 18.
    v := 26.

    textGO := GFTextGO text: char.
    textGO translateBy: (h // 2) @ (0 - h).

    rectGO := GFRectangleGO rectangle: (0@0 extent: h@v).
    rectGO
        fillColor: Color white;
        color: Color black.

    gos := OrderedCollection new: 8.
    gos
        add: textGO;
        addAll: self cellPattern;
        add: rectGO.

    ^ GFGroupGO graphicObjects: gos.
```


The six circles are drawn from the mask in the alphabet dictionary. The mask contains 6 booleans: true for black and false for white. The circles in the cell are numbered in columns top to bottom, from 1 at the top left to 6 at the bottom right. The cellPattern method loops through each circle, determines its required fill color and position within the rectangle, and creates an Ellipse GO of the appropriate shape, color, size and position. Note that the size and position are relative to the GO, and that the size and position constants are somewhat arbitrary. GF/ST provides a number of useful units of measurement: twips, inches, millimeters, and pixels; the default value is pixels. These numbers gave a nice display on both VGA and SVGA displays.

```
cellPattern

    | patterns circleGO mask fill circle position |

    circles := #( (0 0) (0 8)
                  (0 16) (8 0)
                  (8 8) (8 16)
                  ).

    patterns := OrderedCollection new.
    mask := self alphabet at: char.
```

```
    1 to: circles size do: [:each |
        (mask at: each)
            ifTrue: [fill := Color black]
            ifFalse: [fill := Color white].

        circleGO := GFEllipseGO new
            width: 0
            color: Color black
            fillColor: fill.

        position := circles at: each.
        circleGO
            setEllipse: (0@0 extent: 5@5);
            translateBy: (2 + position first) @
                        (3 + position last).

        patterns add: circleGO.
        ].

    ^patterns
```

The only other interesting method we had to invent was the layout algorithm for the cells. When the window is resized, we wanted the cells to word-wrap in the usual fashion. Fortunately, the drawing interface knows the size of its visible area, so this simple loop worked right the first time.

```
addText: message

    "Add a sentence to the display."

    | word stream cursor cellExtent cellGO left right |
    cellExtent := self cellExtent.
    cursor := interface visibleRectangle origin.
    left := cursor x.
    right := interface visibleRectangle extent x.

    stream := ReadStream on: message.
    [stream atEnd] whileFalse: [
        word := stream nextWord.

        "Will this word fit on the same line?"
        (word size * cellExtent x + cursor x > right)
            ifTrue: [cursor := left @
                            (cursor y + cellExtent y)].

        word do: [:char |
            cellGO := GFBrailleCellGO for:
                        (String with: char).
            cellGO
                disableInteraction;
                origin: cursor.

            interface addGO: cellGO.
            cursor := cursor right: cellExtent x.
            ].

