The Smalltalk Report

The International Newsletter for Smalltalk Programmers

March/April 1993 **Volume 2 Number 6**

THE MULTIPLE DOCUMENT INTERFACE

By Tarik Kerroum & Stephane Lizeray

The Multiple Document Interface is a specification for applications written for the Microsoft Windows environment or OS/2 Presentation Manager. The specification describes a window structure and user interface that allow the user to work with multiple documents in a single application. The MDI specification dates back to Windows 2.0 and was first used in Excel for Windows. Since then, most commercial applications implement it for managing multiple documents. This article provides a technical discussion of how MDI support is implemented in the recently released Smalltalk/V Windows 2.0, which comes with full MDI support.

MDI is described in SAA, Common User Access (ADVANCED INTERFACE DESIGN GUIDE, IBM 1989). Windows 3.0 includes an MDI API interface, which greatly simplifies the development of an MDI application.

The main window of an MDI application looks conventional. This MDI frame window may contain child windows, called *MDI child windows*, which are used to display program output. As the name indicates, MDI child windows are supposed to display documents. Each of these MDI children may have a menu that is displayed on the MDI frame's menu bar when the child is active. Only one child is active at any given time.

When maximized, an MDI child window takes the appearance displayed in Figure 1. Minimized documents appear as icons at the bottom of the MDI frame's client area. Each child window may have its own icon.

continued on page 4...

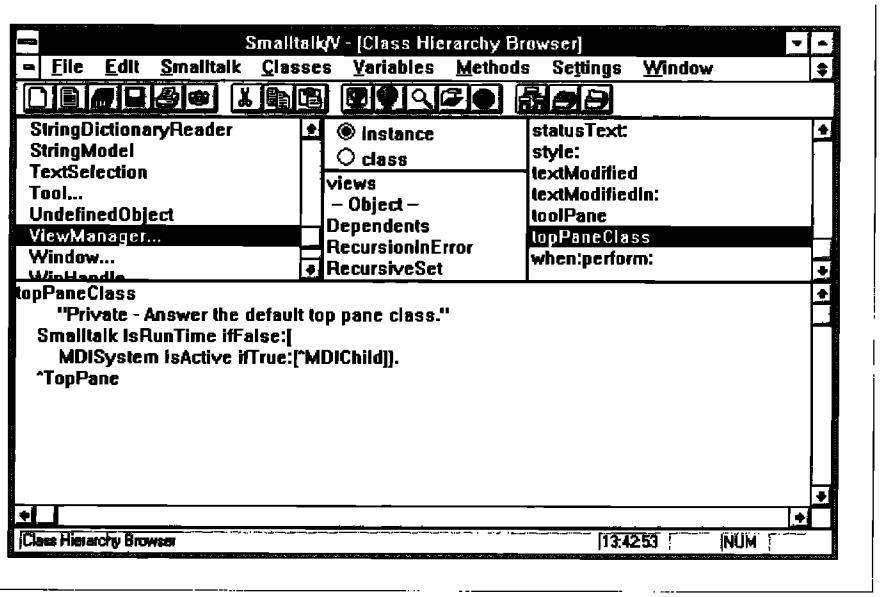


Figure 1: Maximized MDI Child Window.

Contents:

John Pugh

Paul White

EDITORS' CORNER

While most people introduced to object-oriented technology are receptive to the "new" ideas and processes, there are some (the doubting Thomases) who can see the potential benefits but find it difficult to visualize how they can be achieved within their own projects and development teams. Reuse, for example, requires that one team have a positive attitude to using components produced by another. In many organizations, communication is poor even between groups who share the same office space let alone teams who are physically remote from one another. In addition, the "not invented here" syndrome is prevalent—no one trusts anything they did not construct themselves. A cultural change is required to reap the benefits of the technology. A similar change in culture is required of new Smalltalk programmers: they must have an open mind and be trusting of existing objects. Of course care must be taken to ensure the integrity of objects, but objects can't act so paranoid that they don't trust anyone sending them messages. The objects already defined in the library don't do this—arrays don't do bounds checking when someone is accessing them; points don't check to see if their *x* and *y* values being set are numbers. In an organization, synergy cannot be achieved if people aren't flexible and trusting; the same is true of software objects.

The feature article this month by Stephane Lizeray and Tarik Kerroum examines Smalltalk's support for Multiple Document Interface (MDI). As they point out, both Windows and OS/2 provide a specification for applications to deal with multiple documents open within them. They describe in detail the classes included in Smalltalk/V Windows 2.0 to support MDI, explaining the features and mechanisms provided.

In her column this month, Rebecca Wirfs-Brock continues her discussion of the role of classification in our designs. As she rightly points out, it is important for us to understand not only the characteristics of the objects we are using, but also the different types of interactions that can occur between our objects. She makes the case that it is the relationships between our objects that capture the dynamic nature of our systems.

Kent Beck begins a discussion on instance-specific behavior, in which methods can be attached to individual instances, as opposed to being attached to the class only. The power of this metaphor is already being tapped in other languages, notably Self, but as Kent points out, it may be something that good Smalltalk programmers can take advantage of immediately.

