THE Smalltalk REPORT

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

SIGS Publications Advisory Board
Tom Atwood, Object Design
François Bancilhon, O_2 Technologies
Grady Booch, Rational
George Bosworth, Digitalk
Jesse Michael Chonoles, ACC of Martin Marietta
Adele Goldberg, ParcPlace Systems
Tom Love
Bertrand Meyer, ISE
Meilir Page-Jones, Wayland Systems
Cliff Reeves, IBM
Bjarne Stroustrup, AT&T Bell Labs
Dave Thomas, Object Technology International

The Smalltalk Report Editorial Board
Jim Anderson, Digitalk
Adele Goldberg, ParcPlace Systems
Reed Phillips
Mike Taylor, Digitalk
Dave Thomas, Object Technology International

Columnists
Jay Almarode
Kent Beck, First Class Software
Juanita Ewing, Digitalk
Greg Hendley, Knowledge Systems Corp.
Tim Howard, RothWell International
Alan Knight, The Object People
William Kohl, RothWell International
Mark Lorenz, Hatteras Software, Inc.
Eric Smith, Knowledge Systems Corp.
Rebecca Wirfs-Brock, Digitalk

SIGS PUBLICATIONS GROUP, INC.
Richard P. Friedman, President and CEO
Hal Avery, Group Publisher

Editorial/Production
Kristina Joukhadar, Editorial Director
Elisa Varian, Production Manager
Brian Sieber, Art Director
Seth J. Bookey, Production Editor
Margaret Conti, Advertising Production Coordinator
Dan Olawski, Editorial Production Assistant

Circulation
Bruce Shriver, Jr., Circulation Director
John R. Wengler, Circulation Manager
Kim Maureen Penney, Circulation Analyst

Advertising/Marketing
Gary Portie, Advertising Manager, East Coast/Canada/Europe
Jeff Smith, Advertising Manager, Central U.S.
Michael W. Peck, Advertising Representative
Kristine Viksnins, Exhibit Sales Representative
212.242.7447 (v), 212.242.7574 (f)
Diane Fuller & Associates, Sales Representative, West Coast
408.255.2991 (v), 408.255.2992 (f)
Sarah Hamilton, Director of Promotions and Research
Wendy Dinbokowitz, Promotions Manager for Magazines
Caren Polner, Senior Promotions Graphic Designer

Administration
Margherita R. Monck, General Manager
David Chatterpaul, Senior Accounting Manager
James Amenuvor, Business Manager
Michele Watkins, Assistant to the President

SIGS PUBLICATIONS
Publishers of Journal of Object-Oriented Programming, Object Magazine, C++ Report, The Smalltalk Report, The X Journal, Report on Object Analysis & Design, Objects in Europe, and Objekt Spektrum (Germany)

Table of Contents

March–April 1995 **Vol 4 No 6**

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar–Apr, July–Aug, and Nov–Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1995 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is Flatly prohibited. Material may be reproduced with express permission from the publisher. Second Class Postage Pending at NY, NY and additional Mailing offices. Canada Post International Publications Mail Product Sales Agreement No. 290386.
Individual Subscription rates 1 year (9 issues): domestic $79; Mexico and Canada $104, Foreign $119; Institutional/Library rates: domestic $119, Canada & Mexico $144, Foreign $159. To submit articles, please send electronic files on disk to the Editors at 885 Meadowlands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).
POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, P.O. Box 2027, Langhorne, PA 19047. For service on current subscriptions call 215.785.5996, 215.785.6073 (fax), P00976@psilink.com (email). PRINTED IN THE UNITED STATES.

Editors' Corner

While attending the OOP '95 conference in Munich, it was impossible not to notice that the move towards Smalltalk and Smalltalk-based visual programming environments that we have experienced in North America is also beginning to take place in Europe—and most notably in the German-speaking countries. A "Smalltalk-Abend" (the German equivalent of a birds-of-a-feather session) attracted more than 100 conference attendees. The session was in German, but thanks to a friendly translator from Daimler-Benz, it was possible to discern that many groups have significant Smalltalk projects under way. A number of attendees raised the idea of a European Smalltalk Solutions in the near future; now that sounds like a nice idea. How about Paris in the spring?

Richard Helm of "Gamma, Helm, Johnson, and Vlissides"* fame gave a wonderful talk entitled "Using design patterns: Elements of reusable architectures" at the conference. Patterns has been one of the hottest OO topics in North America for the last twelve months or so and it was "standing room" only in Europe too. As Smalltalk educators, we are very much attracted to the notion of patterns. In our work, we find it a real challenge to transfer the knowledge and experience we have acquired from many years of programming in Smalltalk to new developers climbing the Smalltalk mountain. Smalltalk gurus have a sixth sense about which reusable design or micro-architectures can be used to help solve a particular problem. They have an arsenal of reusable patterns up their sleeves and the experience to know when and how to apply them. Patterns provide a way of codifying and communicating these recurring design structures to others.

The work of Gamma *et al.* provides us with a vocabulary for patterns to aid communication and the classification system (e.g., creational, structural, behavioral,...). Their introduction gives us a starting point for cataloging patterns in pattern handbooks and identifying similarities between patterns. Smalltalk people may find it a little strange at first to think of Model-View-Controller in terms of *Observer Pattern* or the use of Wrapper classes (as in VisualWorks) to embellish visual components as an instance of the *Decorator Pattern.*

JOHN PUGH

PAUL WHITE

Because a common vocabulary is so vital, let's hope that the names selected for the patterns by Gamma *et al.* are universally accepted. Since most patterns are language independent, a common vocabulary will allow us to communicate with our brethren who use C++, for example.

We would like to encourage you to contribute patterns which you have found useful in your work. Most of the examples of patterns in the Smalltalk literature draw their examples from the Smalltalk system itself.† While these are interesting, it would be even more valuable to see examples of patterns from domains such as banking, insurance, power systems, or telecommunications. Also, much of the discussion to date has dealt with "how to build something" patterns. As Kent Beck stated at Digitalk's DevCon last year, the patterns that will really provide leverage are the "how to use something" patterns.

There seems to be a fine line between what are called idioms, patterns and frameworks. Idioms are more often than not language dependent and tend to be more concrete manifestations of programming techniques in a particular language than the more abstract higher level design structures we describe as patterns. Similarly, a fine distinction can be made between frameworks and design patterns in that frameworks are more concrete than design patterns. Frameworks represent a reusable architecture for some domain but, unlike the more abstract design patterns, they provide a partial implementation of that architecture. Design patterns, idioms, and frameworks all assist us in transferring good design and programming practices to a new generation of developers. We'd like to hear from you whether you have an idiom, pattern or framework to describe! Kent's columns will give you plenty of inspiration.

Lest we all get carried away in the euphoria of patterns, let's take note that recognizing brand new patterns is a very challenging activity and that, even when we are armed with a catalog of patterns, recognizing situations when a pattern can be applied is not always easy; particularly for beginners. We already see designers and programmers trying to "force" patterns onto problems for which they are not suitable.

Enjoy the issue.

* Gamma, E., et al. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, 1995.

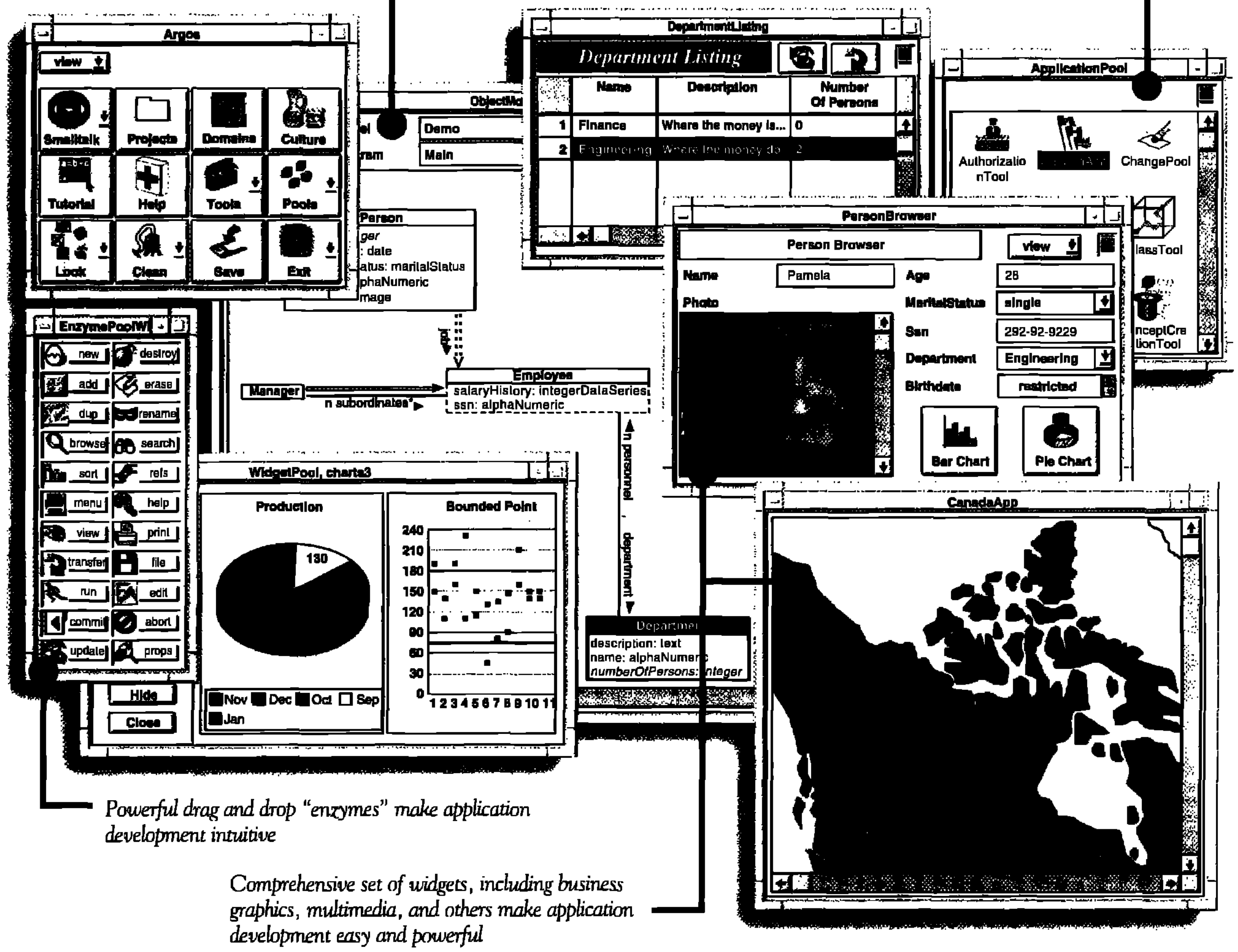
† Lalonde, W. and Pugh, J. COMMUNICATING REUSABLE DESIGNS VIA DESIGN PATTERNS, JOURNAL OF OBJECT-ORIENTED PROGRAMMING, Feb. 1995.

Introducing Argos

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

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

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



Powerful drag and drop "enzymes" make application development intuitive

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

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

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

Leading organizations — in industries from telecommunications to finance — are using Argos to deliver business-critical applications. Find out how Argos can help you deliver your critical applications in weeks, instead of years.

Contact us today at 1-800-VERSANT, ext. 415 or via e-mail at info@versant.com

VERSANT
The Database For Objects™

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

Building a Gopher from sockets and widgets

Patrick Mueller

NO, THIS ISN'T an article on cyberbiology. I'll be describing how to build an Internet Gopher client within IBM Smalltalk, using the Widgets user interface programming model and the sockets communications protocol used on the Internet. Plan on learning a couple of things after reading the article:

- what an Internet Gopher looks like
- how the Gopher protocol works
- an introduction to socket programming in IBM Smalltalk
- an introduction to user interface programming in IBM Smalltalk.

WHAT IS GOPHER?

First let's talk about Gopher. If you don't already know what Gopher is, the best way to learn is to play with a Gopher client. Ask your local Internet guru for a test drive. In case you don't have one locally available, here's a description.

You start up a gopher client by running the gopher program and specifying a gopher server to start at. The gopher client will contact the server and ask for a list of items. Those items will be displayed by the gopher client, with some kind of user interface for you to select items (see Figure 1 for an example of my Gopher client displaying a menu of items). Each item in the list is typed: common types are:

- another gopher menu
- a text file
- a graphics file

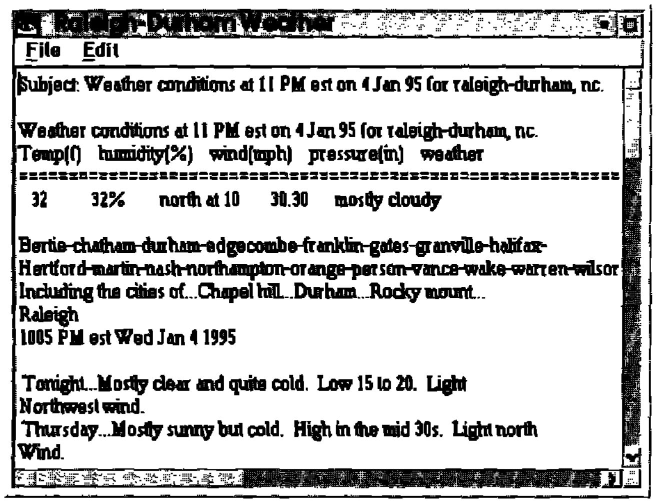
When you select one of the items, the gopher client will send a command to get the appropriate type of item from the server and return it to the client. In the case of another gopher menu, another menu will be displayed. In the case of a text file, a text editor will be displayed with that text (see Figure 2 for an example of my Gopher client displaying a text file). You get the picture. It's a very simple program to use. And there's lots of information available. Within IBM, for example, we have more than 60 well-known gopher servers, servicing more than 8000 different menu items (sorry folks, this is primarily IBM-only information).

THE GOPHER PROTOCOL

The protocol a gopher client and server use to exchange information is one of the simplest used over the Internet. To get a gopher item from a server, the client needs to know three pieces of information: the name of the server (TCP/IP hostname), the TCP/IP port for the server, and a selector string. Most gopher servers use port 70. The main menu for a gopher server uses an empty selector string. So, to get the main menu from a gopher server, you really only need one piece of information: the name of the gopher server.

The client creates a new TCP/IP socket and connects it to the server at the port requested. It then writes the selector string, followed by carriage return and linefeed to the socket. At this point, it starts reading from the socket, terminating when the socket is closed by the server. The data returned by the server is interpreted depending on the type of the item. After receiving the data, the client closes the socket.

The most common type of gopher item is a menu; that is the type of item returned for the main gopher server, when passed an empty selector string. The data returned for a menu



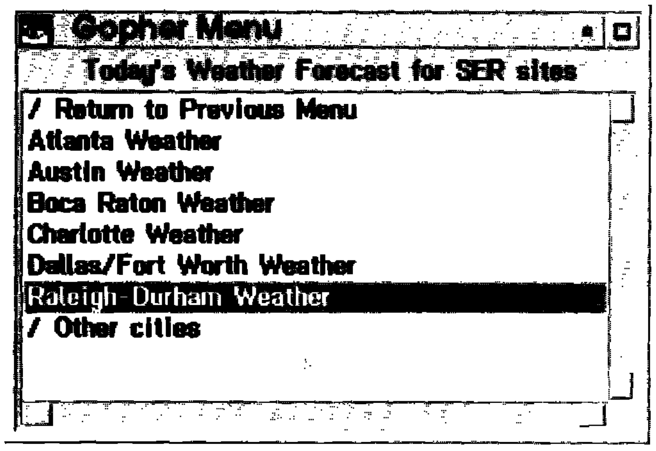
```
Subject: Weather conditions at 11 PM est on 4 Jan 95 for raleigh-durham, nc.

Weather conditions at 11 PM est on 4 Jan 95 for raleigh-durham, nc.
Temp(f)   humidity(%)   wind(mph)   pressure(in)   weather
============================================================
  32       32%      north at 10     30.30    mostly cloudy

Bertie-chatham-durham-edgecombe-franklin-gates-granville-halifax-
Hertford-martin-nash-northampton-orange-person-vance-wake-warren-wilsor
Including the cities of...Chapel hill...Durham...Rocky mount...
Raleigh
1005 PM est Wed Jan 4 1995

 Tonight...Mostly clear and quite cold.  Low 15 to 20.  Light
Northwest wind.
 Thursday...Mostly sunny but cold.  High in the mid 30s.  Light north
Wind.
```

Figure 1. Example Gopher text window.