    cursor := cursor right: cellExtent x.
    ].
```

EXTENSIBILITY

Adding a new kind of GO to the system was simple and natural. Once added, it participated fully in the overall framework, and behaved as expected. I imagine that more complex additions would take correspondingly longer, but there doesn't seem to be anything closed-ended about the system, as long as you stay within the confines of its design. Multimedia applications or 3-D rendering are probably beyond the scope of the product. Simple animations are possible, however, given that double-buffering is already done for you; the source code for several demonstrations of this technique are supplied with the toolkit.

ADDITIONAL TOOLS

Shipping with the system are some excellent free tools, built entirely from the GF/ST framework itself.

The Visual Inspector is similar to Kent Beck's Object Explorer. It displays graphical views of objects. Each object has instance slots, and when a slot is selected the object it points to is displayed with a line connecting back to the parent object. This tool is excellent for visualizing complex object structures, and for teaching the basics of object relationships.

The Drawing Tool is both a simple object painting tool, and a testbed for experimenting with your own Graphic Objects. It lets you add GOs to the interface, manipulate them, and exercise their handles and connection mechanisms. It also demonstrates the use of the Tool and Palette classes for creating drawing environments. Unfortunately, you can't save or restore drawings in the tool, so its use is limited to testing and demonstrations of the framework, but it probably wouldn't be difficult to store the GOs in a file if you really needed to save and restore your drawings.

The 3D Figure Tool demonstrates manipulation of 3-D objects. You can add a cube, pyramid, or tensegrity (sort of a Buckeyball thing) to the drawing area. By manipulating one set of handles, you can control their X, Y, and Z dimensions, and by manipulating the center handle, you can control the pitch and yaw of the objects, causing them to spin around in interesting ways. This tool aptly demonstrates the value of the double-buffering technique, as the move-

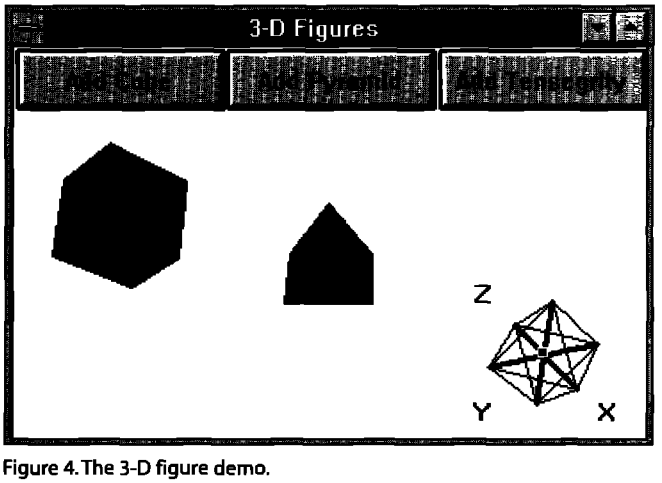


Figure 4. The 3-D figure demo.

SMALLTALK SOLUTIONS 96

March 4–7 1996

NY Marriott Marquis, New York, NY

Where all the talk is Smalltalk

"Finally, a Smalltalk-only conference... Good variety, good speakers. It was refreshing to attend sessions that were not vendor sales pitches."

JILL SEINOLA, SYSTEMS ANALYST, CARGILL, INC.
(SMALLTALK SOLUTIONS '95 ATTENDEE)

Sold out in '95. Register early for '96!

Are You Well Versed in Smalltalk?

Smalltalk Solutions '96, the largest vendor-independent Smalltalk conference and exhibition, provides the comprehensive training needed to keep up with this expanding language. Learn how Smalltalk is being put to more uses, in more companies, in more industries, than any other object-oriented programming language in use today.

Harness the full potential of the Smalltalk programming language at this annual gathering of Smalltalk professionals. Over 1,000 Smalltalk professionals attended last year's premiere in New York. This year's technical program has been expanded and enhanced, with over 30 in-depth classes, panel discussions, hands-on workshops and case studies, taught by Smalltalk's leaders and innovators. Classes focused in 5 educational tracks, allowing you to easily customize your schedule to your specific areas of interest.

A full array of activities includes keynote presenters Adele Goldberg and Glenn Reid; an exhibition hall featuring products and services for Smalltalk in all its dialects; and more. Don't miss this chance to demo the latest products and see Smalltalk in action!

SPONSORED BY: The Smalltalk Report

PRESENTED BY: SIGS CONFERENCES

What You'll Learn at Smalltalk Solutions '96

- Tips and Techniques for Optimizing Smalltalk Applications
- Techniques for Designing Cross-Smalltalk Class Libraries
- Embedded Systems in Smalltalk
- Crossing the Chasm: From Objects to Relational Databases
- Visual Programming Lessons
- Visual Modeling Techniques
- Guide to VisualWorks

- Modeling Under Pressure: Finding & Exploiting Potent Abstractions...Fast
- How to Develop Frameworks
- Patterns for Reuse
- Distributed Smalltalk
- Interfacing Smalltalk to Legacy Systems
- Applications of Meta-Level Programming in Smalltalk

- Testing in Smalltalk
- Errors Patterns Testings

- JP Morgan
- Software 2000
- Bank of Montreal
- Bank of Nova Scotia
- Chubb
- Prudential
- USF&G

- Update on the ANSI Smalltalk Standards Initiative
- Managing Evolutionary Delivery