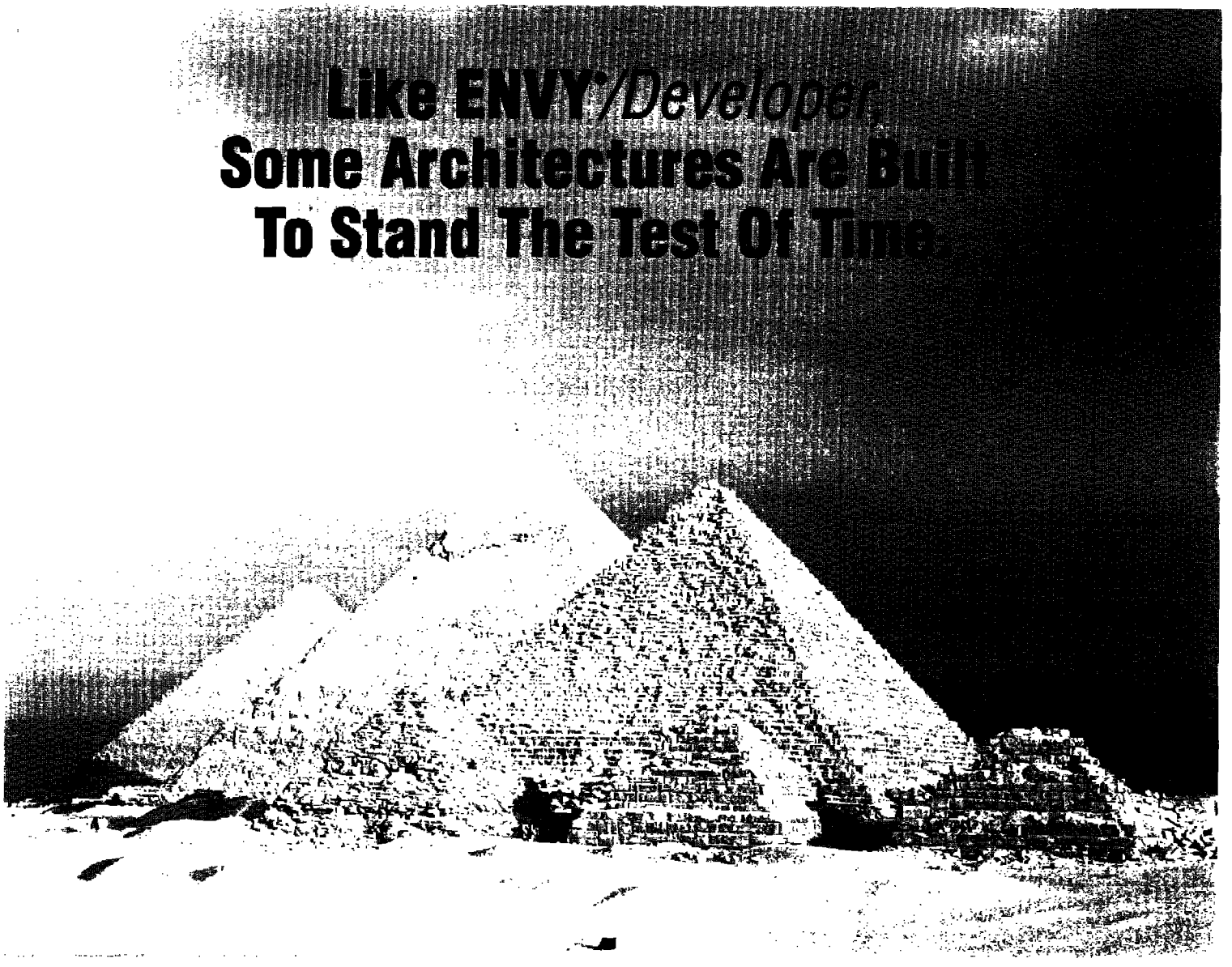
Reflection is the topic of interest selected by Alan Knight in this month's "The Best of comp.lang.smalltalk" column. Alan takes us through a thread of discussion on the bulletin board examining both the ability and the merit of making fundamental changes to the way Smalltalk is structured. Alan's conclusion seems to be, and most experienced Smalltalkers would undoubtedly agree, it's a messy and dangerous thing to do, but it's terrific to have the opportunity to do it.

Finally, Greg Hendley and Eric Smith identify a number of rules of thumb for GUI-based application development.

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by SIGS Publications Group, 588 Broadway, New York, NY 10012 (212)274-0640.

Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada.

Printed in the United States



ENVY/*Developer*: The Proven Standard For Smalltalk Development

An Architecture You Can Build On

ENVY/*Developer* is a multi-user environment designed for serious Smalltalk development. From team programming to corporate reuse strategies, **ENVY/*Developer*** provides a flexible framework that can grow with you to meet the needs of tomorrow. Here are some of the features that have made **ENVY/*Developer*** the industry's standard Smalltalk development environment:

Allows Concurrent Developers

Multiple developers access a shared repository to concurrently develop applications. Changes and enhancements are immediately available to all members of the development team. This enables constant unit and system integration and test – removing the requirement for costly error-prone load builds.

Enables Corporate Software Reuse

ENVY/*Developer*'s object-oriented architecture actually encourages code reuse. Using this framework, the developer creates new applications by assembling existing components or by creating new components. This process can reduce development costs and time, while increasing application reliability.

Offers A Complete Version Control And Configuration Management System

ENVY/*Developer* allows an individual to version and release as much or as little of a project as required. This automatically creates a project management chain that simplifies tracking and maintaining projects. In addition, these tools also make **ENVY/*Developer*** ideal for multi-stream development.

Provides 'Real' Multi-Platform Development

With **ENVY/*Developer***, platform-specific code can be isolated from the generic application code. As a result, application development can parallel platform-specific development, without wasted effort or code replication.

Supports Different Smalltalk Vendors

ENVY/*Developer* supports both Objectworks\Smalltalk and Smalltalk/V. And that means you can enjoy the benefits of **ENVY/*Developer*** regardless of the Smalltalk you choose.

For the last 3 years, Fortune 500 customers have been using **ENVY/*Developer*** to deliver Smalltalk applications. For more information, call either Object Technology International or our U.S. distributor, Knowledge Systems Corporation today!

Object Technology International Inc
2670 Queensview Drive
Ottawa, Ontario K2B 8K1

Ottawa Office
Phone: (613) 820-1200
Fax: (613) 820-1202
E-mail: info@oti.on.ca

Phoenix Office
Phone: (602) 222-9519
Fax: (602) 222-8503

Knowledge Systems Corporation
114 MacKenan Drive, Suite 100
Cary, North Carolina 27511
Phone: (919) 481-4000
Fax: (919) 460-9044

continued from page 1

Key accelerators allow users to close documents, invoke system menus, and switch among MDI child windows. The menu bar has a Window menu that allows the user to arrange the child documents—supported activities include cascading, tiling, and arranging icons. The submenu has a list of all the child documents, referenced by title.

THE WINDOWS 3.X MDI SUPPORT AND THE SMALLTALK MDI HIERARCHY

What follows is a short summary of the structure of an MDI application. Please refer to the Microsoft SDK reference manual, GUIDE TO PROGRAMMING, MULTIPLE DOCUMENT INTERFACE, for additional information. The MDI application structure is a key factor in the Smalltalk class design (see Figure 2).