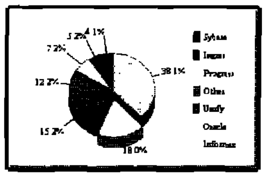


```
/ Return to Previous Menu
Atlanta Weather
Austin Weather
Boca Raton Weather
Charlotte Weather
Dallas/Fort Worth Weather
Raleigh-Durham Weather
/ Other cities
```

Figure 2. Sample Gopher menu.

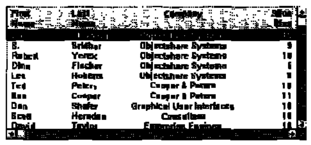
Components for Smalltalk, VA, & PARTS

Tools from Objectshare help you get your applications finished fast. WidgetKit™ components are available for WindowBuilder Pro, VisualAge, and the PARTS Editor in Visual Smalltalk.



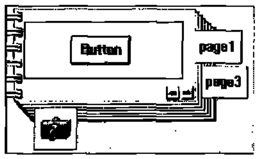
Charts & Graphs

WidgetKit/Business Graphics provides a wide variety of charts and graphs for your applications. The basic types include bar, pie, area, line, tape, high-low-close, and more. Most types have options such as 2-D and 3-D stacked, exploded, absolute and %. Capabilities include printing, autoscaling, control of fonts, colors, label, grids, and more.



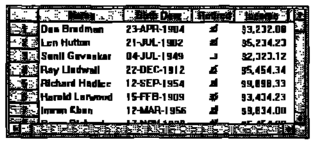
20 New Controls

Subpanes has 20 must-have controls. Columnar list box, hierarchical list box, table pane, bitmap pane, 3-D frames, and a selection of buttons, sliders, and gauges. The table pane and columnar list box support collections of objects and let you control headers, justification, fonts, colors, disabling of items, and more. Date, time, and number editors provide validation and easy editing.



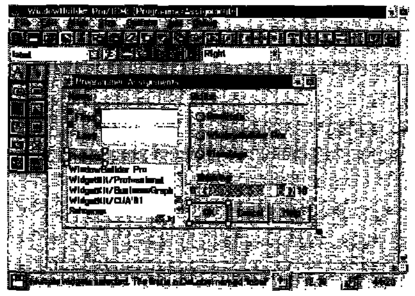
CUA'91 Controls

WidgetKit/Business/CUA'91 has the full suite of CUA'91 controls for your applications. It includes notebook, slider, spin button, and value set.



Spreadsheets & More

WidgetKit/Business Professional provides spreadsheets, multi-column list boxes, table editor, and graphic viewers for BMP, PCX, and GIF. Also includes input validation (12 types), flexible graphical/3-D buttons, file system widgets, spreadsheet printer and more. Spreadsheet power is similar to Excel, with formulas, drag and drop, load, save, and more. Virtual spreadsheet too.



Build UIs Fast

WindowBuilder™ Pro is a pair of interactive tools that lets you build polished user interfaces fast. Versions for VisualAge & IBM Smalltalk, for Visual Smalltalk & PARTS Programmers that use VA or PARTS will generate UIs faster and have more control in WBPro. Bypass the wires. Generate VA or PARTS components. Smalltalk programmers will find that WBPro saves them from building UIs in code.

Select controls from a palette. Place and edit them interactively. Build composites of controls. Includes autosizing, automatic alignment, control of fonts, menus, colors, tabbing order, and more.

PRODUCT AVAILABILITY AND PRICES

Use Objectshare product with...	**Visual Smalltalk and/or Visual Smalltalk Enterprise**				**Smalltalk/V**	**IBM Smalltalk**				**VisualAge**			
and the following app. building environment...	**WB Pro/V**		**PARTS Editor**		**WB Pro/V**	**WindowBuilder Pro**				**VA Composition Editor**			
on this platform...	WINDOWS	OS/2	WINDOWS	OS/2	WIN 16	WINDOWS STANDARD	WINDOWS TEAM	OS/2 STANDARD	OS/2 TEAM	WINDOWS STANDARD	WINDOWS TEAM	OS/2 STANDARD	OS/2 TEAM
WindowBuilder Pro	$495	$495	a	a	$295	$495	$695	$495	$695	b	b	b	b
WidgetKit/Professional	$395 *(Q2'95)*	i	$495 *(Q2'95)*	i	$395	$495 *(Q2'95)*	$795 *(Q2'95)*	i	i	$495	$795	i	i
WidgetKit/ Business Graphics	$495 *(Q2'95)*	$495 *(Q2'95)*	$495 *(Q2'95)*	$495 *(Q2'95)*	$495	$495	$795	$495	$795	$495	$795	$495	$795
Subpanes	$235	$235	$295 *(Q2'95)*	$295 *(Q2'95)*	$235	i	i	i	i	i	i	i	i
WidgetKit/CUA'91	i	$295 *(Q2'95)*			i								

a Order WBPro/V for corresponding Visual Smalltalk platform.
b Order WBPro for corresponding IBM Smalltalk platform.
i Planned, inquire about availability and pricing.

No runtime fees are required for applications developed with these products. Free support for the first 90 days. All products include complete documentation. WidgetKit products for Smalltalk/V Win32 are available, please inquire. Support subscription available.

OBJECT SHARE SYSTEMS INC.

Objectshare Systems, Inc.
5 Town & Country Village
Suite 773
San Jose, CA 95128-2026
Fax 408-970-7282
CompuServe 76436,1063

Call to order today (408) 970-7280

Or call for free info. 9 AM to 5 PM PST, M-F. 30-day money-back guarantee consists of a set of lines, separated by carriage return and line-feed characters, up to the line that contains nothing but a period ("."). For each line, the first character is a type indicator. The rest of the string is tab delimited. The field after the type indicator is a string to display in the user interface for the item. The next field is the selector. The next is the server name, and the last is the port. The selector, server name and port are all used to get that item. The type indicator (first character in the line) indicates what type of item this is (e.g., menu, text, graphics, etc).

"The best way to learn is to play with a Gopher client"

For the text type, the data returned from the server is just the text to display back to the user. For the graphics type, the contents of a GIF or TIFF file might be returned.

CLASSES IMPLEMENTED

First a little class hierarchy creation. We're going to implement a class called GopherItem, with a subclass for each of the gopher data types. GopherItem is defined with instance variables:

display	description of the item to display to the user
selector	selector to send to the server
host	name of the server
port	port number for the server
data	data returned by the server

Besides defining accessors for these variables GopherItem contains:

- the logic to get the data for an item from a server
- the logic to parse a line of menu information returned from the server
- the logic to determine what type of data a particular line is

We'll create the following subclasses of GopherItem:

- GopherItemMenu to display menus
- GopherItemSearch to prompt for a search string, and display a resulting menu (used to search phone books, for instance).
- GopherItemText to display textual information
- GopherItemUnknown to handle Gopher data our client does not understand.

We'll create a class named Gopher to manage the user interface.

USING TCP/IP SOCKETS IN IBM SMALLTALK

Now we'll actually implement the main processing of the gopher client: connecting to a server to get the some data.

If you aren't already familiar with sockets, here's a brief overview. Sockets are a lot like file handles. You open them, read from them, write to them, and close them. Except, instead of having a disk drive to read from or write to, there's another program over the network who is reading what you are writing, or writing what you are reading. And instead of specifying a file name, you specify a host name and a port number to connect to.

```
getData
    "Set data instance variable to the data returned for the gopher
menu item."

    | abtHost abtPort abtSock dataChunk allData |

        self data: ''.

        self port isNil ifTrue: [self port: 70].
        self selector isNil ifTrue: [self selector: ''].

    abtHost := AbtTCPInetHost getHostByName: self host.
        abtHost isCommunicationsError ifTrue: [^nil].

    abtPort := AbtTCPPort usingHost: abtHost portNumber: self port.
        abtPort isCommunicationsError ifTrue: [^nil].

    abtSock := AbtSocket newStreamUsingPort: abtPort.
        abtSock isCommunicationsError ifTrue: [^nil].

        abtSock bufferLength: 8192.

        (abtSock connect) isCommunicationsError ifTrue: [^nil].

    (abtSock sendData: (self selector, Cr asString, Lf asString))
        isCommunicationsError ifTrue: [^nil].

    allData := ''.
    [abtSock isConnected] whileTrue: [
        dataChunk := abtSock receive.
        dataChunk isCommunicationsError ifTrue: [^nil].
        allData := allData, dataChunk contents asString
        ].

    abtSock disconnect.
    self data: allData.
```

Example 1. GopherItem>>getData method.

The logic to get the data for an item from the Gopher server is implemented in the instance method GopherItem>>getData. GopherItem supplies instance methods to return the host, port and selector of the Gopher server we want data from. Example 1 contains the Smalltalk code for this method.

Data is first initialized by getData to an empty string, and defaults its port and selector if not set. It then obtains an instance of AbtTCPInetHost, AbtTCPPort, and AbtSocket from the host and port information. AbtTCPInetHost is used to convert host names into TCP/IP addresses. AbtPort is used to associate a tcp/ip port with a TCP/IP address. AbtSocket is used to manage the actual socket, based on the AbtPort it was created with.

Up to this code, we have defined what we want to connect the socket to, but haven't actually connected it. Sending connect to the socket will cause the socket to connect to the server.

The Exciting WORLD of Object-Oriented Technology

Technical Resource Solutions is a nationwide consulting firm catering to information intensive clients. TRS provides Object Oriented analysis/design and implementation, conversion, training, mentoring, project reviews, and support using Smalltalk and other leading edge tools and methodologies.

TRS offers comprehensive consulting services that include:

- **Contract Staffing & Permanent Placement**
- **Software Development & Training**
- **Business Process Re-Engineering**
- **Joint Product Development & Marketing**

TRS has helped many successful Fortune 1000 companies to overcome their systems challenges. Our consultants have developed expertise working on large development projects interfacing with management, developers and users alike. TRS can manage all or part of a development project **effectively, efficiently & economically**.

TRS has close partnerships with ParcPlace for VisualWorks and Texas Instruments for IEF. Our strong partnerships allow us to work very closely with these vendors, providing continuous feedback to improve their tools and product, the building blocks of your systems.

You can examine our services by calling **1-800-801-1TRS** and one of our professional representatives will help you plan new system solutions.

TRS

TECHNICAL RESOURCE SOLUTIONS

3900 W. Alameda Ave., Suite 1700 • Burbank • CA 91505
818 / 972-1744 • Fax 818 / 972-1685 • email 74601.3324@compuserve.com

Automatic Documentation - Easier Than Ever

With Synopsis for Smalltalk/V Development Teams

New! Release 2.0 Builds Help Files

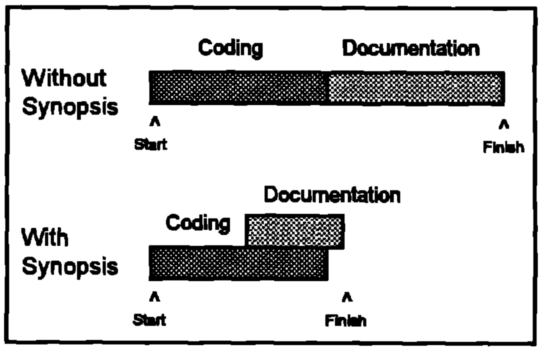
Synopsis produces high quality class documentation automatically. With the combination of Synopsis and Smalltalk/V, you ***cut development time*** and ***eliminate the lag between the production of code and the availability of documentation.***

Synopsis for Smalltalk/V

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

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

Development Time Savings



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

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

Building a Gopher

Once connected, we send the selector, followed by carriage return and linefeed, then start reading from the server. As long as the socket is connected, we receive the data from the socket and append it to the end of a local variable. When the server finally closes the socket, isConnected will return false. At this point, we close our end and set the data to the entire string returned from the server.

That's the only TCP/IP related code in the entire gopher client. Each gopher item subclass is responsible for interpreting the data received by this code.

USING WIDGETS IN IBM SMALLTALK

Widgets are the programming interface used for user interface programming in IBM Smalltalk. The terminology comes from Motif, upon which the user interface classes are based on. If you're already familiar with Motif Widgets, I have real good news for you—you're already familiar with IBM Smalltalk's Widgets. If not, don't worry—it's a simple and elegant model.

Widgets are used to model all the visual building blocks needed to create a user interface:

- the shell, to contain the frame, system menu, title bar, and minimize/maximize buttons
- main windows to contain the menu bar
- forms and bulletin boards to contain other widgets
- core widgets like buttons, list boxes, text fields, etc.

Each type of widget is a subclass of CwWidget. There are two primary ways to change the behavior of a widget: through resources and through callbacks.

Resources control the basic state of a widget, such as color and font information. Most widgets have a unique set of resources associated with them, and resources are inherited down the CwWidget class hierarchy. Resources are set and queried via instance methods named after the resource. For instance, to query the width of a widget, send it the message width.

Callbacks are a way to get feedback from the user when they interact with the system. Like resources, each widget class implements its own set of callbacks, which are inherited down the CwWidget class hierarchy. As an example, to be notified when the user presses a button, the following code may be used.

```
buttonWidget
    addCallback: XmNactivateCallback
        receiver: self
        selector: #pressed:clientData:callData:
        clientData: nil.
```

Each callback has a name, in this case XmNactivateCallback. This particular callback is invoked when a button is pressed. When the button is pressed, the message pressed:clientData:callData: will be sent to the object that executed this code (since the receiver was specified as self). The callback message is passed the widget, the client data specified when the callback was added (in this case, nil), and an object containing information specific to this type of callback.

Ok, so those are the basics, let's dive right into our gopher client. Our user interface is going to be a new window, with a read-only text field at the top giving a description of the cur-

```
listWidget items: (OrderedCollection with: a with: b with: c)

createWindow
    "Create the gopher menu window"

    | shell main form text list |

    shell := CwTopLevelShell
        createApplicationShell: 'gopherMenu'
        argBlock: [:w| w
            title: 'Gopher Menu';
            width: (CgScreen default width) // 2
        ].

    form := shell
        createForm: 'form'
        argBlock: nil.
    form manageChild.

    text := form
        createLabel: 'label'
        argBlock: [ :w | w
            labelString: ' '
        ].
    text manageChild.

    list := form
        createScrolledList: 'list'
        argBlock: [ :w | w
        selectionPolicy: XmSINGLESELECT;
            visibleItemCount: 20
        ].
    list manageChild.

    text setValuesBlock: [:w | w
        topAttachment: XmATTACHFORM; topOffset: 2;
        leftAttachment: XmATTACHFORM; leftOffset: 2;
        rightAttachment: XmATTACHFORM; rightOffset: 2
        ].

    list parent setValuesBlock: [:w | w
        topAttachment: XmATTACHWIDGET; topWidget: text;
        bottomAttachment: XmATTACHFORM; bottomOffset: 2;
        leftAttachment: XmATTACHFORM;leftOffset: 2;
        rightAttachment: XmATTACHFORM; rightOffset: 2
        ].

    list
        addCallback: XmNdefaultActionCallback
        receiver: self
            selector: #selectItem:clientData:callData:
        clientData: nil.

    shell realizeWidget.

    self listWidget: list.
    self textWidget: text.
    self shellWidget: shell.

    self menuStack: OrderedCollection new.
```

Example 2. Gopher>>createWindow method.