- Panel Discussion: Creating a Corporate Object Center
- Smalltalk on the Web
- Advice on Succeeding with Objects
- Smalltalk Metrics

Yes!

Please send me more information on Smalltalk Solutions '96

❑ Attending ❑ Exhibiting

❑ Receiving a Free Exhibits Pass

Name _____

Title _____

Company _____

Address _____

_____

City/State _____

Zip/Postal Code _____

Country _____

Phone _____

Fax _____

Fax: 212.242.7578, Phone: 212.242.7515
Mail: Smalltalk Solutions, 71 W. 23rd St., 3rd Flr., NY, NY 10010, e-mail: conferences@sigs.com, WWW: http://www.sigs.com

ADSR

ment of the objects is smooth and completely without flicker, even on a relatively slow computer.

SUPPORTED PLATFORMS

GF/ST is currently shipping on Visual Smalltalk for Win32; support for the Visual Smalltalk OS/2 product is planned. The product is in Beta for VisualWorks and VisualAge; given that these products each use a common graphics model across their respective platforms, the release for these products should support all supported platforms. GF/ST also integrates nicely with other products. In particular, you can build your application interface with WindowBuilder, Parts, VisualAge, or VisualWorks widgets, then simply connect a drawing interface to a graphics pane and you're on your way.

SUMMARY

The GF/ST framework is a robust implementation and factoring of common graphical display techniques. It supplies excellent support for simple 2-D drawings and object manipulation. It is not a CAD framework, nor does it give much support for 3-D graphics or multimedia. However, it goes a long way toward simplifying the representation of dynamic systems as manipulable graphical objects.

The pervasive use of events in GF/ST makes finding bugs and figuring out the flow of control a whole new challenge; tools, locators, and graphical objects send events all over the place, and magic happens. The usual technique of looking for senders does not work, because the events use stored selectors. Perhaps a dynamic graphical browser of the event model would make a nice addition to the tool set.

Supplying the source code to the system makes it easy to extend the framework, and it makes excellent code available for the graphics newcomer to study. In particular, it's nice to be able to read and understand the difficult and platform-dependent process of double-buffering the display.

CONCLUSIONS

If you do any kind of graphics in Smalltalk, you need this toolkit. It's priced reasonably, and if you've ever used Tensegrity, you know Polymorphic's reputation for high-quality products. Even if it doesn't do everything you need, it is easily extensible, and even Smalltalk old-timers could learn a thing or two from the techniques it uses.

Product information

GF/ST is available from Polymorphic Software,
1091 Industrial Rd., Ste. 220, San Carlos, CA 94070;
v: 415.592.6301; f: 415.592.6302; 75010.3017@compuserve.com.

References

1. Gamma, E. et al. DESIGN PATTERNS, Addison-Wesley, Reading, MA, 1995, p 283.
2. Knuth, D. THE ART OF COMPUTER PROGRAMMING, VOL. 3—SORTING AND SEARCHING, Addison-Wesley, Reading, MA, 1975, p. 555.

Jim Haungs is the founder of TeamTools Inc. He specializes in Smalltalk consulting, training, project management, and software development. He has a BSCS from RIT, and an MSE degree from Wang Institute. Jim lives in Boston, and can be reached at jhaungs@teamtools.com.

INFO@SIGS

SIGS Publications, Inc., 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

ARTICLE SUBMISSION

To submit articles for publication, please contact: John Pugh & Paul White, Editors, 885 Meadowlands Dr. #509, Ottawa, Ontario, K2C 3N2 Canada; email: streport@objectpeople.on.ca

PRODUCT REVIEWS AND ANNOUNCEMENTS

To submit product reviews or product announcements, please contact the Editors at the address above.

CUSTOMER SERVICE

For customer service in the US, please contact PO Box 5050, Brentwood, TN 37024-5050; 800.361.1279; Fax: 615.370.4845; in the UK, please contact Subscriptions Department, Tower Publishing Services, Tower House, Sovereign Park, Market Harborough, Leicestershire, LE16 9EF, UK; +44.(0)1858.435302; Fax: +44.(0)1858.434958

SIGS BOOKS

For information on any SIGS book, contact: Don Jackson, Director of Books, SIGS Books, Inc., 71 West 23rd Street, New York, NY 10010; 212.242.7447; Fax: 212.242.7574; email: donald_jackson@sigs.com

SIGS CONFERENCES

For information on all SIGS Conferences, please contact: SIGS Conferences, 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7515; Fax: 212.242.7578; email: info@sigs.com

BACK ISSUES

To order back issues, please contact: Back Issue Order Department, SIGS Publications, 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

REPRINTS