The message loop of an MDI application is similar to a normal message loop, with the exception of menu accelerator handling, which is handled by the Notifier.

The main window, called the *frame window*, resembles a normal top-level window except that a special window, the *MDI client window*, fills all or part of its client area. The frame window has its own default procedure, the DefFrameProc. It is essential for a certain number of window messages to reach the default procedure; otherwise, the MDI functionality will be impaired. The frame's menu is set by sending the message WmMdisetmenu to the client, with the handle of the Window menu as additional parameter.

The client window is a preregistered window maintained by Windows. Additional creation parameters indicate the handle of the Window pop-up menu (more about this later) and the ID of the first MDI child document. This information is required so that Windows can maintain the list of MDI children under the Window pop-up menu.

MDI children are displayed on top of the client window. Usually, the client window is resized so that it fills the frame's client area. In certain cases, such as when the frame displays a status pane and/or a tool pane, its size may be less. The client resizing is handled by the MDI frame window, similar to the reframing of other child windows (but not MDI child windows). Besides just being there, the MDIClient class does not do much.

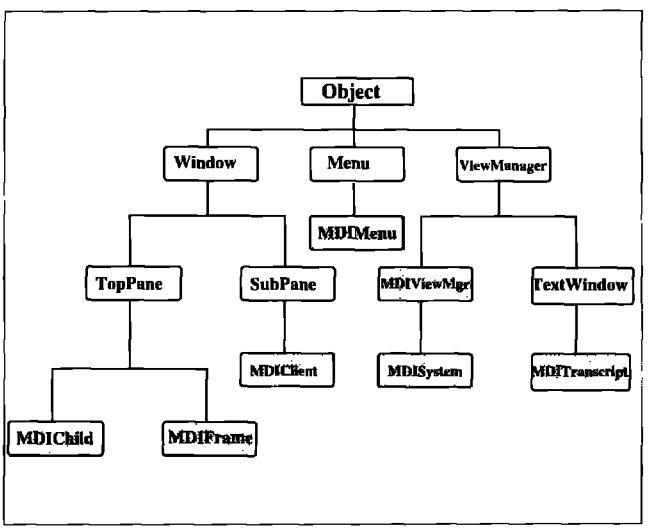


Figure 2: MDI class hierarchy.

The child window looks and acts very much like a top-level window, except that it cannot move outside the client window and does not have a menu. Instead, logically attached menu items are displayed by the frame.

An MDIChild is not created like a regular window; instead, mdiCreate is passed to the frame window with additional information in an MDICreateStruct. The WmMDICreate message is then passed to the client window and handled by Windows.

The MDIChild has its own default procedure that is set during the creation message processing. Note also that no other MDIChild should be created while an MDIChild is being created. The creation process starts by sending mdiCreate to the client and ends with the MDIChild returning from the wmCreate message.

The MDI classes in Smalltalk/V 2.0 offer a transparent framework for writing or integrating MDI applications. An important asset is the menu handling support, which is currently unique to Smalltalk /V 2.0.

> “The child window looks and acts very much like a top-level window, except that it cannot move outside the client window and does not have a menu.”

There are two kinds of MDI menus: the original frame menu and menus added by MDIChild documents. Each time an MDIChild is activated, its menu becomes the frame menu. On the other hand, the original frame menu stays for the duration of the MDI session. The Window menu represents a special case because the part where child document titles are added is managed by Windows.

Another feature provided by the Smalltalk/V Windows 2.0 MDI classes is automatic numbering of the child titles. Child documents with the same title receive a number as suffix. For instance, a child titled MyChild changes to MyChild:1 if you create a new document with the same title, and the new one is labeled MyChild:2. This is part of the CUA specifications. For the application, the numbering is transparent because the title the application manipulates differs from the actual caption text.

Given this information, the Smalltalk hierarchy design is straightforward. MDIFrame and MDIChild are both subclasses of TopPane, while MDIClient is a subclass of ControlPane (a child window whose window procedure is managed by Windows).

THE MDI PROGRAMMING INTERFACE

The MDI classes take advantage of the ViewManager class. The two main benefits of the ViewManager are:

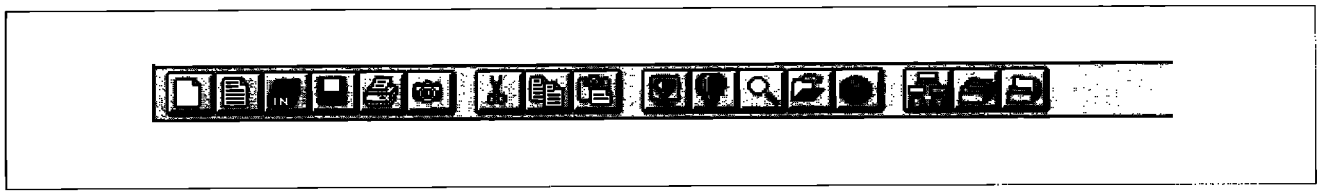


Figure 3: The ToolPane

- clean separation of the user interface from the application logic
- creation of applications that have multiple views on the same data

An MDI application will need one object per MDI child in order to maintain data unique to each window. There are no limitations regarding the capabilities of the child documents. They may have any subpanes as children.

The TopPane class is specified by the method ViewManager>>topPaneClass and defaults to TopPane. Specifying MDIChild instead will make the receiver's main view an MDI Child window, without further modifications. This transparency allows applications such as TextWindow, Class Hierarchy Browser, and Graphics Demo to function unchanged in MDI mode. In addition, the Smalltalk workplace can be switched back and forth between standard and MDI mode.

EXAMPLE: A WINDOWS SYSTEM EDITOR

SysEdit is an MDI application for presenting the following Windows system configuration files:

win.Ini
system.ini
config.sys
autoexec.bat

This application is provided in Windows 3.x in the system directory. A similar application, the multipad, is also provided in Windows SDK. We have chosen to rewrite this application in Smalltalk by using the MDI classes. In Listing 1 we show how to reuse ViewManager subclasses in an MDI application.

In the open method, we create MDIFrame and StatusPane and invoke the createDocuments method reponsible for the creation of the documents. Each MDI document's owner is a TextWindow object. By sending the frame: message to a TextWindow object, we make the TextWindow mainView an MDIChild and at the same time set the frame window of this MDIChild object.