Database Solution for Smalltalk/V

A class library for ODBC Database Access

ODBTalk

- ODBC 2.0 support for 50+ databases
- OO to RDBMS mapping
- Native data type support
- Online help, source included, no runtime fees

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

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

Client Server Solution for Smalltalk/V

A class library for Windows Sockets Development

Socktalk

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

Available for Win16, Win32s, Win-NT

LPC *Consulting Services*
Tools for the Smalltalk developer

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

rent gopher menu item we're viewing and a list box containing the items available on this gopher menu. Gopher text items will be displayed in a separate window (a Workspace), which is not described here.

The widgets we'll need are:

- a shell, to contain the frame, system menu, title bar, etc.
- a form, to contain the text field and list box
- a text field
- a list box

A form is a widget that knows how to resize the widgets contained within it. We're using it to allow the user to resize the window and have the widgets contained in the form automatically resize themselves.

As mentioned before, we'll be implementing a class called Gopher to handle the user interface. Gopher is defined with the following instance variables:

data	to hold the data associated with the menu-items (ie, the selector, server, and port of the menu items)
listWidget	to hold our list box widget
textWidget	to hold our text field widget
shellWidget	to hold our shell widget
menuStack	to keep track of where we came from, so we can backtrack through the gopher.

The instance method createWindow is used to create and setup all the widgets. Example 2 contains the code for this method.

Building a Gopher

The first thing we do is create a shell window. This is done with the message CwTopLevelShell class>>createApplicationShell:argBlock:. The first parameter is the name of the widget. All widgets have a name, which is usually not externally visible to the user. The second parameter is a block used to set resources when the widget is created. In this case, we're going to set the title of the shell window, which will be placed in the frame's window bar, and the width of the frame, making it half the size of the screen.

You might be wondering why we use the argBlock parameter (and the setValuesBlock: later in the code) to set our resources.

The message to create the shell widget could also have been written as:

```
shell := CwTopLevelShell
    createApplicationShell: 'gopherMenu'
    argBlock: nil.
shell
    title: 'Gopher Menu';
    width: (CgScreen default width)  // 2
```

In IBM Smalltalk, widget resources are "hot"—that is, when changed, the user interface is immediately updated. In order to allow the system to optimize changes to a widget, the argBlock parameter and setValuesBlock: message are the recommended ways to set resource values for a widget.

Next, we create the form. Most widgets are created using widget creation convenience methods named createXXXX:argBlock:, where XXXX is the type of widget to create. These messages are sent to the widget that will contain the widget to be created. In this case, we'll create a form with the name form, and don't need to set any resources.

After the widget is created, we send it the message manageChild. This is a Motif-ism, which you don't need to be too worried about, but will need to call it after creating your widgets. Managing and mapping widgets allows some interesting behaviors, such as causing widgets to instantly appear and disappear as needed.

Contained within the form will be a label widget, created with createLabel:argBlock:. We'll set the initial text of the label to a blank string.

Also contained within the form is a list box, created with createScrolledList:argBlock:. The selectionPolicy resource sets the type of selection allowed - -single select, multiple select, etc. The visibleItemCount resource sets the initial size of the list box, eg. the list box will be sized to contain 20 items.

As mentioned previously, we're using a form so that the widgets inside the form can be automatically resized. In order to make this happen, we have to attach the widgets to the form.

For each of top, bottom, left and right, there are three basic types of attachment:

- attach the widget to the edge of the form
- attach the widget to a position in the form (position based on 100—setting to position 50 attaches the widget to the middle of the form)
- attach the widget to another widget.

In our case, we attach the label widget to the top, left, and right sides of the form. We don't need to attach the bottom, since a label field has a default height (the height of the font the text is being displayed in). The list box is attached to the bottom, left and right sides of the form, and it's top is attached to the label widget. Note also an offset is specified for aesthetic reasons (to keep the user interface from looking as if it's all crammed together).

Now when the window is resized, the label and text windows will have their widths changed automatically, since they are attached to the sides. When the height changes, the label won't change size but the list box will, since it's attached to the label widget at the top and the form on the bottom.

As a further example of attachments, if we change the label widget to attach the bottom as in:

```
bottomAttachment: XmATTACHPOSITION;
    bottomPosition: 25;
```

the label widget would take the top 25% of window and the list box would have take the bottom 75%.

Note that for the listbox, we send setValuesBlock: to the parent of list, not list itself. This is because a CwScrolledList widget is a list box with a set of scrollbars around it. It's the widget (which we don't see) that contains the list box and scrollbars that we need to attach to the form.

To be able to execute some code when an item in the list is selected, we need to use a callback. In the previous code, the XmNdefaultActionCallback is used on the list widget. This callback is invoked when an item is double-clicked in the listbox. We specify sending the message selectItem:clientData:callData: to self. The actual callback is implemented as follows:

```
selectItem: widget clientData: clientData callData: callData
    "Callback sent when an item is selected.  Open a viewer
    for the appropriate GopherItem subclass for the item."

    | pos menuItem |
    pos := callData itemPosition.
    menuItem := (self data) at: pos.
    menuItem view: self.
```

callData is an object containing information specific to this callback; in this case, sending it itemPosition answers the one based offset of the item within the menu that was selected. The data instance variable of Gopher contains an ordered collection of GopherItem instances returned from the server. We just get the appropriate menu item and tell it to view itself.

Finally, we tell the shell to realize itself, which causes it to be displayed, and set our instance variables.

The contents of the listbox are maintained with the items resource. The data associated with this resource is an OrderedCollection of Strings. For instance, to set the contents of a list box to the items a, b, and c, you would use the code in Example 2.

CONCLUSION

The source for the gopher client is available via anonymous ftp to st.cs.uiuc.edu, and will work on OS/2 and Windows, with IBM Smalltalk or VisualAge with the Communications Component. ♀

Patrick Mueller is a member of the IBM Smalltalk Distributed team at IBM Cary. He co-authored the HP/IBM submission to OMG for Smalltalk mappings to CORBA. Patrick can be reached by e-mail at pmuellr@vnet.ibm.com,

VisualWorks dialog development

Tim Howard

DEVELOPING CUSTOM DIALOGS can often be quite frustrating even for the well seasoned VisualWorks developer. Ideally, custom dialog development should be as easy and straightforward as that of non-modal window development. In this article I will cover some issues concerning VisualWorks dialogs. First, the basic role of dialogs in the application, as well as the current VisualWorks approach to custom dialog development, will be examined. Then I will discuss an abstract subclass of SimpleDialog, called ExtendedSimpleDialog, which makes dialog development much easier and straight forward. ExtendedSimpleDialog works in conjunction with ExtendedApplicationModel.[1]

Some related enhancements to ExtendedApplicationModel are also discussed. Full source code and examples are available from the archives at the University of Illinois (st.cs.uiuc.edu).

Here are some questions pertaining to dialog development that confront most VisualWorks developers. See if any of these are familiar to you.

- When should a dialog be used in an application?
- When is it necessary to build a custom dialog instead of using a stock dialog?
- Should custom dialog classes be subclasses of ApplicationModel or SimpleDialog?
- Why is it that an application model opened as a dialog cannot access its components at runtime?
- Why is it that an application model opened as a dialog will not execute preBuildWith: and postBuildWith: methods?
- Why is it that accept and cancel action methods are never executed?
- What should a custom dialog return—an edited object, nil, a boolean?
- Should a dialog ever edit an object directly, or just a copy of the object?
- How can a dialog be opened at a specific location?

The answers to these questions are provided in the remainder of this article.

PURPOSE OF DIALOGS

In VisualWorks, a dialog is a modal window. A modal window receives all user input until such time as it is closed. As long as a modal window is open, no other windows can receive user input. Dialogs are used to perform certain functions in the application and these functions can be placed into four broad categories.

1. Notify user of an error or display a simple message.
2. Acquire permission or simple information from user.
3. Allow user to instantiate and edit objects in the application.
4. Perform other application specific services.

The first two categories of functionality listed above are handled quite nicely by the stock dialogs provided by the Dialog class. The Dialog class is not meant to be instantiated, it is merely an access point for the stock dialogs. For the meager price of a single line of code, the stock dialogs provide the following functionality: display a message, ask for a yes/no confirmation, solicit simple information such as a string or a selection in a list, and provide simple file access. The stock dialogs should be used whenever possible however, many applications may have slightly varying requirements for displaying messages and acquiring simple information. For such cases, the developer can enhance the Dialog class, subclass the Dialog class, or build custom dialogs. While custom dialogs are optional when performing such simple dialog functions, they are absolutely essential when it comes to performing application specific services and editing objects.

VISUALWORKS CUSTOM DIALOG DEVELOPMENT

In VisualWorks, dialog behavior is implemented in the SimpleDialog class, a subclass of ApplicationModel. SimpleDialog differs from its superclass in the following ways.

- The initialize method instantiates three instance variables—accept, cancel, and close—to be ValueHolders with the initial value of false.
- The initialize method populates the builder's bindings with three aspects—#accept, #cancel, and #close. The associated value for each of these aspects is the corresponding instance variable.
- The window opened by a SimpleDialog is necessarily modal.
- The window constantly polls its model, the SimpleDialog, by sending it the value message that returns the value of the close instance variable, which is initially false. As soon as this value becomes true, the window closes.
- A SimpleDialog is often used to open the interface of another application model.
- Changes in accept or cancel instance variables will trigger a change message that sets the value of the close variable to true—thus closing the window.
- The statement that opens a dialog suspends the current thread of execution until the dialog is closed.
- The return value of a SimpleDialog is the value of the accept variable—true or false.

VisualWorks dialog development

Currently, there are two approaches to custom dialog development—and both have limitations. The first approach is to subclass SimpleDialog. One problem with this approach is that accept and cancel action methods will not be executed by action buttons bound to the #accept and #cancel aspects. This is because, the initialize method binds the #accept and #cancel aspects to their corresponding instance variables, eliminating the chance of binding them to corresponding action methods. Also, a subclass of SimpleDialog cannot open a nonmodal version of its interface—the window is necessarily modal. Furthermore, a subclass of SimpleDialog will not inherit functionality from an abstract subclass of ApplicationModel such as ExtendedApplicationModel. Therefore, you must duplicate such behavior in a complimentary abstract subclass of SimpleDialog or do without.

The second approach to custom dialog development is to subclass ApplicationModel and open it as a dialog. When such an application model is opened as a dialog, it does not assume the responsibility of building the interface with its builder but delegates this to an instance of SimpleDialog. For this discussion, I will refer to such an application model as the client application model and the instance of SimpleDialog as the surrogate application model. This approach suffers from the dormant accept and cancel methods that plagues the aforementioned SimpleDialog subclass approach. Another drawback of the ApplicationModel subclass approach is that the client application model's pre- and postbuild methods are never executed. Instead, it is the pre- and postbuild methods of surrogate application model that are executed. While this can be remedied with SimpleDialog's pre- and postbuild blocks, such a solution is awkward and cumbersome. By far the greatest drawback to subclassing ApplicationModel for custom dialogs is that the client application model cannot access its own interface during runtime. The reason for this is that it is the surrogate application model, and its builder, which builds the interface (browse ApplicationModel>>openDialogInterface: and SimpleDialog>>openFor:interface:). The client application model's builder instance variable references nil and therefore, the client application model has no means of accessing the interface.

INTRODUCING ExtendedSimpleDialog

The class ExtendedSimpleDialog has been developed to work in tandem with ExtendedApplicationModel to enhance VisualWorks custom dialog development. Custom dialog classes should be subclasses of ExtendedApplicationModel—do not bother with either ApplicationModel or SimpleDialog. By designing your dialogs as subclasses of ExtendedApplicationModel, you gain the following functionality.

- Prebuild and postbuild methods are executed without having to define pre- and postbuild blocks.
- All interface components can be accessed via the builder.
- Implementations for accept and cancel action methods can be bound to corresponding action buttons.
- All functionality in ExtendedApplicationModel is available for implementing the behavior of the dialog.
- The application model can open either a modal or non-modal version of its interface.

When a subclass of ExtendedApplicationModel (or an instance of such a class) is told to open its interface as a dialog, it instantiates an ExtendedSimpleDialog. The ExtendedSimpleDialog class re-implements the allButOpenFrom: aSpec method shown below:

```
allButOpenFrom: aSpec
    "Make sure the client references the builder and send pre and post
     build messages to the client."

    self builder source isNil
        ifTrue:        [builder source: self]
        ifFalse:       [builder source builder: builder].
    preBuildBlock == nil
        ifTrue:        [self builder source preBuildWith: builder]
        ifFalse:       [preBuildBlock value: self value: builder].
    builder add: aSpec window.
    builder add: aSpec component.
    self preOpen.
    postBuildBlock == nil
        ifTrue:        [self builder source postBuildWith: builder]
        ifFalse:       [postBuildBlock value: self value: builder]
```

This implementation does two things. First it makes sure that the client application model references the surrogate application model's builder. This gives the client application model access to the interface during runtime. Second, this method sends the pre- and postbuild messages to the client application model instead of the surrogate application model (provided the pre- and postbuild blocks are not defined). This implementation uses the knowledge that the builder's source is the client application model. The ExtendedSimpleDialog class also redefines the #accept and #cancel aspects in its own initialize method shown below.

```
initialize
    "Initialize such that the source can implement accept and cancel which are
     triggered whenever that button is pressed but prior to the actual closing."

    super initialze.
    self builder
        aspectAt:      #accept
        put:           [self builder source accept. self accept value: true].
    self builder
        aspectAt:      #cancel
        put:           [self builder source cancel. self cancel value: true]
```

This implementation allows the client application to define accept and cancel action methods that can be bound to action buttons. When the button is pressed, the action method is executed first, and then the corresponding ValueHolder is set to true that results in the closing of the window.

DIALOGS AS OBJECT EDITORS

Many custom dialogs are used to instantiate or edit an object of some type. I refer to such a dialog as an object editor. Object editors allow the user to edit an object and accept the changes or cancel to roll back to the previous state of the object. Since the object editors are dialogs, the user must conclude the object editing session one way or the other before moving on to anything else. This gives you the developer a great deal of control over the user's navigation of the application. A good way to approach object editing is the following three step process.

1. Copy the object to be edited.
2. Open a dialog object editor on the copy.
3. If the dialog returns nil, do nothing. If the dialog returns the copy, then replace original with copy.

Step 3 has our dialog returning either nil or the edited object and it was stated previously that dialogs always return a boolean (the value of the accept instance variable). Is this a contradiction? No, not really. It is the instance of our dialog that returns the boolean but we send the interface opening message to the class. As an example, lets consider a dialog that edits a ColorValue object. A ColorValue object has three attributes—red, green, and blue— one for each of the RGB values required to define a color. To edit a ColorValue object, we might create a class called ColorValueEditor as a subclass of ExtendedApplicationModel. On the class side, we would have an interface opening method such as this one:

```
edit: aColorValue
    "Open a dialog on aColorValue and return the edited
    ColorValue or nil."

    | colorValueEditor |
    ^(colorValueEditor := self new color: aColorValue) openAsDialog
        ifTrue:     [inst color]
        ifFalse:    [nil]
```

In this implementation, we first create an instance of our application model, colorValueEditor, and pass it the argument aColorValue. Then we open the application model as a dialog—allowing the user to edit the ColorValue object—and wait for its return value that is a boolean. If the dialog returns true, then the method returns the edited ColorValue object. If the dialog returns false, then the method returns nil. As an example of how this might be used, suppose we would like to allow a user to edit the background color of a window. To do this using ColorValueEditor, we might do write like the following.

```
insideColor := aWindow insideColor.
reply := ColorValueEditor edit: insideColor copy.
reply notNil ifTrue: [aWindow insideColor: reply]
```

In the code above, we first access the window's inside (or background) color that is a ColorValue object. Then we open a ColorValueEditor on a copy of this object. Remember, an object editor operates on a copy and not the orignal object. The reply variable will be either nil in which case we do nothing, or the edited ColorValue, in which case we replace the original.