For information on ordering reprints, please contact: Reprint Management Services, 505 East Airport Road, Box 5363, Lancaster, PA 17601; 717.560.2001; Fax: 717.560.2063

ADVERTISING

For advertising information, please contact: Advertising Department, SIGS Publications, 212.242.7447; Fax: 212.242.7574

SIGS HOME PAGE

To access the SIGS Home Page on the World Wide Web: http://www.sigs.com.

SMALLTALK POSITIONS

ParcPlace-Digitalk is seeking experienced Smalltalk instructors and consultants for our world-class Professional Services team. At **ParcPlace-Digitalk** you will work with one of the world's leading development teams, use state-of-the-art products and assist companies on the forefront of adopting object technology in client-server applications.

Requirements for Senior Consultants: solid experience with Smalltalk (3-5 years) and/or PARTS Workbench experience. OOA/D experience and GUI design skills. Mainframe database experience is a big plus. Requirements for instructors: previous training experience in a related field (2-4 years), understanding of OO concepts and Smalltalk.

Positions are available in various sites throughout the U.S. Compensation includes competitive salary, bonuses, equity participation, 401(k) and medical and dental coverage. All positions require travel. **ParcPlace-Digitalk** is an equal opportunity employer.

Please forward your resume to:
Director of Enterprise Services
ParcPlace-Digitalk, 7585 S.W. Mohawk Drive
Tualatin, OR 97062 fax: (503) 691-2742
internet: holly@digitalk.com

The Pathway To Progress HBOC

Meeting the multifaceted information management needs of the ever-evolving healthcare industry requires software solutions that are as advanced as they are flexible: the kind of solutions that **HBO & Company (HBOC)** has been developing for over 20 years. A member of the NASDAQ 100, we have been ranked by *Kiplingers Financial Magazine* as one of the top 15 companies poised for continued success in the year 2000 and beyond.

INFORMATION TECHNOLOGY PROFESSIONALS

Atlanta, GA • Amherst, MA • Minneapolis, MN
Eugene, OR • Salt Lake City, UT • Orlando, FL • Charlotte, NC

We have challenging opportunities for innovative software professionals to analyze, design, develop and implement our highly progressive health care information systems. Requires experience in one of the following:

C/C++ • Smalltalk • Visual Basic
SQL Windows • Sybase • Informix • Mumps

Your expertise will be rewarded with excellent benefits, a competitive salary and the opportunity to advance your career in an environment where promotion from within is the standard. For consideration, forward your resume, indicating location preference, to: **Corporate Recruiting, SEH/ST/1095, HBO & Company, 301 Perimeter Center North, Atlanta, GA 30346. FAX: (404) 392-3050. E-Mail (sharon.hay@hboc.com).** No phone calls, please. EOE M/F/D/V.

HBO&Company

Smalltalk and C++ Experts
30 IMMEDIATE OPPORTUNITIES
Chief Architects • Instructors • Mentors

ObjectSpace, a leader in **the Object-Oriented arena,** has enjoyed 300% growth in the last year, and as a result, has IMMEDIATE opportunities for extraordinarily talented people dedicated to the creation and deployment of advanced technologies. Our areas of interest include: CORBA, OODBMS, Constraint-based Programming, Rule-based Programming, Prototype-based Languages (Classless), as well as Agent Technology, Design Patterns, Biological Systems, Cognitive Science, OOA/OOD and Self.

Our requirements for **EXPERTS** committed to excellence include 4+ years of experience with C++, Smalltalk, Distributed Smalltalk, VisualWorks or VisualAge. In addition, candidates should also possess expertise in Object-Oriented Software Development Methodologies.

We offer competitive compensation, performance-based bonuses and a complete benefits package. For immediate consideration, forward your resume to:

Fax (214) 663-9099

ObjectSpace, Inc., 14881 Quorum Dr., Suite 400, Attn: ST1195, Dallas, TX 75240; jobs@objectspace.com; or call (800) OBJECT1. EOE. http://www.objectspace.com/

ObjectSpace™

Smalltalk RothWell Smalltalk RothWell

SMALLTALK PROFESSIONALS

This is your opportunity to join the finest team of Smalltalk professionals in the country!

RothWell International has challenging projects across the US and abroad.

Excellent compensation and immediate participation in the Employee Stock Plan.

RothWell™ International

(CHECK OUT OUR WEB PAGE!)
http://www.rwi.com/

BOX 270566 Houston TX 77277
(713) 660-8080;Fax (713) 661-1156
(800) 256-9712; landrew@rwi.com

RothWell Smalltalk RothWell Smalltalk RothWell Smalltalk RothWell Smalltalk RothWell Smalltalk RothWell Smalltalk

Smalltalk RothWell Smalltalk RothWell