The MDIMenu is used for the standard Window menu, which displays the active documents. The StatusPane will be used to display the title of the current active document and to display hint texts for the menu items.

In this example, we take full advantage of one of the promises of object-oriented technology: reusability. The MDI classes inherit from the TopPane class and reuse the usual TopPanes' owners (ViewManager subclasses).

ADDITIONAL FEATURES OF THE SMALLTALK/V WINDOWS 2.0 MDI CLASSES

The MDI extras include two supplemental windows—the ToolPane and the StatusPane. These are child windows that can be added transparently to the top and bottom area of a topPane's client area. The key element allowing this transparency is the method freeClientArea: aRectangle, which is sent to child windows prior to resizing. The ToolPane and the StatusPane will both modify the rectangle to exclude their region, if they are visible. The resulting rectangle is then passed in the resize: message and becomes the argument of the framing block in each child window. This feature allows these supplemental windows to be added without modification to the application's code.

The ToolPane uses Tools to specify the bitmap and the action of a tool button. Each tool has the functionality of an owner-drawn button. It is actually drawn by the ToolPane for performance reasons. The ToolPane uses DIBs (Device Independent Bitmaps) for representing highlighted and normal states. The DIB technique can be generalized to any kind of owner drawn controls; it provides fast and efficient mapping to system colors. System colors, such as the button face color, are customized through the control panel. To have ToolPane support these customizations, we define the Tool bitmap as a DIB with arbitrary colors; for example, blue at palette index 3 for the button background. At runtime, we load the DIB and change the color entries in the color table to the actual system colors. For example, index 3 will get the current button face color. We then realize the resulting palette and get a DDB (Device Dependent Bitmap) with the correct colors. Of course, this process has to be repeated if the user changes system colors during the session (see Figure 3).

“An MDI application will need one object per MDI child in order to maintain data unique to each window.”

DIB manipulations provide a fast and efficient means to achieve color support. The drawback is that there is currently no standard that defines system palette entries; that is, palette index 3 may have a different meaning for another owner-drawn control.

The StatusPane is a static pane (the user cannot interact with it) that displays status or help information. Its design fol-

Listing 1. The SysEdit implementation.

```
MDIViewManager subclass: #SysEdit
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries:
    'WinConstants VirtualKeyConstants '

!SysEdit class methods

mdiMenu
    "Private - Answer a standard MDI Window menu"
    ^MDIMenu new
        appendItem: '&New Window' selector: #mdiNewWindow
            accelKey:$n accelBits:AfControl;
        appendItem: '&Cascade Shift+F5' selector: #mdiCascade
            accelKey: VkF5 accelBits: AfVirtualkey|AfShift ;
        appendItem: '&Tile Shift+F4' selector: #mdiTile
            accelKey: VkF4 accelBits: AfVirtualkey|AfShift ;
        appendItem: 'Arrange &Icons' selector: #mdiArrange;
        title: '&Window'.

optionMenu
    "Private - Answer the option menu"
    ^Menu new
        appendItem: '&StatusPane' selector:#toggleStatusPane;
        title: '&Options'.

!SysEdit methods

childActivate:aPane
    "Private - Update the StatusPane"
    |mdiActive|
    (mdiActive := self frame mdiGetActive) notNil
        ifTrue:[(self statusPane statusBoxAt: #status)
            contents: mdiActive label].

close:aPane
    "Private - Close the receiver application"
    Smalltalk isRunTime
     ifTrue: [
          (MessageBox confirm: 'Are you sure you want to exit?')
              ifTrue: [self close.
                  ^Smalltalk exit]
              ifFalse: [^self]]
     ifFalse: [^self close].

createDocuments
    "Private - Create the MDI documents"
    |buffer winaddress pathName file|
    buffer:=String new: 160.
    winaddress:=WinAddress copyToNonSmalltalkMemory: buffer.
    KernelLibrary getWindowsDirectory: winaddress asParameter
        length:buffer size.
    pathName:=String fromAddress:winaddress.
    winaddress unlockAndFree.
    self openTextWindow:pathName,'\SYSTEM.INI'.
self openTextWindow:pathName,'\WIN.INI'.
    self openTextWindow:'C:\CONFIG.SYS'.
    self openTextWindow:'C:\AUTOEXEC.BAT'.

label
    "Private - Answer the receiver's mainView label"
    ^'System Configuration Editor'

mdiMenu:anMDIFrame
    "Private - Create the menuBar for the frame"
    anMDIFrame mdiMenuWindow
addMenu: self class optionMenu owner: self.
    anMDIFrame mdiMenuWindow
        addMenu: self class mdiMenu owner:self.

open
    "Create and open a SysEdit application"
    self addView: (self frame:
        (MDIFrame new
                owner: self;
                icon: (Icon fromModule: self resourceDLLFile
                    id:'Default');
                when: #mdiMenuBuilt perform: #mdiMenu:;
                when: #childActivate perform: #childActivate:;
                when: #close perform:#close:;
            label: self label)).
        self addSubpane:(
            StatusPane new
        owner:self;
                when: #getContents perform: #statusPane:).

    self openWindow.
    self toggleMenu:'&Options' item:#toggleStatusPane.
    self createDocuments.

openTextWindow:aFileName
    "Private - Open a TexWindow application as an MDI document"
    |file|
    file := File pathName:aFileName.
    (self statusPane statusBoxAt: #status) contents: file file name.
    TextWindow new
        frame:self frame;
        disableSystemMenuItemClose;
        icon:(Icon
            fromModule: self resourceDLLFile id:'TextWindow');
        openOnFile: file.
    file close.

resourceDLLFile
    "Private - Answer the DLL filename for resources"
    ^Icon defaultDLLFileName

statusPane: aStatusPane
    "Private - Set the StatusPane contents"
    | statusBoxes |
    statusBoxes := OrderedCollection new
        add: (StatusBox new
            space: aStatusPane font width;
            name: #status);
        yourself.
    aStatusPane contents: statusBoxes.

statusPaneHelp:aKey
    "Private - Answer the hint text to the StatusPane"
    ^HelpSysEdit at:aKey ifAbsent:[^super statusPaneHelp:aKey].

toggleMenu: menuName item: itemName
    "Private - Toggle the selected menu item"
    | theMenu aBoolean|
    theMenu := self frame menuWindow menuTitled: menuName.
    (aBoolean:=theMenu isChecked:itemName)
        ifTrue: [self frame
            uncheckItem: itemName
            forAllMDIChildMenus: menuName]
        ifFalse: [self frame
            checkItem: itemName
            forAllMDIChildMenus:menuName].
    ^aBoolean!

toggleStatusPane
    "Private - Show/Hide the ToolPane"
    self toggleMenu:'&Options' item:#toggleStatusPane.
    self statusPane show.
    self mdiArrange.
```

lows the lines of the ToolPane—each statusPane maintains a collection of StatusBoxes to display information. A StatusBox appears as a small box with a 3-D effect. The StatusBoxes can be left- or right-justified or can resize themselves according to the dimensions of the frame's client area.