INTERFACE OPENING PROTOCOL

ExtendedApplicationModel has been enhanced to provide more flexibility, functionality, and consistency in the interface opening protocol. This protocol is implemented on both the class and instance side and accounts for the following variations.

- Opening a specific interface.
- A reference to a parent application model.
- Opening at specific locations—both absolute and relative.
- Opening as a modal dialog.
- Guaranteeing that no more than one instance will ever be opened.

To account for all the permutations of interface opening methods, the following message naming convention has been adopted.

```
open{Interface: aSymbol}{From: aParentApp}{At: aLocation}
```

The bracketed elements are optional. The From: and At: options sometimes appear as key words in which cases they are not capitalized as follows:

```
openInterface: aSymbol at: aLocation
```

And sometimes they are just part of a key word and therefore appear capitalized as is shown here:

```
openAt: aLocation
```

The argument aLocation is used to determine where on the screen the window should be placed. If aLocation is a Point, then it indicates the intended opening origin of the window. If aLocation is a Rectangle, it specifies the origin and dimensions of the window. If aLocation is a Symbol, then it can be one of several values— #centerOfParent, #centeredOfScreen, or #centeredAroundCursor, for example. Each of these opens the window in the manner described by the Symbol.

A dialog is opened with an opening method of the form:

```
openAsDialog{Interface: aSymbol}{From: aParentApp}{At: aLocation}.
```

The single instance behavior is a guarantee that only one instance of an application model will ever be open at one time. For example, if the application model class SessionParameters is a subclass of ExtendedApplicationModel, then the following interface opening message will guarantee that only one such application will ever be open at any given time:

```
SessionParameters openSingleInstance
```

If the window is already open, then it is brought to the front of all windows and made the current active window. If the window is collapsed, then it is expanded. If the window is not currently open, then a new one is created and opened. A single instance is opened with an opening method of the form

```
openSingleInstance{Interface: aSymbol}{From: aParentApp}{At:
aLocation}.
```

Unlike the opening protocol discussed earlier, single instance opening protocol only applies to the class and does not apply to instances.

The following are just a few of the many permutations of interface opening protocol provided by ExtendedApplicationModel:

```
openAt: aLocation
openInterface: aSymbol from: aParentApp
openSingleInstanceInterface: aSymbol
openAsDialogAt: aLocation
openAsDialogInterface: aSymbol
openAsDialogInterface: aSymbol from: aParentApp at: aLocation
```

continued on page 32

Cleaning up after yourself

Alec Sharp & Dave Farmer

WHAT DO WE MEAN by cleaning up after yourself? Whenever you reference external resources, such as files, sockets, or UNIX processes, the garbage collector will not take care of closing or terminating these things. For example, you may have an object that opens a file. You can certainly close the file yourself when you are finished with it, but what happens if you simply stop referencing the object that opened the file? The garbage collector will clean up the object, but the operating system still has the file open.

Don't believe us? Do the following experiment:

```
count := 1.
[ file := 'foobar' asFilename writeStream.
Transcript cr; show: count printString.
count := count + 1.
ObjectMemory garbageCollect ] repeat.
```

When we ran this on a Windows system, we got an exception after opening 15 files (the number depends on what other files you have open and on the files setting in config.sys). If you are on a UNIX system, you'll get a lot more files open. On a UNIX system you can look at the files open by doing the UNIX command crash. Once in crash, type p then look for the oe20 process (st80 if you are using VisualWorks 1.0) and find its number. Then do u <number>, for example u 49. In Smalltalk you can see the open files (as long as you opened them in a standard way, such as sending the #writeStream message) by doing an inspect of ExternalStream classPool at: #openStreams. Then you can close individual files by inspecting the element of this collection and doing a self close.

Okay, but what does all this really mean? Shouldn't you close files after using them? Well, here's how our product works, and why we need to clean up after ourselves.

The Smalltalk part of our product consists of several Smalltalk processes, each one sitting in an infinite loop, waiting for input from either a socket or a shared queue. We have at least one socket permanently open. When we are developing and debugging, we keep a debug file permanently open. On top of that, some of the processes talk to robot tape libraries. Unfortunately, our device drivers block waiting for a response from the libraries, so we can't talk to them directly from Smalltalk because of performance issues. Instead, some of our Smalltalk processes fork and exec UNIX processes which they communicate with via pipes.

At this point, we have sockets and files permanently open, and UNIX processes sitting out there waiting to talk via pipes. We also

* *Note:* This article is based on work we have done at StorageTek using VisualWorks 2.0 from ParcPlace Systems.

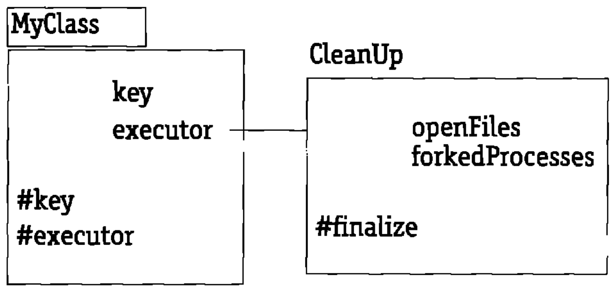


Figure 1.

have several Smalltalk processes waiting for things to do. As we are busy developing and debugging the code, perhaps we get a notifier window because something went wrong. After stepping through some of the code, we are at a point where we can't continue, so we terminate the operation. Alternatively, something might have got stuck in a loop, so we press ctrl-C to get control back.

If we didn't have some way to clean up after ourselves, we would now have stray UNIX processes, open files and sockets, and stray Smalltalk processes. In fact, this is exactly what we did have at first, to our frustration, so we had to come up with a way to prevent it.

FIRST WAY

The first scheme we came up with was to have an OrderedCollection called ThingsToCleanUp in our Pool Dictionary. Whenever we opened a file, created a process, etc., we recorded this event in ThingsToCleanUp. Here are a couple of examples (with code removed to show only the relevant portions):

```
startInputOutput
    ...
    ThingsToCleanUp add:
    'terminate LM Input' ->
    ([(LMInput newWithSocket: socket
        andQueue: self inputQueue) start]
            forkAt: StkConstants forkedProcessPriority).
    ...

initialize: aDeviceName
    OSErrorHolder errorSignal
handle: [:ex | ex restartDo: [^nil]]
do: [writeDevice := aDeviceName asFilename readWriteStream.
        ThingsToCleanUp add:
            'close writeDevice: ' , aDeviceName -> writeDevice].
```

The items we add to ThingsToCleanUp are associations. The key is a string that both specifies the operation which closes or terminates the thing, and gives us debug information we can log. The value is the forked process or opened file, etc. We thought this was pretty slick when we created it! To give a better idea of what is going on, here's the code that does the cleanup.

```
cleanUp
    ThingsToCleanUp do:
    [:thing | thing value notNil
        ifTrue:
            [Log debug: thing key.
            thing value perform:
                (thing key copyUpTo: Character space) asSymbol]].
        ThingsToCleanUp := nil.
        ...
```

If the key is the string "close debug file," thing key copyUpTo: Character space gives the string "close." If the value is the file itself, we get file perform: "close" asSymbol, which sends the #close message to the file.

Now, suppose we open a file and add it to ThingsToCleanUp and then later on want to close the file. We do something like:

```
close
    file notNil ifTrue:
        [ThingsToCleanUp removeAllSuchThat:
            [:element | element value = file].
        file close.
        file := nil].
```

All right, now we have in place a structure that allows us to record and perform cleanup operations. What triggers off the cleanup operations? The following method is how we start our Smalltalk product, and you'll see we've wrapped the entire product operation inside a valueNowOrOnUnwindDo block. This allows us to specify an operation that will take place when this method is being unwound, such as when we terminate a debug window or use ctrl-C and close the notifier window.

```
start
    [ self startInputOutput.    [ nextRequest := self inputQueue next.
        nextRequest queueYourselfUsing: self channelManager.
            ] repeat
        ] valueNowOrOnUnwindDo: [self cleanUp]
```

SECOND WAY

What is wrong with this technique? Well, there is one thing that can definitely cause problems, and another that has the potential to do so. First, ThingsToCleanUp is not threadsafe. If two processes were to do simultaneous operations, there is the possibility of problems (in fact we'd probably be okay because the Smalltalk processor is non-preemptive). Second, there is the possibility (although unlikely) that there are order dependencies. For example, if we closed a file before terminating a process that read the file, we might run into a problem.

So, phase two was to make ThingsToCleanUp threadsafe and to guarantee a certain amount of ordering of cleanup operations. To do this, we created a new class called CleanUp, which has several instance variables. It has a mutual exclusion semaphore and OrderedCollections for things such as files, sockets, UNIX processes, and Smalltalk processes. Our example here shows just files and Smalltalk processes.

The mutual exclusion semaphore lets us make access threadsafe, and the separate instance variables for the different objects we want to close, terminate, etc, allow us to make decisions about what order to do things. Here are examples of the new code showing how the object is initialized, and how you can add and remove files from the collections of objects that will need to be cleaned up.

```
initialize
    processes      := OrderedCollection new.
    files          := OrderedCollection new.
    accessProtect  := Semaphore forMutualExclusion.

addFile: anAssociation
    ^accessProtect critical: [files add: anAssociation]

removeFile: aFile
    ^accessProtect critical:
        [files removeAllSuchThat:
            [:association | association value = aFile]]
```

And here is the clean-up code. Our mutual exclusion semaphore protects everything, and within the protection, we terminate processes before closing files. As before, cleanup would be invoked in the valueNowOrOnUnwindDo: block.

```
cleanUp
    accessProtect critical:
    [ processes do:
        [:association | Log debug: 'Terminating process ',
            association key.
                association value terminate.
            processes := nil].

        files do:
        [:association | Log debug: 'Closing file ' ,
            association key.
            association value close.
                files := nil]
        ]
```

HANDLEREGISTRY WAY

One advantage of the way that we handle cleanup is that it happens quickly. A disadvantage is that we rely on a globally accessible object rather than handling things locally.

Another way we could have handled the problem, without the use of our global ThingsToCleanUp object, is to use the HandleRegistry class. A HandleRegistry is a very interesting object that allows us to set up a special relationship between our work object and a clean-up object. When our work object is garbage collected, the garbage collector informs the clean-up object about the garbage collection, and allows the clean-up object to do whatever cleanup we have coded (see Fig. 1).

Let's create two classes: MyClass and CleanUp. MyClass is where we open files and fork processes. CleanUp is where we close the files and terminate the processes when the MyClass object is garbage collected. In these examples we write to the Transcript so that you can try the examples yourself. In a real system we would not do that because the Transcript is not threadsafe. Try the code that follows—it's an interesting exercise in magic!

Here are the definition and the class initialization method for MyClass. Since class initialization happens at FileIn, we need to explicitly initialize MyClass. After you have typed in the class initialization method, select the self initialize text and execute it. We'll explain all the variables after the code:

```
Object subclass: #MyClass
    instanceVariableNames: 'key executor '
    classVariableNames: 'LastKey Registry AccessProtect'
    poolDictionaries: ''
    category: 'Examples'

initialize
    "self initialize"      AccessProtect := Semaphore forMutualExclusion.
    Registry := HandleRegistry new.
    LastKey := 0.
```

We have to register our MyClass object in a HandleRegistry object, so the first question is where to put the HandleRegistry. To avoid keeping a global object or using a pool dictionary, we put the HandleRegistry in a class variable in MyClass. When a MyClass object is created, during object initialization it registers itself in the HandleRegistry:

```
initialize
    key := self class newKey.
    executor := CleanUp new.
    self class register: self.
```

Each object that registers with the HandleRegistry has to be registered with a unique key, usually a SmallInteger. We don't pass in the key when we register an object; instead, the HandleRegistry asks the object for its key, so our MyClass object must return the key when sent the #key message. To generate unique keys for the specific HandleRegistry, we'll have the class keep track of the last key used. And because we may create MyClass objects from different processes, we'll create a mutual exclusion semaphore to make sure that access to the class methods is threadsafe.

Now we need some class methods to return the unique key, and to register our object in the HandleRegistry. The first one, #newKey, simply increments the key and returns the new value. The #register: method registers the object in the HandleRegistry and logs a message to the Transcript:

```
newKey
    AccessProtect critical:
        [LastKey := LastKey + 1.
        ^LastKey]

register: anObject
    AccessProtect critical:
        [Transcript cr; cr; show: Registering: ' , anObject printString , '
                                    with key: ' , anObject key printString.
        Registry register: anObject]
```

Now we need some instance methods for MyClass. We've already seen the initialize method, which gets a unique key for the object, stores a CleanUp object in the executor instance variable, then registers itself with the HandleRegistry. We also define accessors for the key and executor instance variables.

```
executor
    ^executor


key
    ^key
```

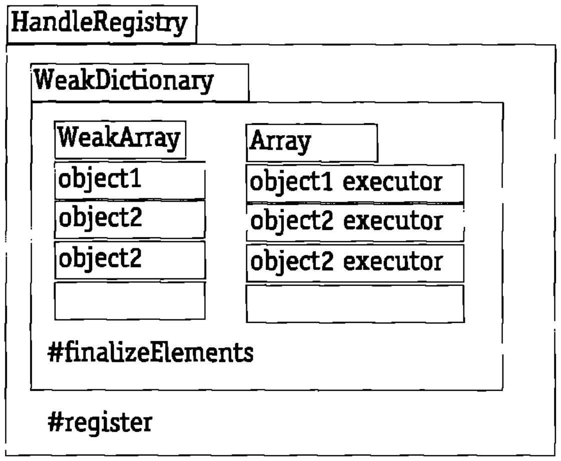


Figure 2.

Finally, we create a method called start, which adds files and processes to the CleanUp object. We'll take a very simple approach here and just add strings; in a production system we would open a file and add the actual file, not a string. Similarly, we would fork a process and add the actual process, not a string. What we have here is just to make the example a little simpler:

```
start
    self executor addFile: 'file1'.
    self executor addProcess: 'process1'.
    self executor addFile: 'file2'.
    self executor addProcess: 'process2'.
```

Well, that's about it for the main application class. Now we need to define our CleanUp class and its methods:

```
Object subclass: #CleanUp
    instanceVariableNames: 'openFiles forkedProcesses '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Examples'

new
    ^super new initialize.

initialize
    openFiles := OrderedCollection new.
    forkedProcesses := OrderedCollection new.

addFile: aFile
    Transcript cr; show: 'Executor: adding file ', aFile printString.
    openFiles add: aFile.

addProcess: aProcess
    Transcript cr; show: 'Executor: adding process ' ,
                    aProcess printString.
    forkedProcesses add: aProcess
```

To see all this work, simply type in MyClass new start.

To understand how the clean-up work is done, we need to take a look at three classes: HandleRegistry, WeakDictionary, and WeakArray. A HandleRegistry is a subclass of WeakDictionary which understands the #register: message. Most of the work done by a HandleRegistry is actually inherited, so most of the HandleRegistry methods we will talk about are implemented in WeakDictionary (see Fig. 2).

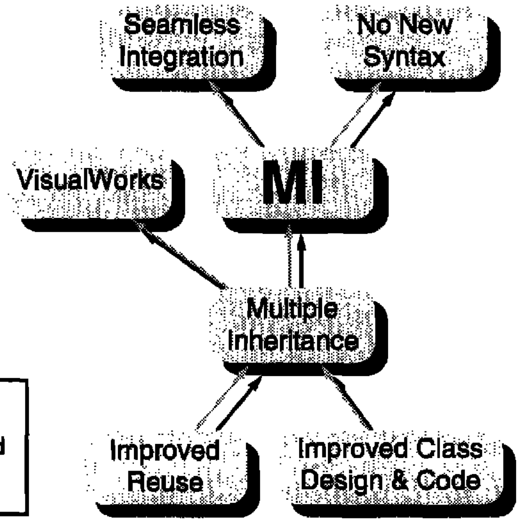
Are you maximizing your Smalltalk class reuse? Now you can with...