The application may display information in the boxes or assign menu help text for each menu item. The help text is then displayed in the leftmost status box as the user scrolls through the menu. For this to take effect, the application defines a dictionary that associates the selectors to the help text.

CONCLUSION

The MDI classes in Smalltalk/V 2.0 offer powerful Windows 3.x support that compares favorably to other interactive MDI implementations such as Visual Basic's. The Smalltalk application programmer gets the tools and the technology to create professional-looking Windows applications.

Future enhancements of the MDI interface will include improved frame menu support such as the ability to add items dynamically (for example, WinWord does this for the File menu that lists the most recently loaded files). Still, the MDI project has been an important experience regarding Windows 3.x integration. We believe that Windows integration will play an ever increasing role in the Smalltalk/V Windows product; it may well end up as the single most important factor when considering development products. Except for the window procedure which is hidden, Smalltalk/V is open and flexible enough to accommodate new functionality such as the MDI.

When designing a Windows-related hierarchy, it is important to follow the SDK documentation and to model the classes around it. The MDI class design results directly from the MDI internal Windows architecture. Take existing commercial applications as a reference. With a little guesswork and the resource workshop, it is not difficult to guess how graphical functionality such as ribbon windows is implemented.

Windows is a cooperative environment where performance and resource management is an issue. For example, the system heaps (User, GDI) are limited to 64K and all the memory in the world cannot change this fact. We usually think of performance/resource requirements and clean object-oriented design as two impeding issues. It is generally a good idea to experiment with different algorithms and then implement one in the cleanest possible way. If the algorithm is fast and efficient, the object-oriented implementation won't have to trade design quality against performance. ■

Tarik Kerroum is working with Siemens Nixdorf Ges.m.b.H. He holds an engineer's degree from Ecole Centrale de Paris. Stephane Lizeray is currently a staff engineer at Siemens Nixdorf where he is working on FINIS, a customizable next-generation banking software using Smalltalk. His specialty is custom graphical user inerfaces using Smalltalk/V. They can be contacted at Siemens Nixdorf Ges.m.b.H. Obere Donaustrasse 19-27 A-1020 Vienna, Austria or via email at 70262.1762@compuserve.com (Tarik Kerroum) and 70474.3003@compuserve.com (Stephane Lizeray).

Rebecca Wirfs-Brock

Characterizing object interactions

In my last column I discussed ways to characterize object behaviors. I introduced a vocabulary for describing individual objects that enabled us to discriminate between alternative ways to design an object's responsibilities. To further develop our model, we need to design interactions *between* objects. This column highlights some considerations for these designing interactions. Before we start, let's review how to further characterize object roles and behaviors.

A BRIEF REVIEW

We distinguished whether an object serves a non–application-specific purpose or if it models a concept specific to the problem domain. Utility objects are generally useful, non–application-specific objects. Business objects model some essential aspect of the problem domain. Business objects have some correlation with concepts familiar to our software users. We can further characterize our objects' behavior:

- Controlling objects are responsible for performing a cycle of action.
- Coordinating objects are the traffic cops and managers within a system: pairing client requests with objects performing the desired service.
- Structuring objects primarily maintain relationships between other objects.
- Informational objects hold values that other objects can ask about.
- Service objects typically perform a single operation or activity on demand.
- Interface objects support communication between objects within our program and external systems or users.

Behavioral stereotypes are a useful starting point for thinking about objects. However, a stereotype represents an oversimplified viewpoint. As we look more closely, we may discover that objects seem to fit several behavioral profiles. For instance, an object can coordinate activities and structure other objects.

ADDING DETAIL

People often try to fit objects into a behavioral profile too early. If you start with objects from the user's vocabulary, you may have only a set of informational objects and their information content. It is easy to be missing any notion of how the application should work! We need to work out how various objects might perform required tasks. In the process of figuring out these details, we undoubtedly will invent new objects that embody more varied behavior. Then we will have a richer set of objects whose behavior we can stereotype.

In the very early stages, functionality may not be cleanly separated into many distinct objects. We may have objects with multiple behaviors, rather than many different objects each with a singular behavior. As a consequence, few if any service objects have been identified. Another indicator of an early design is that control is mixed in with other behaviors in a seemingly haphazard fashion.

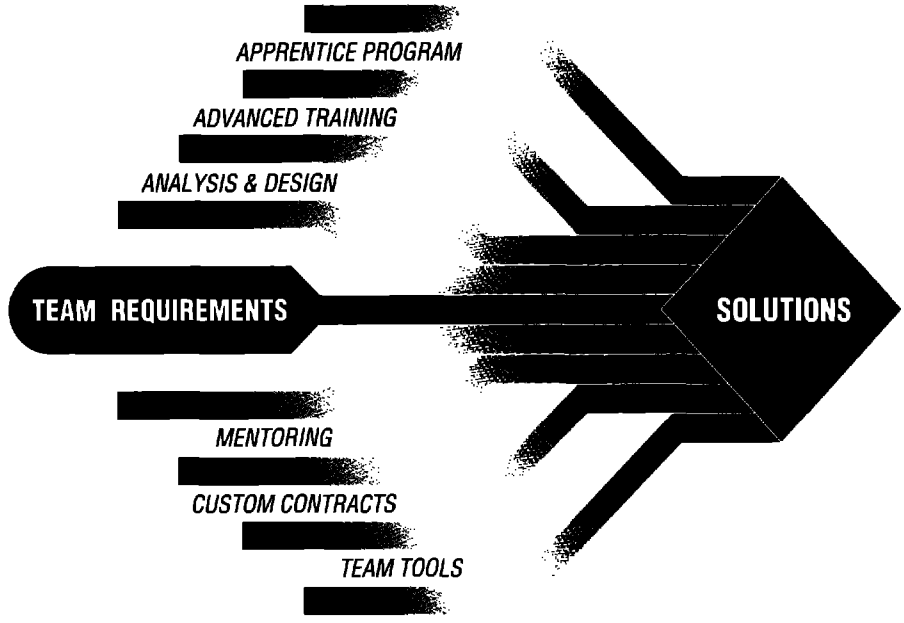
As I gain more experience, I create more designs with highly specialized classes. Classes are inexpensive conceptual tools that I use to divide and conquer my modeling problems. If I find out, as I work out more details, that a class adds unnecessary complexity, I simply collapse its behavior into its clients. On the other hand, I often find that I can rework smaller concepts more easily so as to be usable in several places within a single application.

MAKING DECISIONS

Before I can go on much further, I need to decide upon a control strategy. I need to develop a predominant pattern for distributing the flow of control and sequencing of actions among collaborating objects. I also need to understand how each object accomplishes its tasks. A class may incorporate more or less intelligence depending on how much it knows or does, and how many other classes of objects it affects. Control strategy decisions have a surprisingly strong influence on how functionality and information are distributed among objects.

I like to consider alternatives—I don't just go with the flow and let interaction sequences happen. I like to mull over different ways of accomplishing the same task, constantly seeking ways to construct my design that preserve encapsulation, maximize reuse and minimize the complexity of a class methods and information structure. In considering alternatives, I also

Object Transition by Design



Object Technology Potential

Object Technology can provide a company with significant benefits:

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

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

Transition Solution

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

KSC Transition Services

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

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

KSC Development Environment

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

Design your Transition

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

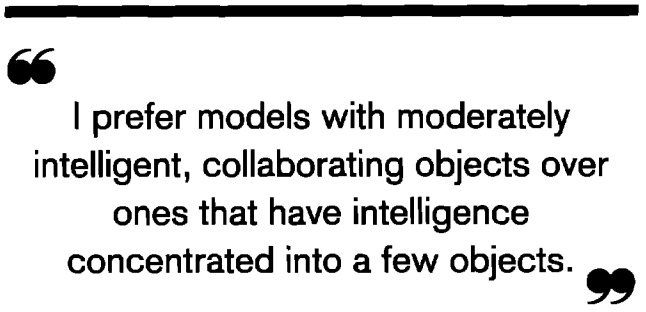
Knowledge Systems Corporation

OBJECT TRANSITION BY DESIGN

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

seek out ways to distribute control between business objects. I prefer models with moderately intelligent, collaborating objects over ones that have intelligence concentrated into a few objects.

For objects that play a more central role, I also want to decide how they are created and initialized and what happens when they finish performing their assigned tasks. These details are important so that collaborators don't become burdened with unnecessary details or complex message sequences.



I am a strong proponent of constructing simple interfaces to objects. I also like to reduce the internal complexity of any one object by spreading responsibilities among cooperating objects. As I look at alternatives, here are some questions I ponder:

- What are the consequences of distributing control among collaborators?
- Does one way require one object to know more things than it might otherwise? Are there any significant problems or issues that arise if it does?
- Does an object really need to know and retain information, or can it just pass that information along?
- What are the consequences of building intelligence into more objects? How does object intelligence shift between alternatives?
- Is there a way to exploit polymorphism?
- Is there a way to exploit inheritance?
- Is there a way to design a more general-purpose solution or more general-purpose objects? What are the costs of building a general solution? Are those costs warranted, or am I over-engineering my solution?

CONSIDERING ALTERNATIVES: AN EXAMPLE

Let's consider interactions among four objects in our Automated Teller Machine simulation: an ATM object, a concrete subclass of Financial Transaction, a Customer, and the customer's Accounts. In our simulation, there are several concrete classes of Financial Transaction, corresponding to the different financial services that the customer can select from a menu (Withdraw, Deposit, Transfer Funds, or Inquire Account Balance). As we work through the alternatives, I'll point out a few key decisions that we should make.

In our design, the ATM object has a controlling role. It controls this cycle of user interaction:

1. By collaborating with user interface classes the ATM presents a greeting message to the bank customer.
2. Once the user has entered a valid bank card and typed in the correct PIN number, the ATM is then responsible for providing a menu of financial transactions to the user.
3. The user can select and perform one or more transactions and can press the Cancel key instead of selecting a new financial transaction when finished.
4. Transactions will be logged to a history file.
5. A receipt of the transactions will be printed for the user upon completion of all transactions.

Objects in the Financial Transaction class hierarchy are service objects performing a particular financial transaction on demand. A Customer object contains data about our user that are pertinent to selecting and perform financial transactions: the customer's name and identification number, and a list of accounts. An Account object knows and maintains facts about a particular customer account, the balance being just one of those facts. Most likely, there would be different classes of Accounts (Checking, Savings, plus possibly even a richer hierarchy, depending on how accounts operate). If we design our Account class hierarchy so that all classes support the same set of messages needed by transactions, then we can use them interchangeably within transaction code.

Here are two different ways we could design an ATM to interact with a transaction:

First scenario

In this scenario, the ATM takes responsibility for collecting necessary information from the user (e.g., which account and how much) before asking the Transaction to do its job:

1. ATM determines account, amount, and all other relevant information.
2. ATM then creates an appropriate Transaction, telling it the information it needs to know.
3. ATM then tells the Transaction to perform the transaction.
4. The Transaction collaborates with Account to perform the transaction.

Second scenario

In this slight variation, the ATM delegates responsibility for collecting necessary information to the Transaction:

1. The ATM object creates a Transaction object, handing it a list of Accounts.

2. ATM then tells the Transaction to perform the transaction.

3. The Transaction first needs to determine the amount, desired account, and other pertinent information (based on the kind of transaction object it is).

4. The Transaction collaborates with an Account object to perform the transaction.

What are the trade-offs in choosing one design over the other?

In the first case, transaction code does not have to deal with collecting information from the user (such as desired account and transaction amount). Transaction need not be aware of any user interface classes or be responsible for collecting any information. We could conceive of designing transactions usable in any application involving financial transactions, not just for our ATM application. What is the downside of this scenario?