MI - Multiple Inheritance for Smalltalk

MI™ from ARS

- adds multiple inheritance to VisualWorks™ Smalltalk†
- provides seamless integration that requires no new syntax
- installs into existing images with a simple file-in
- is written completely in Smalltalk

*Leading methodologies (OMT, CRC, Booch, OOSE) advocate multiple inheritance to facilitate reuse. Smalltalk's lack of multiple inheritance support impedes the direct application of these methodologies and limits class reuse. MI is a valuable tool which enables developers to apply advanced design techniques that **maximize reuse**.*



Introductory Price: $195

To order MI or for more information on ARS's family of products and services, please call **1-800-260-2772** or e-mail **info@arscorp.com.**

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

3509 Haworth Drive • Suite 406 • Raleigh NC • 27609

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

†Implementations in VisualAge™ and Smalltalk/V™ are forthcoming

A HandleRegistry inherits two particularly interesting instance variables: valueArray (a WeakArray) and executors (an Array). When we register an object in the HandleRegistry, the HandleRegistry computes an array index from the object's key. It puts the registered object in valueArray and the object's executor in executors.

This is the reason we implemented the executor method in MyClass—so that the HandleRegistry can ask the object for its executor. The default executor method is implemented in Object and returns a shallow copy of self, so even if you don't implement the executor method, the HandleRegistry still gets something to put in the executors array.

Where things get interesting is in the WeakArray. Objects referenced by a WeakArray are referenced *weakly*. This means that they can be garbage collected if the WeakArray is the only thing that references them. (Actually, they can be referenced by multiple WeakArrays and still be garbage collected.) This is in contrast to normal references between objects, which are *strong* references. For example, an object in an OrderedCollection will not be garbage collected until the OrderedCollection no longer references it.

Furthermore, when the garbage collector collects an object that is referenced by a WeakArray, it sends the WeakArray a #changed message. The WeakArray informs all its dependents by sending them an #update:with:from: message.

When our HandleRegistry creates the WeakArray in the valueArray instance variable, it immediately registers itself as a dependent of the WeakArray. When the HandleRegistry receives the #update:with:from: message from its WeakArray, it sends itself a #finalizeElements message.

The finalizeElements method finds all the objects in valueArray which have expired and sends a #finalize message to the executor of each one. The default finalize method, implemented by Object, does nothing. However, since we want to close files and terminate processes, we override the finalize method in our CleanUp class.

```
finalize
    forkedProcesses do: [:process |
        Transcript cr; show: 'Executor: terminating process ',
                        process printString]
    openFiles do: [:file |
        Transcript cr; show: 'Executor: closing file ',
                        file printString].
```

In our example, we first terminate (or pretend to) the forked processes, then we close (or pretend to) the open files. We do it in this order just to make sure that no processes try to access closed files (although they shouldn't because we only get here if the object that opened the files is being garbage collected).

CONCLUSION

In conclusion, external resources usually need special processing to make sure they are released. We have presented some different techniques to do this, from the simple approach using a Pool Dictionary object which tracks the external resources, to the more interesting and magical approach of using a HandleRegistry. ♀

Alec Sharp is an Advisory Software Engineer at StorageTek. He is the author of Software Quality and Productivity, published by Van Nostrand Reinhold. He can be reached at alec_sharp@stortek.com. Dave Farmer is a Senior Software Engineer at StorageTek. He can be reached at david_farmer@stortek.com. They both work on the UNIX Storage Server software, which manages connections to networked hosts and drives the StorageTek family of robotic tape libraries.

Suggestions for a successful user interface

Amy S. Gause

THE MAJOR DECISIONS in creating a user interface for a network management system that monitors a switched digital video network and is the access to video services is examined here. The complete system was rewritten using Smalltalk (it had all been primarily in C). Several problems that were encountered are addressed and their solutions are offered. The design is based on the Model View Controller (MVC) paradigm, where the model is the data holder and notifies dependents of data changes, the view is the screen display and the controller controls the view and accesses the model.

ANALYSIS AND DESIGN PHASE

The first step taken in creating a user interface for this project was initially prototyping the basic screens and investigating how they would work together. After completing three different prototypes, it became clear what would work and what would not. The first prototype just ported the screens from the previous user interface (ASCII based), just to get familiar with VisualWorks. The second prototype did not include any of the screens in the first prototype. These screens were completely new; again, just to see what VisualWorks could do and how our reorganization of the screens would mesh. New ideas were tried to see if they were doable. A third prototype was done to investigate a completely different idea of presenting data. These prototypes helped give us a basic foundation to make more informed analysis and design decisions.

Second, analysis began with itemizing the information and functions to provide for the user. By grouping these items, decisions were reached on what types of screens to implement. Several methodologies were examined, but none seemed to address the special needs of user interfaces. Booch had this to say about user interface A & D: "The design of an effective user interface is still much more of an art than a science. For this domain, the application of prototyping is absolutely essential. Feedback must be gathered early and often from end users...The generation of scenarios is highly effective in driving the analysis of the user interface." It is true that feedback from users is extremely important, but something must be designed/implemented for which the users can give feedback.

The third step taken was using the CRC technique to further the analysis phase, but augmented to suit the specific needs of a user interface. For example, CRC cards are normally used for listing Class/Responsibility/Collaborator; this was changed to Window (which also turned out to be the class), Navigation (where could the user go from this screen), Actions (what the user can do on this screen) and Data (what data is displayed on this screen). Each card represented a screen.*

It became clear how to derive the superclasses of these windows(classes) after several cards were done and commonalities could be seen. The superclass names were added to the cards as more of a reminder to factor out the common functionality. Cards were also made for the controller classes (controllers for the windows) and windows not implemented in the prototypes. These cards more closely followed the CRC methodology by listing the responsibilities and collaborators.

The cards also helped to make the logical groupings of the windows more apparent. Laying the cards on a table and organizing them in different arrangements helped to decide how to group the windows and how the navigational hierarchy could be set. ENVY/Developer also made these groupings easier by allowing the windows to be grouped into SubApplications according to behavior and inheritance.

Another round of prototyping the screens then began. Using the augmented CRC cards, screens were implemented going by their data, navigation and actions listed. After some initial screens were created, they were "hooked" together navigationally. This prototype (called a mock-up) helped determine how the screens could logically work together.

SCENARIOS

Thirty (an arbitrary number) typical functions that this UI should provide served as the basis of these scenarios. These functions were attained from the user requirements documentation. The prototyped screens were used to follow the scenarios to see how these functions would be accomplished by the user. When deficiencies were noticed, they were fixed in the prototype. The prototype was being iterated upon and eventually grew into the completed UI. One of the problems encountered was once the prototype was stable and provided on-the-surface functionality, an approach on handling demos should have been considered (see the following).

USER FEEDBACK

After a good collection of navigational screens (one view being able to launch another view, and so on) was attained, user feedback was requested. The users worked with the screens (although they had no real functionality at this point) and noted what they liked and disliked. We watched and noted the users confusion, problems, and comments for about two days. By doing this we gained valuable information from our users, and the users were pleased because we were interested in providing them with a tool they

* The three prototypes were done before the cards (see Fig. 1).

wanted and liked. We enhanced the prototype with the users suggestions; in most cases we found alternate ways to provide them with the functionality they desired. It must be understood that they may not know the ramifications of their suggestions. It is the UI designers responsibility to provide the user with the functionality they want without deteriorating the interface or the design.

DEVELOPMENT PHASE

The next step was to provide the screens with a mechanism of obtaining "real" data. All that these screens contained until this point were labels, fields, buttons, empty tables, empty lists, etc. An initial data model was stubbed-up to provide data to the views. The data model is a logical view of the network and exists to provide the user interface with a view of the network, but from a users perspective. It has connections that do not exist in the network, but provided navigation among screens. It provides the views with the data. The views each had their own data "controller" to convert the data from the data model to exactly the way it was to be displayed (format). For example, some data may be displayed as a string on one screen and as an integer on another depending on the widget. At this point, more enhanced functionality was provided, such as: disabling and enabling certain areas of the screens depending on which items were selected and providing different types of menus depending on which navigational path the user happened to choose, etc.

PROBLEMS AND SOLUTIONS

(Using ParcPlace Smalltalk with VisualWorks) During the development of this user interface, many problems were encountered. The significant ones along with the solutions implemented are listed below.

Context Menus

The yellow and blue button menus (on all the screens) provided by the development environment still were available to the user at runtime—which could be potentially dangerous to the running image if the user happened to get curious. For the yellow button, an extension was made to the ApplicationModel that provided a class method that returned nothing but an empty PopUpMenu. On each screen that contains fields, this method answered the menu of that field. For the blue button, a subclass of ApplicationStandardSystemController was created to override controlActivity as follows:

```
controlActivity
    ^self controlToNextLevel
```

Extending the Application Model

Some screens had large amounts of data, meaning many of components (labels and fields). When the control layer (the data controller and the view controller) was put in, there was a lot of `not very readable' code when it comes to disabling/enabling components, grabbing the value of a Label off the view, etc. When disabling/enabling components, the following code was implemented:

```
aBuilder := self builder.
(aBuilder componentAt: #name) disable.
(aBuilder componentAt: #socialSecurityNumber) disable.
(aBuilder compenentAt: #salary) disable.
```

The article "Extending the Application Model"[1] addresses a solution to this problem by extending the ApplicationModel with a method called "disable" that could look something like:

```
disable: anArrayOfComponents
    anArrayOfComponents do: [ :each |
    (self builder componentAt: each) disable ].
```

The same could be done for enable. A method could be added for returning the value of a Label. Our initial try at this looked like:

```
labelFor: aSymbol
    ((self builder componentAt: aSymbol) component component label
  text string) asNumber.
```

Even in Smalltalk, GUI code can look complicated. Many extensions could be made to the ApplicationModel depending on what is necessary. If they are diversified, each type of screen might have an ApplicationModel of its own. Extensions can be developed for such behavior as invisibility, taking keyboard focus, or any other view functionality that is implemented across several screens.

Updating Tables

Tables are not as easy to develop as the examples in the VisualWorks User's Guide, especially if dynamic updates are necessary. The tables on our UI were to be read-only with dynamic updates. When a window with a table on it is installed and methods for this window are defined using the default initialization, a method is created on the instance side of the class under the aspects protocol. The example provided in the user's guide uses a TwoDList. TwoDLists will not let dynamic updates happen to the table. The solution was to edit the method that was created in the aspects protocol to use laissez-faire initialization and return a TableInterface whose selection InTable is set to a TableAdaptor on a List. So it looked something like:

```
exampleTable
^exampleTable == nil
    ifTrue: [ exampleTable := TableInterface new
    selectionInTable: (SelectionInTable with: TableAdaptor on: List new
    adaptors: (RowAdaptor adaptForIndexes: #( 1 2 ))));
    columnWidths: #(100 100);
    columnLabelsArray: #( `Name' `Address' );
    yourself.
    "For dynamic updates add self as a dependent of the
selectionIndexHolder" ]
    ifFalse: [ exampleTable ].
```

For dynamic updates, in the update: method it is necessary to provide a way to re-read the data into the table—a call to the same method that read the data into the table in the first place would do.

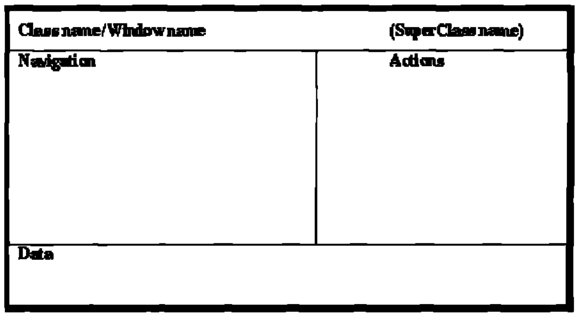


Figure 1.

Row Labels for Tables

Row label widths did not dynamically resize according to the longest label and were sometimes chopped on the left for long labels. Several screens displayed tables that had row labels. If the row label width was set to 100 (pixel width) then a long string as a row label would get chopped on the left side. If all the row labels were short strings then the space at the left was too large. By dynamically calculating the pixelwidth of the longest row label and setting the row label size to that value, the table resizes automatically and no row labels get chopped. This seemed a simple enough solution, but the hard part was how to calculate that pixelwidth. If the graphicsContext of the table is gotten, its font appears to be #fixed (we were using the #default font). This throws off the pixelwidth calculations if the widthOfString: method is used. As of present, changing the font of the row labels to #fixed is the implemented solution. Another solution is to convert the row labels to Composed Text and use the width to recalculate the row labels width.

Demonstrations

Marketing and Documentation departments requested demonstrations for the current state of the system during all phases of the project. Demonstrations should be considered during the design phase. One of our problems was that we were not thinking about providing demos, so when a demo was needed, the process was not an easy one and the demo had functions in it that would throw walkbacks. Depending on how classes are set up in a system there will be different approaches on how to handle this problem. Our system is arranged in such a way that utilizing subclasses could be helpful. For example, there is a class for a data entry window. A subclass of this class, DemoDataEntryWindow, could have the same windowSpec, same functionality, but override the necessary behavior that would not be appropriate for a demo. Replace functionality with descriptive Dialogs.

Displaying Data

Every window opening should default with the most important data for the user using as little screen real estate as possible, but other data needed to be on the window too. Several suggestions on how to handle this were given: have two windows, but if the parent window closes so should the child; have one window with a toggle switch to expand the window if set and shrink the window if not set. The latter was implemented. The window was created with important data on top and less important data on bottom. A check box was added for expanding and shrinking the window.

The minimum size of the window was set such that the bottom data could not be seen. This is done by choosing the window->bounds->fixed size options on the <operate> menu when editing the canvas. When the window is expanded, it was desired that some buttons should stay with the bottom of the window. Instead of moving a component to another part of the window, the invisibility property was used.

When the check box in Figure 2 was checked then the top divider and button were made invisible and the screen was expanded to its complete size. When the check box was not checked the top divider and button were made visible and the screen returned to the minimum size. The method that handled the expanding and shrinking is dependent on the value of the check box.†

```
initialize
self toggle value: false.
self toggle onChangeSend: #resizeView to: self.
resizeView
self showDetails value
ifTrue: [ (aBuilder componentAt: #topButton) beInvisible.
    (aBuilder componentAt: #topDivider) beInvisible.
    winRectangle := aBuilder window displayBox.
    (aBuilder window) moveTo: (currentRectangle origin)
    resize: ((winRectangle extent) + (0@400)) ]
    ifFalse: [(aBuilder componentAt: #topButton) beVisible.
    (aBuilder componentAt: #topDivider) beVisible.
    winRectangle := aBuilder window displayBox.
    (aBuilder window) moveTo: (currentRectangle origin)
    resize: ((winRectangle extent) - (0@400)) ]
```

The 400 depends on how much expansion is necessary to reveal the lower data. Since this is a hardcoded number, whoever edits the canvas and changes the length of the window will have to be aware that this value may very well need changing, too. A better way would be to derive the value instead of hardcoding it. Consideration of the window manager is necessary when dealing resizing capabilities.

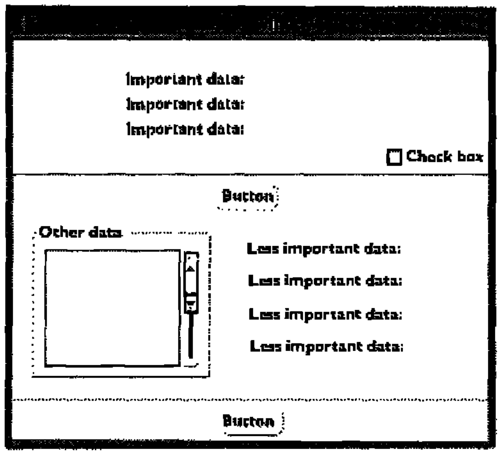


Figure 2.

CONCLUSION

Prototypes are extremely important, especially if the developers are new to Smalltalk. Prototypes help developers gain a basis for making important design decisions. Scenarios help developers relate to users. They should be understandable for both developers and users. Scenarios are an excellent test of the user interface and help to locate potential usage problems early. ♀

Reference

1. Howard, T., and B. Kohl, Extending the application model, SMALLTALK REPORT, 3(7):1–7, May 1994

Amy S. Gause is a software engineer at BroadBand Technologies, Inc. and has been working with ParcPlace Smalltalk and VisualWorks 1.0 for a year. She can be reached at asg@bbt.com.

† Code enhancements as described earlier are directly applicable here.

Storing objects into files in VisualAge

Wayne Beaton

VISUAL PROGRAMMING IS the way of the future! A bold statement that certain players in the computer industry hope to make come true. VisualAge offers a step in this direction. Currently, VisualAge lacks the rich set of parts that will make it an overwhelming success. With some third party involvement, and some ingenuity, we can solve this problem by providing VisualAge parts to visually solve any problem. Okay, not *any* problem, but we can come very close.

Before I go too far, let me get this out into the open. I like VisualAge. It helps me to do my job faster and easier. VisualAge just lacks some features that I would like to have. When it is decided that a new part is required, a decision must be made on how to implement it in a way that makes it the most useful. I've heard it said that "in general, general solutions don't work." I believe that this means if you try to anticipate what the next guy wants your part to do, you're likely to be wrong.

The first part that I thought that I might like to have that was missing, was a part that could provide file access. All I really want to do is write a single object to a file and then read it back. From this was born the ObjectLoaderAndDumper.

THE OBJECT SWAPPER CLASSES

IBM Smalltalk comes with two classes that provide file access. The class ObjectDumper can be used to write objects to a file; the class ObjectLoader will read objects from a file. The interfaces for these classes are pretty straight forward. To write a single object to a file, the following code can be used:

```
ObjectDumper new
    unload: anObject
    intoFile: `c:\junk.dat'
```

Retrieving the object back from the file is another simple matter.

```
anObject := ObjectLoader new
    loadFromFile: `c:\junk.dat'
```

Using these classes, we can easily build a reusable nonvisual part to use in VisualAge. How would such a part work? The ObjectLoaderAndDumper has two attributes, fileName and object. Further, it has two actions, load and dump (see Fig. 1).

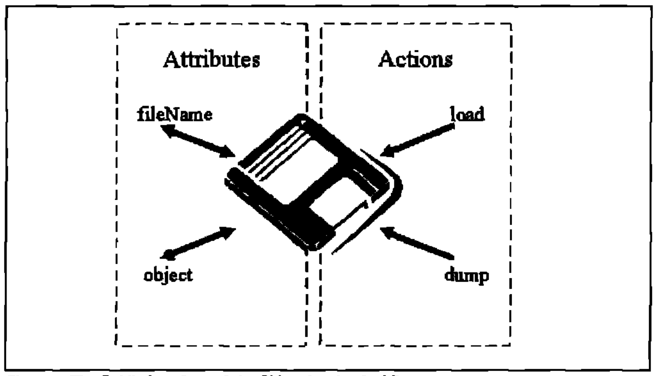


Figure 1. The Public Interface of the Object Loader And Dumper

To load an object from a file, the fileName attribute is set to the full name and path of the file to load, and the action load is invoked. The result of the load action puts the contents of the file into the object attribute. To dump an object to a file, the fileName attribute is set to the name of the file, and the object attribute is set to any object. The action dump is invoked, and the contents of the object attribute is written into the file.

BUILDING THE OBJECTLOADERANDDUMPER

The ObjectLoaderAndDumper was created as a nonvisual part. Both attributes use the default attribute settings, with the fileName attribute of type String and the object attribute of type Object. The load action invokes the script load and the dump action invokes the script dump. The generated scripts for the attributes will not be included here, the scripts for load and dump are as follows.

```
ObjectLoaderAndDumper instance methods:
load
    "Sets the receiver's `object' attribute to
    the contents of the file named by the
    receiver's `fileName' attribute."
    self object:
        (ObjectLoader new
            loadFromFile: self fileName)

dump
    "Writes the contents of the receiver's `object'
    attribute to the file named by the receiver's
    `fileName' attribute."
    ObjectDumper new
        unload: self object
        intoFile: self fileName
```

This code alone makes the ObjectLoaderAndDumper easy to use visually. However, a nice extension might be to provide file prompters to allow the user to select a file name. Again, it would be nice to do this all visually.

FILE SELECTION

The class CwFileSelectionPrompter provides a connection to an operating system specific file browser. We can use this class to get file information from the user. Actually, there's a better class to use, one that itself employs the CwFileSelectionPrompter. The class EtFileNamePrompter provides a little more behavior for file browsing. This class has two class methods. Using the class method #promptForFileName: default: shouldExist: at: we can provide a message to the user, suggest a default, stipulate whether or not the file specified by the user should exist, and suggest the location (in screen coordinates) where to open the file browser.

The message to the user is the message that is displayed at the top of the file browser. The default is the name of a file, which can include wild-card characters, to initially provide to the user.

I like VisualAge. It helps me to do my job faster and easier.

Stipulation of whether or not the file exists allows us to force the user to select a file that already exists. This can be particularly useful for specifying a file to load. If the user specifies a file name that does not exist, they are informed and motivated to provide a different name. The last parameter, the screen coordinate to open the file browser at, does not seem to have any affect in the code.

This message will answer the name of the file specified by the user including the full path, or nil if the Cancel button is clicked.

```
| fileName |
fileName := EtFileNamePrompter
    promptForFileName: `Select a file to open:'
    default: `*.dat'
    shouldExist: true
    at: 0@0.
fileName isNil ifTrue: ["the user cancelled"]
```

Using this new knowledge, the ObjectLoaderAndDumper can be extended to include actions called promptForFileNameAndLoad and promptForFileNameAndDump. The corresponding methods follow:

ObjectLoaderAndDumper instance methods:

```
promptForFileNameAndLoad
    "Prompts the user for the name of the file
     to load and then loads it."
    | newFileName |
    newFileName := EtFileNamePrompter
        promptForFileName: `Load from file:'
        default: self defaultFileName
        shouldExist: true
        at: 0@0.
    newFileName isNil ifTrue: [^self].
    self
        fileName: newFileName;
        load
promptForFileNameAndDump
    "Prompts the user for the name of the file
     to save and then saves the contents of the
     receiver's `object' attribute to that file."
    | newFileName |
    newFileName := EtFileNamePrompter
        promptForFileName: `Save to file:'
        default: self defaultFileName
        shouldExist: false
        at: 0@0.
    newFileName isNil ifTrue: [^self].
    self
        fileName: newFileName;
        dump
```

ObjectLoaderAndDumper private methods:

```
defaultFileName
    "Private - Answers the name to use by
     default when prompting the user."
    ^self fileName isNil
        ifTrue: [`*.*']
        ifFalse: [self fileName]
```

READ-ONLY ATTRIBUTES

Read-only attributes were added to provide the ability to disable buttons or menu items when loading or dumping are not possible. Read-only attributes, when created in the public interface editor, have only the get selector, changed event symbol and type fields specified. Further, instance variables need not be created for read-only attributes.

The read-only attributes isLoadEnabled and isDumpEnabled both have the type Boolean.

ObjectLoaderAndDumper instance methods:

```
isLoadEnabled
    "Answers whether or not it is possible to
     load. The receiver can load only if the
     `fileName' attribute is specified."
    ^self fileName isString
isDumpEnabled
    "Answers whether or not it is possible to
     dump. The receiver can dump only if the
     `fileName' attribute is specified and the
     `object' attribute is not nil."
    ^self fileName isString
        and: [self object notNil]
```

To make these read-only attributes work as expected, we must signal when they change. Clearly, isLoadEnabled changes when the fileName attribute changes; isDumpEnabled changes when either the fileName attribute or the object attribute changes. Extensions can be made to the set methods for these attributes.

ObjectLoaderAndDumper instance methods:

```
fileName: aString
    "Sets the `fileName' attribute to aString."
    fileName := aString.
    self
        signalEvent: #fileName with: aString;
        signalEvent: #isLoadEnabled
            with: self isLoadEnabled;
        signalEvent: #isDumpEnabled
            with: self isDumpEnabled
object: anObject
    "Sets the `object' attribute to anObject."
    object := anObject.
    self
        signalEvent: #object with: anObject;
        signalEvent: #isDumpEnabled
            with: self isDumpEnabled
```

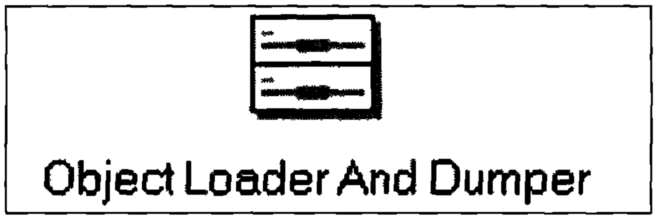


Figure 2. The object loader and dumper in the composition editor.

PRETTY ICONS

By default, new parts created by developers represent themselves in the Composition Editor as simple puzzle pieces. It might be better to use a different icon to distinguish the ObjectLoaderAndDumper from other parts. To do this, we can specify a class method #abtInstanceGraphicsDescriptor.

```
ObjectLoaderAndDumper class methods:
abtInstanceGraphicsDescriptor
    "Answers the descriptor for the icon to
     display in the Composition Editor."
    ^AbtIconDescriptor new
        moduleName: `ABTICONS';
        id: 288
```

This method answers an instance of the class AbtIconDescriptor, which requires the name of a module and an id within that module. In the OS/2 world, modules are DLLs containing icons. Fortunately, VisualAge comes with a DLL filled with icons that we can use (an appendix in the USER'S GUIDE lists them all). Icon id 288 holds a picture of two disk drives stacked on one-another (see Fig. 2). I thought this icon was appropriate enough.

A SIMPLE TEXT EDITOR

With the ObjectLoaderAndDumper specified it is an easy matter to create a simple text editor without scripting.

The required parts were assembled using the Composition Editor. A multiple line text was added to a window with appropriate sizing information. Several menus were also added, along with an Object Loader And Dumper. Figure 3 shows the entire assembly, including all required connections.

The connection labeled "1" in Figure 3 connects the fileName

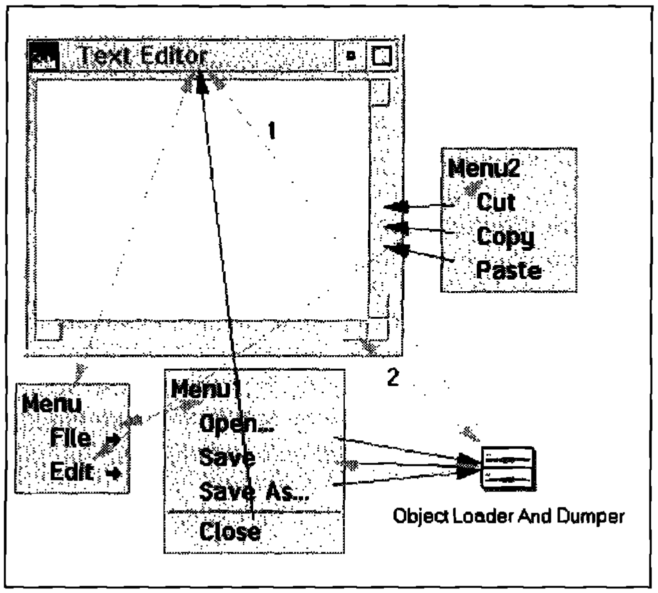


Figure 3. The simple text editor.

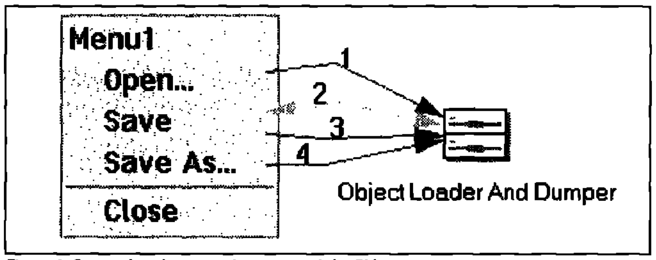


Figure 4. Connections between the menu and the Object

attribute of the Object Loader And Dumper by a one-way attribute-to-attribute connection to the "title" attribute of the window. This has the effect of changing the title of the window to the name of the file specified by the user in any open, or save operation. The connection labeled "2" connects the object attribute of the Object Loader And Dumper to the object attribute of the multiple line text by a bi-directional attribute-to-attribute connection. For this connection to work, the Notify change on each keystroke attribute of the multiple line text must be set to true.

Figure 4 shows a close up of the connections between the File menu and the Object Loader And Dumper. The connection labeled "1" hooks the clicked event of the Open... menu item to the action promptForFileNameAndLoad. For the "Save" menu item, the connection labeled "2" connects the enabled attribute to the isDumpEnabled attribute; the connection labeled "3" connects the clicked event to the dump action. The connection labeled "4" connects the clicked action of the Save As... menu item to the action promptForFileNameAndDump.

"If you try to anticipate what the next guy wants your part to do, you're likely wrong."

From these connections, the File menu springs to life. The "Save" item is disabled until a valid file name has been specified, either by opening a file, or by saving one. Once enabled, the Save item will save to the existing file.

CONCLUSION

VisualAge is missing some obvious features that would make life easier for developers. Unfortunately, anticipating the needs of thousands or millions of developers is a daunting task, and providing a general solution that everyone can live with is nearly impossible. However, creating our own reusable parts is an easy matter.

The ObjectLoaderAndDumper is an example of a reusable nonvisual part which will read a single object from a file or write a single object to a file. Large applications may require a more sophisticated part to satisfy disk file access. The ObjectLoaderAndDumper is a starting point which demonstrates the amazing potential of VisualAge.

Wayne Beaton is a senior member of the technical staff and part-time instructor at the Object People Inc., Ottawa, ON, Canada. He finds just about any kind of problem "darned interesting." He can be reached e-mail at wayne@objectPeople.on.ca.

Alan Knight

Instance initialization

ONE OF THE goals of the object-oriented approach is not to have to worry about the internal representations of objects. One aspect of this is that clients should not have to care about initializing the objects they use, and that newly created objects can be expected to be in a usable state. There are a number of ways of accomplishing this in Smalltalk, and (naturally) a wide variety of opinions on the relative merits of each.

OVERRIDING NEW

The most common initialization technique is to override the class method new to be

```
new ^super new initialize.
```

With this override, any object created using new is guaranteed to be initialized before any other messages are sent to it. It does have a drawback, however. We have to implement this method in quite a few places, and be quite careful about which places. The default implementation of new doesn't call initialize, so we must provide an implementation in our classes, but only if they inherit directly from Object. If we provide it in other classes, then their initialization code will end up running more than once.

To see how this happens consider the following class hierarchy.

```
Object
    AbstractClass
        ConcreteClass
```

We override new in both AbstractClass and ConcreteClass, and provide initialize methods. The expression ConcreteClass new will result in the following sequence of calls:

```
Concrete>>new
Abstract>>new (called via super from ConcreteClass)
Behavior>>new (called via super from Abstract)
Concrete>>initialize (called from Abstract>>new)
Abstract>>initialize (called via super from Concrete)
Concrete>>initialize (called from Concrete>>new)
Abstract>>initialize (called via super from Concrete)
```

This multiple initialization is inefficient and can cause problems if it's not safe to run the initialize routine multiple times. It's important that new only be overriden once, in immediate subclasses of Object. It's also important that we call super initialize in our initialize methods unless we inherit directly from Object, in which case we must not call it.

This is a bad thing, since we must be aware of a class's position in the inheritance hierarchy and modify code if the inheritance hierarchy changes. It's not that difficult, but it's an unnecessary and tricky detail that detracts from something that tries to be extremely simple.

There's a very simple way to fix this. The default new method (in Behavior) should be:

```
^self basicNew initialize.
```

and there should be an Object>>initialize that does nothing. This would allow the elimination of almost all overridden new methods and make the usage of initialize much more consistent.