Any client using any financial transaction object needs to do more. For Balance Inquiry, Withdraw, Deposit, and Funds Transfer, slightly different pieces of information are needed for each transaction. Balance Inquiry requires that the client determine the account. For Withdraw, we need to determine an account and an amount. Deposit requires determining the account and amount. Funds Transfer requires two accounts and the amount to transfer between them.

If we leave the responsibility for determining this information with the ATM, we will end up with a fair amount of code in the ATM just to set things up. It doesn't feel quite right to write slightly different code sequences to set up each kind of transaction.

In the second alternative, we define our transaction objects to have a very simple client interface. The ATM creates the desired Transaction object, then sends it a message asking it to perform the transaction. Since the ATM has visibility of Customer, it can ask the Customer for its account list, and pass only the information needed along to the Transaction object. The code that initiates a funds transfer might look like this:
(Funds Transfer for Accounts: Customer accounts)perform Transaction.

Our challenge is to construct methods within the Transaction hierarchy that allow us to get reuse through inheritance. We need to clearly define and document the substeps of a "generic" transaction algorithm. We need to specify which substeps are replaceable by subclasses and which are not. We would probably write common methods that support subsets of performing a transaction for gathering information from the user and executing a transaction, such as locking the account and committing changes to the account.

It clearly seems appropriate for a Transaction in the second scenario to also assume responsibility for logging the transaction and printing the contents of the receipt, since it knows all the associated information. We haven't said anything about creating new objects to help Transaction accomplish that task, but that is definitely a possibility. In the first scenario, both ATM and Transaction know how much money and which accounts are involved, so who should do logging and receipt printing is much less clear.

It all comes down to deciding what awareness and involvement each object in a collaboration should have. We can design our ATM to simply cycle through menus creating transactions without much awareness of what happens within a transaction. We accomplish this by empowering Transaction with the capability for gathering information it needs from the user. This requires Transaction to collaborate with a few more objects (some user interface classes, printing, and logging services). Transaction has more responsibility for establishing the context to do its job.

This makes it harder to reuse Transaction objects designed in this fashion in another application, but it certainly is possible. We could design "smart" transactions that either could be told what information they need or, lacking this information, figure it out for themselves. It is important to work out all the ramifications of establishing the context required for an object to do its job.

My preference at this point is to work through the second alternative. It clearly establishes the ATM object with a controlling role and defines Transaction to perform a service. There are still problems with this design. Transaction has to know too much context to be portable between different financial applications—it works only in the ATM application. How can we fix this problem?

> "I am a strong proponent of constructing simple interfaces to objects."

Making the extra effort to design Transaction objects that are reusable in any financial application leads us to consider yet another design alternative. We could separate the gathering of information and performing ATM-specific functions from the actual performing of the transaction. In this third design variation, ATM Transaction objects serve as interface objects with responsibility for gathering user input, logging results, and printing receipts. The ATM Transaction object creates and collaborates with an appropriate Transaction object to execute the transaction. We have created a new class of objects, ATM Transaction, to encapsulate the ATM-specific tasks of performing transactions.

Our third and final scenario

In this slight variation, the ATM delegates responsibility for collecting necessary information to an ATM Transaction object. The ATM Transaction collaborates with a Transaction object to execute the Transaction:

1. The ATM object creates an ATM Transaction object, handing it a list of Accounts.

2. ATM then tells the ATM Transaction to perform the transaction.

3. The ATM Transaction first needs to determine the amount, desired account, and other pertinent information (based on the kind of transaction object it is).

4. The ATM Transaction creates an appropriate Transaction object.

5. The ATM Transaction then tells the Transaction the account and amount, and tells it to execute the transaction.

6. The Transaction collaborates with an Account object to perform the transaction.

It's also important to understand what kind of feedback, if any, occurs between collaborators. Is it direct or indirect, complex or simple? In the case of ATM and ATM Transaction, if we design our ATM Transaction so that it logs and prints receipt information, very little feedback is required between it and the ATM. If we give the ATM Transaction the task of printing receipt information, it needs to collaborate with Transaction to do so. It probably also has to know whether the transaction successfully completed or not before it can print the results.

Ideally, a client requires no feedback, but simply makes a request and expects the server to quietly do its job. If possible, we can shift responsibilities between clients and servers to eliminate any such requirement. If not possible, then obviously, the next simplest solution is direct response in the form of status returned from the server upon completing its task. We could design the transaction to return some indication of success or failure upon completion. This would allow the ATM Transaction to print the receipt or log the transaction without having to first ask the Transaction for further clarification.

CONCLUSION

We've focused on designing interactions between collaborators and explored impacts of shifting responsibilities between collaborators. If we take an even broader viewpoint, we can stylize interactions between subsystems of objects and factor design objects for even more general utility. Keep in mind, however, that the goal of any designer should be to construct an appropriate object model for the job. It is better to finish a design and revisit it than spend too much time looking for the "best" way. Each choice has consequences. A good designer weighs the alternatives and constructs a pragmatic solution. ■

Rebecca Wirfs-Brock is the Director of Object Technology Services at Digitalk and co-author of DESIGNING OBJECT-ORIENTED SOFTWARE. *She has over 17 years' experience designing, implementing, and managing software products, with the last nine years focused on object-oriented software. She managed the development of Tektronix Color Smalltalk and has been immersed in developing, teaching, and lecturing on object-oriented software. Comments, further insights, or wild speculations are welcomed by the author. She can be reached via email at rebecca@digitalk.com. Her U.S. mail address is Digitalk, 7585 S.W. Mohawk Drive, Tualatin, OR 97062.*

Kent Beck

Instance specific behavior: how and why

This and the next column will discuss technical and philosophical matters. The technical material covers implementing and using instance-specific behavior, the idea that you can attach methods to individual instances rather than a class. You might use it in animation, or in building a Hypercard-like system. It is not a new idea. Lisp-based object systems have had it for years, and languages like Self rely on it exclusively. It is not well known in the Smalltalk community, though, and deserves a place in the mature Smalltalker's bag of tricks.