There are, however, a couple of problems. The first is that not all objects require initialization, and some of those that don't are important system objects. We don't really want to add an extra message send to the cost of every Point, Rectangle, or Float creation. This is easily overcome. Instead of overriding new in most user classes, override it in those system classes where it's important.

"In the absence of a proper specification, an initialize method provides an easy way to see at a glance the expected types of the instance variables."

Most of these objects are already created with special class messages, and these can be changed to call basicNew instead of new. The name basicNew even makes sense as a basic creation operation that does nothing else, not even the normal initialization. This is simpler and more consistent than current usage, and it takes the burden of worrying about the problem off of "normal" users.

The second objection is backward compatibility. It would have been nice if Smalltalk was originally designed with this initialization scheme, but it wasn't. If a Smalltalk vendor were to make this change today, it would break almost every class written for the old initialization scheme. In fact, it would introduce exactly the multiple initialization problem it's designed to avoid. If only one vendor introduced this convention, it would add an additional incompatibility with all the other dialects. Because of this, I doubt such a change will be adopted unless it's mandated by a higher body like the ANSI standardization committee.

This suggestion isn't original, but has been suggested by quite a number of people. I believe I first heard it suggested by **Bobby Woolf (woolf@acm.org).**

OTHER INITIALIZATION ISSUES

There's another potential problem with automatic initialization. Even if the initialize only runs once, it can still do unnecessary work. It's rare that I actually want an instance of something

Alan Knight performs a wide variety of Smalltalk-related work with the ObjectPeople. He can be reached at 613.225.8812 or by e-mail at knight@acm.org.

with the default values. Generally I'm going to create an object, initialize it to default values, then immediately overwrite those with the correct values. This wastes at least one memory allocation for each initialized variable, and probably more if those objects have their own initialization code.

This isn't usually seen as very significant, for several reasons. First of all, there isn't the same possibility of actual error as with multiple initialization. While initialize methods often make assumptions about the state of the object, setting values after initialization will normally use public accessors, which are much safer. For example, an initialize method might be written as

```
initialize
    tempFile := (File named: self defaultFileName) open.
```

If this is run repeatedly because the original file variable will get overwritten and the first file will never be closed. On the other hand, public access methods usually assume they may be run repeatedly and take appropriate precautions.

```
tempFileName: aFileName
    (tempFile notNil and: [tempFile isOpen])
        ifTrue: [tempFile close].
    tempFile := (File named: aFileName) open.
```

Overwriting initialized variables is not only safe, but it doesn't usually cost much. Object creation is extremely cheap in Smalltalk, and if the default values are simple, the cost just isn't worth worrying about under normal circumstances.

If you do want to worry about it, it's possible to work around this problem using class creation methods. Lots of objects aren't created with new, but with custom class messages which either require specification of the important variables or provide defaults. Often there will be simpler versions in which most of the arguments default to simple values and more complex messages where all the parameters must be fully specified. A typical example might look something like:

```
new
    ^self foo: self defaultFoo.
foo: aFoo
    ^self foo: aFoo bar: self defaultBar
foo: aFoo bar: aBar
    ^self basicNew foo: aFoo; bar: aBar.
```

If all instance creation is done through these class messages, then all the variables requiring initialization should be initialized exactly once, with no wasted effort.

Of course nothing's perfect. This method has the disadvantage of spreading code for default values even more than lazy initialization (see below). It also suffers from Smalltalk's strict requirements on message form. To define something like this, I really want to say that there is one creation method with N arguments and that some or all of them may be omitted, in which case they should use the default value. In Smalltalk I have to explicitly define 2^N different messages if all the combinations are possible. The usual compromise is that only a few different messages are defined, representing what the developers feel are the most common cases.

LAZY INITIALIZATION

A more general way of overcoming these difficulties is to use lazy initialization. With this technique, variables are not initialized when an object is first created, but on first access to the variable. This usually involves writing get methods as:

```
foo
    foo == nil
        ifTrue: [foo := #defaultFoo]
        ifFalse: [^foo].
```

There's no danger of a variable being initialized twice, and if we set the variable any time before it's accessed there's no duplicated work.

This technique has its own disadvantages. While it eliminates the possibility of unnecessary work in initialization, it introduces some overhead on each variable access. There are two sources, the nil test and the inability of the compiler to optimize the access method as is normally done for pure get/set methods. This inefficiency is still negligible for most cases, and if there are significant numbers of variables that aren't accessed at all, the savings from not initializing at all can easily outweigh this overhead.

“*The default new method should be* **^self basicNew** *initialize.*”

A stylistic objection is that lazy initialization pretty much requires all variable access to be through message sends. That isn't such a bad thing, and in fact lots of people advocate it as good style. The big disadvantage is that you must define access methods for everything, even private variables. Since no Smalltalk currently supports enforced private methods these methods must be public. Some of the issues involved with this style of coding are discussed in Kent Beck's article “To accessor or not to accessor?” (THE SMALLTALK REPORT, 2(8):8).

A number of people I spoke to didn't like lazy initialization because they believed it was important to keep all the initialization code in one place. None of them had a convincing explanation why it was important, but I have a theory. I think it has to do with specification rather than initialization.

Since Smalltalk is dynamically typed, there's almost no information in a class declaration. You know the superclass, and the number and names of the instance variables. There's no information on the expected type of each variable, and even if the names are good they don't necessarily indicate types. Information about possible types is one of the valuable things class comments provide. Unfortunately an awful lot of code is written without class comments, and those that exist aren't necessarily accurate. In the absence of a proper specification, an initialize method provides an easy way to see at a glance the expected types of the instance variables.

WHICH IS BEST?

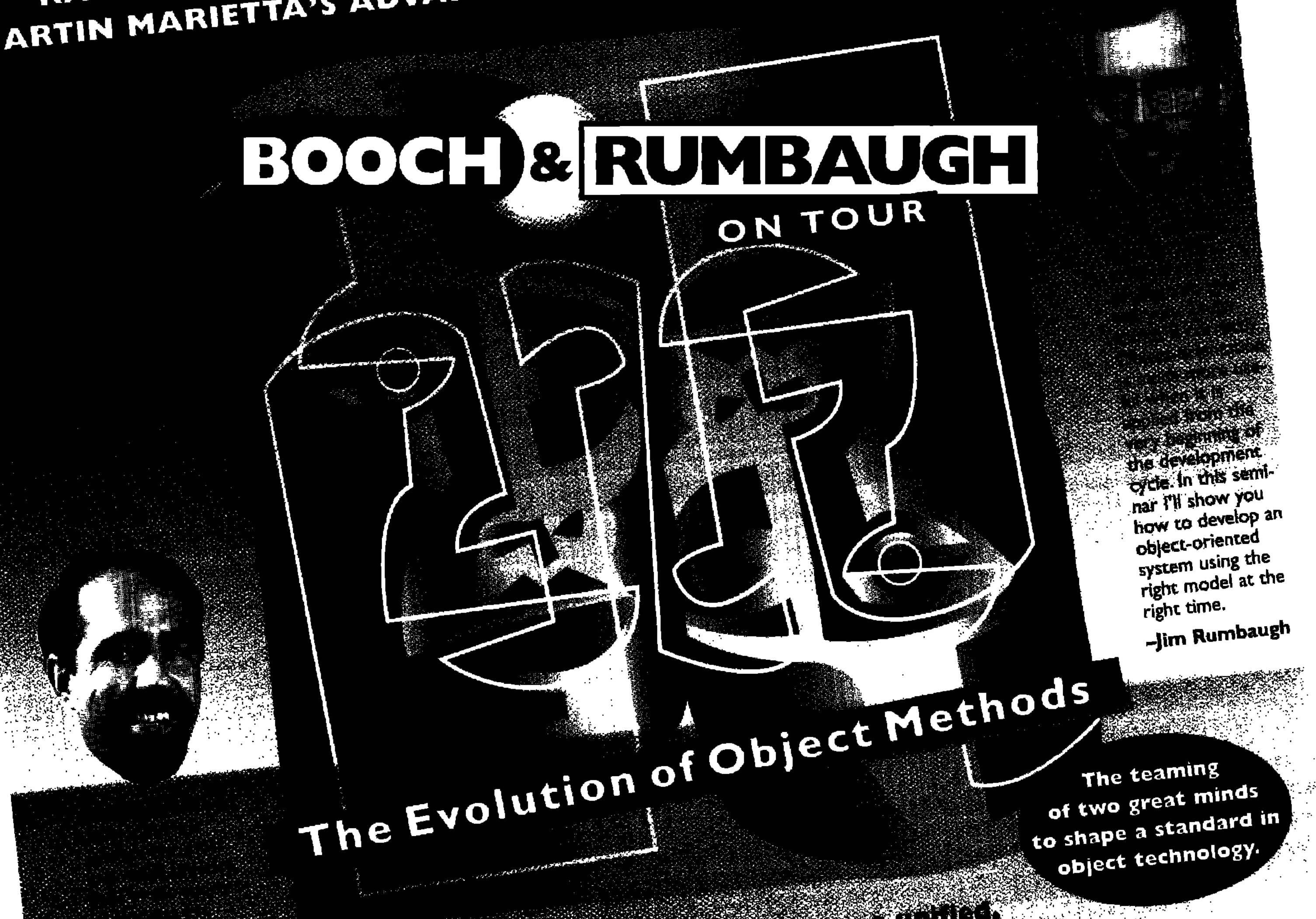
The ideal initialization mechanism depends on what you're trying to accomplish. My normal technique is to use super new initialize by default. If I start to run into difficulties or serious inefficiencies with that approach I'll use lazy initialization as it seems appropriate. One situation where lazy initialization is particularly useful is with class variables and class instance variables. There are lots of other issues associated with initializing classes. One reference for these is Juanita Ewing's article “Should classes be initialized?” (THE SMALLTALK REPORT: 1(3):6). ♀

DON'T MISS IT! APRIL 24-28

RATIONAL SOFTWARE CORPORATION, SIGS CONFERENCES, AND MARTIN MARIETTA'S ADVANCED CONCEPTS CENTER PROUDLY PRESENT:

BOOCH & RUMBAUGH ON TOUR

The Evolution of Object Methods



The teaming of two great minds to shape a standard in object technology.

[illegible] from the very beginning of the development cycle. In this seminar I'll show you how to develop an object-oriented system using the right model at the right time.

–Jim Rumbaugh

Hear from the creators themselves how a unified, industry-standard method is emerging. Spend one full day at this seminar and see for yourself how the two leading OOA&D methods are evolving to create a single, seamless method for successful development.

[illegible] seminar I'll explain the importance of architecture, the nature of a well-structured object-oriented architecture, and the pragmatics of the development process that help a team develop and refine an architecture.

–Grady Booch

AGENDA

- Conceptual models underlying the Booch method and the OMT
- A practical example illustrating both methods
- Roadmap to method unification
- Future Trends

April 24 Chicago
April 25 New York City
April 27 Dallas
April 28 San Francisco

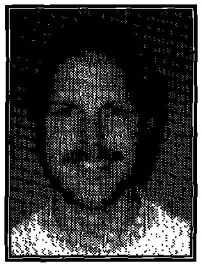
TO REGISTER
Phone: 212-242-7515
Fax: 212-242-7578
Email:
info@sigs.uucp.netcom.com
Price: $550 per person

Register by March 31 and get a FREE copy of Rational Rose® for Windows (Solo version) or OMTool™ for the PC.

ACT NOW! SEATING IS LIMITED.

CALL: 212-242-7515

RATIONAL SOFTWARE CORPORATION   SIGS CONFERENCES   MARTIN MARIETTA



Kent Beck

What? What happened to garbage collection?

I'LL TELL YOU what happened to garbage collection. I sat down three times to write the next column about garbage collection, and nothing came out. Between that, my wife's ten-game-winning streak at Cribbage, and 46 inches of rain so far this year (and it's still January), I'm pretty frustrated.

I've been reading Edward DeBono's book Thinking Course (highly recommended). One of the techniques he suggests, and also one I've seen in art, is when you're stuck, do the opposite. In art, if you're having trouble drawing a thing, try to draw something that is completely the opposite. Of course, it's impossible to draw "not a flower," so you end up with something which gets at "flowerness" backwards. I'm writing a column about "not garbage collection." I'm not sure where I'll end up, but at least the column will be done.

CLASS

Bob Williams pointed out a problem with the column I wrote a year and a half ago or so on instance specific behavior. The Smalltalk/V version works fine, but when you specialize an instance in VisualWorks, all of a sudden a bunch of code stops working. The problem? Class tests.

For example, Point comparison (comparison is where this happens most often) is implemented like this:

```
Point>>= aPoint
    ^self class = aPoint class and: [self x = aPoint x & (self y = aPoint y)]
```

It's implemented this way so you can put Points and Rectangles and Arrays and Strings and a bunch of other objects that don't respond to x and y in the same Set and not have things blow up. All well and good, until you start specializing instances.

The problem is that "class" returns whatever is in the class field of the object. Instance specialization in VisualWorks operates by creating an anonymous Class (really a Behavior), and setting the class of the instance to it. That way, you can make changes to the Behavior without affecting all the other objects of the same class.

The Point comparison code above, though, will fail, even if two Points are equal. If the receiver has been specialized and the argument not, the class of the receiver will be reported as this funny Behavior while the class of the argument is just good old Point. Are they equal? No way. Therefore the two Points aren't equal, even if they both print as "2@3".

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).

I turned to David Liebs, my own personal guide to VisualWorks arcana, for ideas. Here's what we came up with.

When you ask an object for its class, it should return a real Class, the thing you defined in the browser. If you want to use instance specific behavior in VisualWorks, you need to make the following changes. Note that if you try the following, the order in which the methods are defined is important. Trashing images is exciting, but it doesn't rank high on the productivity scale.

First, the class primitive, the one that just returns the contents of the receiver's class field, has to be renamed:

```
Object>>primClass
    <primitive: 111>
    self primitiveFailed
```

Next, we have to be able to go up the superclass chain looking for a real class. Instances of Class and MetaClass are real.

```
Class>>realClass
    ^self


MetaClass>>realClass
    ^self
```

Behavior, however, needs to ask its superclass for a real class. Note that this code ignores the kinky case of a Behavior without a superclass, which doesn't arise in normal use, nor in the instance specialization code. I'd have to think carefully about what I wanted the code to do in that case.

```
Behavior>>realClass
    ^self superclass realClass
```

Finally, Object>>class needs to be modified so it finds a real class:

```
Object>>class
    ^self primClass realClass
```

Now it works. You can specialize Points and still have "=" work correctly.

FORMATTING

On to the chosen secondary topic for the day—code formatting. What? You think this is a dull, dry, boring topic best relegated to corporate style guides? Not so. As soon suggest that typography is useless, that content is all that matters. The medium is the message—formatting your code is an opportunity to communicate subtle but important information to your readers. It is the first thing people will look at when they see your work. In groups, it is the one topic most likely to cause friction. Everybody has to do it the same or everyone is frustrated, but no one wants to do it like anyone else.

I decided to apply the power of patterns to the problem of source code formatting. Ward Cunningham and I used to have long discussions at Tektronix about just the right way to format a method. Roxie Rochat also produced an excellent style guide, which I didn't entirely agree with, but that took a comprehensive look at the issue of formatting. In the years since, I have often wondered if there were rational rules of formatting, or if it really was just a matter of personal style.

The appearance of the new Cooper and Peters product edIt, with its cool programmable formatting, also drove me to try to canonize my own formatting style.

When I started writing these patterns, I thought I'd end up with fifteen or twenty. As it turns out, I only found five, and Type Suggesting Parameter Name isn't really about formatting. The code they turn out isn't exactly like I would have formatted it before I enumerated the patterns, but I like it. It is simple and consistent, and it meets the main goals of code formatting.

What are the goals of formatting? As far as I can tell, the main forces influencing any code formatting style are:

- Minimize height—Formatting should produce the fewest possible number of lines, consistent with the rest of the constraints. This is important in Smalltalk, because fewer lines translates into more browsers, or less scrolling in the existing browsers.
- Minimize width—Formatting should produce code that doesn't have to be either scrolled horizontally or line wrapped. Line wrapping makes reading more difficult, because it messes up the shapes made by indentation, and horizontal scrolling slows down typing because you're always adjusting that darned scroll bar.
- Quick recognition—Formatting should produce code whose gross structure is apparent at a glance. Important features like flow of control and the presence of blocks should be obvious within a fraction of a second of seeing the code.
- Careful reading—Formatting should produce code that reads well in detail. You should be able to accurately read selectors. You should be able to understand the flow of control in detail.

These constraints are often in conflict. A good formatting style finds the right balance between them. I'm not saying that what follows is the be all and end all of formatting, but it is simple and consistent. If you disagree (and I'm sure some of you will), try to write up your own formatting style as patterns. Figure out what constraints you are resolving and how you are resolving them.

These patterns, and a whole lot more, also live on the Portland Pattern Repository, a Web server operated by Cunningham and Cunningham. Check them out by pointing your Web client at "http://c2.com/".

TYPE SUGGESTING PARAMETER NAME

What should you call a method parameter?

There are two important pieces of information associated with every variable—what messages it receives (its type) and what role it plays in the computation. Understanding the type and role of variables is important for understanding a piece of code.

Keywords to communicate their associated parameter's role. Since the keywords and parameters are together at the head of every method, the reader can easily understand a parameter's role without any effect on the name.

Smalltalk doesn't have a strong notion of types. A set of messages sent to an object appears nowhere in the language or programming environment. Because of this lack, there is no direct way to communicate types.

Classes sometimes play the role of types. You would expect a Number to be able to respond to messages like +, -, *, and /; or a Collection to do: and includes:. Therefore:

Name parameters according to their most general expected class, preceded by "a" or "an." If there is more than one parameter with the same expected class, precede the class with a descriptive word.

An Array that requires Integer keys names the parameters to at:put: as

```
at: anInteger put: anObject
```

A Dictionary, where the key can be any object, names the parameters:

```
at: keyObject put: valueObject
```

After you have named the parameters, you are ready to write the method. You may have to declare Role Suggesting Temporary Variables. You may need to format an Indented Control Flow. You may have to use a Guard Clause to protect the execution of the body of the method.

INDENTED CONTROL FLOW

You are writing a method following Type Suggesting Parameter Name.

How do you indent messages?

The conflicting needs of formatting to produce both few lines and short lines is thrown in high relief with this pattern. The only saving grace is that Composed Method creates methods with little enough functionality that you never need to deal with hundreds or thousands of words in a method.

One extreme would be to place all the keywords and arguments on the same line, no matter how long the method. This minimizes the length of the method, but makes it difficult to read.

If there are multiple keywords to a message, the fact that they all appear is important to communicate quickly to a scanning reader. By placing each keyword/argument pair on its own line, you can make it easy for the reader to recognize the presence of complex messages.

Arguments do not need to be aligned, unlike keywords, because readers seldom scan all the arguments. Arguments are only interesting in the context of their keyword. (This would be a good place for a diagram with an arrow going down the keywords in order to read at:put:, and another scanning left to right as the reader understand the message and its arguments.)

Therefore, put zero or one argument message on the same lines as the receiver.

```
foo isNil
2 + 3
a < b ifTrue: [...]
```

Put the keyword/argument pairs of messages with two or more keywords each on its own line, indented one tab.

```
a < b
	ifTrue: [...]
	ifFalse: [...]
array
	at: 5
	put: #abc
```

Rectangular Block formats blocks. Guard Clause prevents indenting from marching across the page.

Oddly enough, a company with possibly the largest and most deployable Smalltalk/OO workforce is virtually unknown - Until Now.

Over 400 Experienced Smalltalk/OO Developers, Mentors & Trainers Available Today.

Object*Intelligence*

The Object Services Company

- On-Site Smalltalk/OO Programming & Mentoring
- On-Site Customized Smalltalk/OO Training
- OODBMS Development: ObjectStore, Gemstone & Versant
- GUI Front-End Design/Build to Legacy Systems
- Object Modeling, Analysis & Design
- Smalltalk/Object Mapping to Sybase, Oracle & DB2

IBM VisualAge Solutions Provider

Call (919) 859-7384 or e-mail: **info@objectint.com**

Object*Intelligence* Corporation • 6300-138 Creedmoor Rd., Ste. 196 • Raleigh, NC 27612 • (919)848-0045 Fax

Smalltalk Idioms

RECTANGULAR BLOCK

How should you format blocks?

Smalltalk distinguishes between code that is executed immediately upon the activation of a method and code whose execution is deferred. To read code accurately, you must be able to quickly distinguish which code in a method falls into which category.

Code should occupy as few lines as possible, consistent with readability. Short methods are easier to assimilate quickly and they fit more easily into a browser. On the other hand, making it easy for the eye to pick out blocks is a reasonable use of extra lines.

One more resource we can bring to bear on this problem is the tendency of the eye to distinguish and interpolate vertical and horizontal lines. The square brackets used to signify blocks lead the eye to create the illusion of a whole rectangle even though one isn't there. Therefore:

Make blocks rectangular. Use the square brackets as the upper left and bottom right corners of the rectangle. If the statement in the block is simple, the block can fit on one line:

```
ifTrue: [self recomputeAngle]
```

If the statement is compound, bring the block onto its own line and indent:

```
ifTrue:
    [self clearCaches.
    self recomputeAngle]
```

GUARD CLAUSE

How should you format code that shouldn't execute if a condition holds?

In the bad old days of FORTRAN programming, when it was possible to have multiple entries and exits to a single routine, tracing the flow of control was a nightmare. Which statements in a routine got executed when was impossible to determine statically. This lead to the commandment "Every routine shall have one entry and one exit."

Smalltalk labors under few of the same constraints of long ago FORTRAN, but the prohibition against multiple exits persists. When routines are only a few lines long, understanding flow of control within a routine is simple, it is the flow between routines that becomes the legitimate focus of attention.

Multiple returns can simplify the formatting of code, particularly conditionals. What's more, the multiple return version of a method is often a more direct expression of the programmer's intent. Therefore:

Format conditionals that prevent the execution of the rest of a method with a return.

Let's say you have a method which connects a communication device only if the device isn't already connected. The single exit version of the method might be:

```
connect
    self isConnected
        ifFalse: [self connectConnection]
```

You can read this as "If I am not already connected, connect my connection." The guard clause version of the same method is:

continued on page 32

Smalltalk Developers

JMB Realty Corporation, one of the nation's largest and most diversified real estate owners and managers, is developing high impact knowledge-based software for the retail real estate industry.

This is an opportunity for you to play a significant role in state-of-the-art product development, and to work in a unique, highly collaborative environment. If you're a professional with strong business and OO or GUI experience, please mail your confidential resume to:

JMB Realty Corporation
Information Services Recruiting
900 N. Michigan Ave ▪ Chicago, Illinois 60611
Internet: LLN@jmbcorp.mhs.compuserve.com
Fax: 312-915-1193

JOIN THE FASTEST GROWING LEADER IN HEALTHCARE SOFTWARE

Information Technology Professionals

Atlanta, GA, Amherst, MA, Minneapolis, MN & Central Florida

HBO & Company (HBOC) designs, develops and maintains software solutions for our business partner hospitals and other members of the healthcare enterprise. Celebrating over 20 years of profitable growth and with 1994 revenues exceeding $300 million, HBOC is the second largest and fastest growing provider in our industry.

Smalltalk

The ideal candidates will have experience with object-oriented analysis and design, PC software development and Smalltalk programming.

Visual C++

Positions require 2+ years of development experience with Visual C++ in a Windows environment.

Sybase

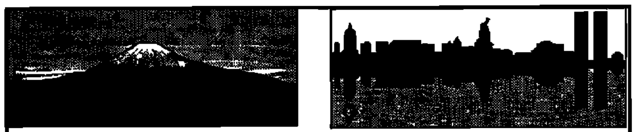
Requirements include 2+ years Sybase SQL Server administration experience and knowledge of relational table designs.

The professionals we seek must possess excellent communication skills and the ability to work in a team environment.

HBOC offers excellent benefits, competitive salaries and a team-oriented professional work environment where promotion from within is the norm. Forward/fax/email your resume to: **HBO & Company, Corporate Recruiting, STR3/95, 301 Perimeter Center North, Atlanta, GA 30346; fax to: (404) 393-6063; email to: lisa.phillips@hboc.com.**

An Equal Opportunity Employer, M/F/D/V.

HBO &Company

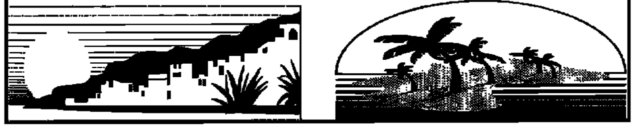


Smalltalk

Pick Your Location

Smalltalk careers from Wall Street to Main Street. Access the best Smalltalk cultures & applications. Contract & direct careers nationwide. Client fee paid.

800-220-1044
Paul Morris Personnel
Send ascii to ptm@pmorris.com



To advertise in the Recruitment section, please call Mike Peck at 212.242.SIGS

OBJECT TECHNOLOGY PROFESSIONALS

We're just as committed to object technology as you are.

At SHL SYSTEMHOUSE, client/server computing isn't just a part of our business. It is our business. We're a billion dollar systems integrator dedicated entirely to business transformation through client/server computing.

And we're using object technology to make these transformations a reality.

Join SHL and help us build mission critical applications using object technology from analysis to construction. We'll challenge you and support you. You'll collaborate with the industry's top object technology professionals. And you'll make a major impact.

As a leader in a $100 billion industry, our potential for growth is extraordinary. If you've got the knowledge, imagination and vision, your career opportunities at SHL are endless.

If you're committed to object technology, join a company that is firmly committed to your future. Please send your resume and letter of introduction to: SHL, Manager of Human Resources, Dept. MJS-395, 300 South Wacker Dr., Suite 2500, Chicago, IL 60606. FAX: (312) 939-0066. E-mail: oops@chi.shl.com.
SHL is an Equal Opportunity Employer M/F/D/V.

SHL SYSTEMHOUSE

SHL SYSTEMHOUSE

Smalltalk Developers

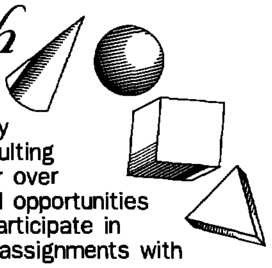
Experienced Smalltalk and ENVY/Developer consultants wanted for 3-6 month contracts in Europe. OS/2 and Visual Age experience helpful.

Lingo Allegro U.S.A., Inc.
113 McHenry Road, Suite 161
Buffalo Grove, Illinois, 60089 USA
Phone: +1 312 203 4926
Fax: +1 708 459 8501

Lingo Allegro

SUCCESS

With



Smalltalk Developers

At QSYS we have successfully provided Object Oriented consulting services to our customers for over seven years. This has created opportunities for Smalltalk Specialists to participate in leading edge, mission critical assignments with our Fortune 1000 clients.

If you have demonstrated experience implementing large OO systems using IBM Smalltalk or Visual Age,™ ParcPlace VisualWorks,® Digitalk Smalltalk/V,® we would like to hear from you!

For further information, contact
Elspeth Koor at **1-800-999-9776.**

1 Yonge Street, Suite 1801, Toronto, Canada
M5E 1W7 Fax: (416) 369-0515

QSYS

90 Park Avenue, Suite 1600, New York, NY
10016 Telephone: (212) 984-0715

Email: 72072.2575@compuserve.com

Smalltalk Idioms

continued from page 29

```
connect
    self isConnected ifTrue: [^self].
    self connectConnection
```

You can read this as "Don't do anything if I am connected. Connect my connection." The guard clause is more a statement of fact, or an invariant, than a path of control to be followed.

You may need to return a Nil Return Value to signal an unusual condition.

SIMPLE ENUMERATION PARAMETER

What should you call the parameter to an enumeration block?

It is tempting to try to pack as much meaning as possible into every name. Certainly, classes, instance variables, and messages deserve careful attention. Each of these elements can communicate volumes about your intent as you program.

Some variables just don't deserve such attention. Variables that are always used the same way, where their meaning can be easily understood from context, call for consistency over creativity. The effort to carefully name such variables is wasted, because no non-obvious information is communicated to the program. They may even be counter productive, if the reader tries to impute meaning to the variable that isn't there.

Call the parameter "each". If you have nested enumeration blocks, append a descriptive word to all parameter names.

For example, the meaning of "each" in:

```
self children do: [:each | self processChild: each]
```

is clear. If the block is more complicated, each may not be descriptive enough. In that case, you should invoke Composed method to turn the block into a single message. The Type Suggesting Parameter in the new method will clarify the meaning of the object.

The typical example of nested blocks is iterating over the two dimensions of a bitmap:

```
1 to: self width do:
    [:eachX |
    1 to: self height do:
        [:eachY | ...]]
```

Nested blocks that iterate over unlike collections should probably be factored with Composed Method.

You may need Composed Method to simplify the enumeration block.

INTERESTING RETURN VALUE

When should you explicitly return a value at the end of a method?

All messages sends return a value. If a method does not explicitly return a value, the receiver of the message is returned by default. This causes some confusion for new programmers, who may be used to Pascal's distinction between procedures and functions, or C's lack of a definition of the return value of a procedure with no explicit return. To compensate, some programmers always explicitly return a value from every method.

The distinction between methods which do their work by side effect and those that are valuable for the result they return is important. An unfamiliar reader wanting to quickly understand the expected use of a method should be able to glance at the last line an instantly understand whether a useful object is generated or not. Therefore:

Return a value only when you intend for the sender to use the value.

For example, consider the implementation of topComponent. Visual components form a tree, with a ScheduledWindow at the root. Any component in the tree can fetch the root, by sending itself the message topComponent. VisualPart implements this message by asking the container for its topComponent:

```
VisualPart>>topComponent
    ^container topComponent
```

ScheduledWindow implements the base case of the recursion by returning itself. The simplest implementation would be to have a method with no statements. It would return the receiver. However, using Interesting Return Value, because the result is intended to be used by the sender, it explicitly returns self.

```
ScheduledWindow>>topComponent
    ^self
```

VisualWorks dialog development

continued from page 13

SUMMARY

In VisualWorks, a dialog is an application model that opens a modal window. There are four basic purposes for dialogs: displaying simple messages, acquiring simple information, providing application specific services, and editing objects. The stock dialogs provided by the Dialog class provide basic dialog functionality but there is still a need for custom dialogs. There are a few drawbacks with the current approach to custom dialog development in VisualWorks. A dialog that is a subclass of SimpleDialog cannot leverage off of the powerful features provided in ExtendedApplicationModel, nor can it implement the accept and cancel methods, nor does it provide the option of opening a nonmodal window. A dialog that is a subclass of ApplicationModel cannot access components during runtime and its pre- and postbuild methods are never executed. For these reasons, the ExtendedSimpleDialog class was created to compliment the ExtendedApplicationModel class and facilitate custom dialog development. A dialog should be designed as a subclass of ExtendedApplicationModel. This provides the following benefits: execution of pre- and postbuild methods, accept and cancel action method execution, runtime interface access, option for modal or nonmodal versions of the interface, and all the additional features of ExtendedApplicationModel. Full source code for ExtendedSimpleDialog and ExtendedApplicationModel, as well as examples, is available from the archives at the University of Illinois (st.cs.uiuc.edu). ♀

Reference

1. Howard, T., and B. Kohl, Extending the application model , SMALLTALK REPORT, 3(7): 1–7, May 1994.

Tim Howard is a senior consultant at FH Protocol, Inc. He is interested in application development using OO technologies in general, and using the language of Smalltalk in particular. He can be reached at 74213.1517@compuserve.com or 214.931.5319.