The philosophical material illuminates the differences between Digitalk's and ParcPlace's view of good Smalltalk style. ParcPlace grew out of a research atmosphere where truth and beauty were admired. Although established in business now, ParcPlace continues to favor elegant solutions. Digitalk has always been driven by the desire to build commercial software and has often been staffed with engineers whose experience comes from other languages. Digitalk's solutions tend to be more pragmatic and the workings easier to follow operationally, even if they don't have the most elegant high-level models.

This month's column will present a pattern for choosing and using instance-specific behavior and its implementation in VisualWorks. In the next issue, I will describe its implementation in Smalltalk/V PM 2.0 and summarize the differences in philosophy revealed by the two implementations.

PATTERN

In my previous column I introduced the idea of a pattern. Before I write a pattern for instance-specific behavior, let me review. A pattern is a program transformation. It takes a program with certain attributes and makes a new program that is somehow better—more concrete, compact, reusable, maintainable, flexible, or efficient. Patterns occur at all levels of programming. Some of them are low-level, like naming arguments and variables; some are tied to a specific language or library, like patterns for using the collection classes; and some are at the level of design, describing ways of dividing behavior between objects. The patterns for instance-specific behavior are at this most far-reaching level.

Notice that I didn't say "abstract" level. Patterns always call for a concrete transformation of a program. Even if the objects are only in your head or on cards, a pattern that applies to embryonic objects will still call for you to do specific things to those objects. I have heard complaints that patterns are too vague, or connected only to a certain language. The pattern here stands as an example of how it sometimes can apply regardless of language or implementation. Instance-specific behavior is not limited to Smalltalk, and any language that provides it can use the following pattern to guide when its use is appropriate.

Each pattern has the same four parts:

- Trigger. How to recognize when the pattern applies. This often takes the form of "You have noticed. . ."
- Constraints. The (often conflicting) constraints on the solution.
- Solution. The result of applying the pattern. The insight in the pattern is largely contained in finding the right balance between the constraints.
- Transformation. How to transform a program to conform to the pattern.

Here is a pattern I have discovered for instance-specific behavior, observed in Digitalk's PARTS. I don't claim that it is the only reason for using instance specialization. If you find uses for it not covered here, please send them along.

SCRIPTABLE OBJECTS

Trigger

- You have objects that need to change their logic at runtime.
- You have added flags—symbols used as messages or blocks in instance variables to account for this variation.
- Your users want to add logic to your objects that you can't anticipate, but are not prepared to use the full Smalltalk environment.

Constraints

- Code complexity. The solution must result in less complex code than you currently have.
- Simple programming model. If you have users who are not prepared to use all of Smalltalk, the solution must be simple enough for them to understand.
- Cannot anticipate all needed behavior. The solution is not simply a matter of adding enough flags and switches. The

objects will require entirely new, unanticipated logic after they leave your hands.

- Expressive power. The solution should be as powerful as possible and ultimately as expressive as Smalltalk itself.

Solution

Make each instance specializable (see the remainder of the article for implementation details). At runtime, you or your users can change the meaning of any message without affecting other instances. If you want to affect all instances, you can, at your discretion, make it possible to change the class. The solution provides a simple programming model at the expense of expressiveness, but the flexibility of instance specialization makes up for most of the lost power. It should be possible to remove the ad hoc specialization of the original code in favor of a more uniform approach where all changes to logic are done by changing methods.

Transformation

Flags. If a method uses a boolean flag to differentiate between cases, replace it with a method that defaults to the case using the default value of the flag. For example, if you have a method like this:

```
display
    isHighlighted
        ifTrue: [self displayHighlighted]
        ifFalse: [self displayUnHighlighted]
```

where isHighlighted defaults to false, you would replace it with the contents of the displayUnHighlighted method. In the methods that set isHighlighted you have to copy the correct method into the instance. You may find that after you have done this throughout the class, you will be able to apply the pattern "Eliminate Dead Variables."

Symbols. If a method uses perform: with a symbol, isolate the perform in its own method (use the pattern "Composing Methods"), and replace it with a method that sends the default symbol as a message. Thus:

```
initialize
    listMessage := #list
getList
    model perform: listMessage
```

would become (in the class):

```
getList
    model list
```

If you wanted to default to the case where listMessage was nil, you could change getList to:

```
getList
    ^#()
```

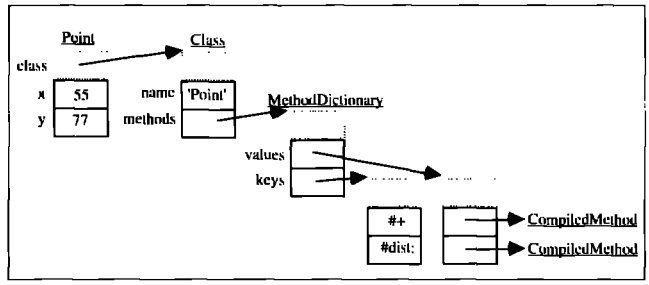


Figure 1. Objects supporting method lookup.

Any object that set the listMessage would have to instead specialize getList in the instance.

As is the case with flags, after applying the symbol transformation, the instance variable holding the symbol may no longer be needed.

Blocks. The transformation for blocks is similar to the transformation for symbols. The method in the class is the default to which the block is set. The method will have as many arguments as the block did. Thus, a block used for display:

```
initialize
    displayBlock := [:aMedium | aMedium black]
displayOn: aMedium
    displayBlock value: aMedium
```

in the class would become:

```
displayOn: aMedium
    aMedium black
```

Objects that set the block would have to specialize the instance instead. Note that this transformation will work only for blocks that use block temporary or argument variables, or instance variables of the object being specialized. Blocks used as a full closure, accessing variables in another object creating the block, generally cannot be transformed in this way.

PARCPLACE IMPLEMENTATION

Runtime structures

To understand how to implement instance specialization, you first need to understand how the current model works. As shown in Figure 1, every object has a hidden instance variable that holds its class. The class in turn has an instance variable that holds a MethodDictionary, which maps Symbols to CompiledMethods. When an object is sent a message:

1. Its class is fetched.
2. The class MethodDictionary is fetched.
3. The selector of the message is looked up in the dictionary.
4. The CompiledMethod found there is activated.

That's what happens conceptually, but there are many clever tricks to make it go faster in common cases where so much flexibility isn't needed.

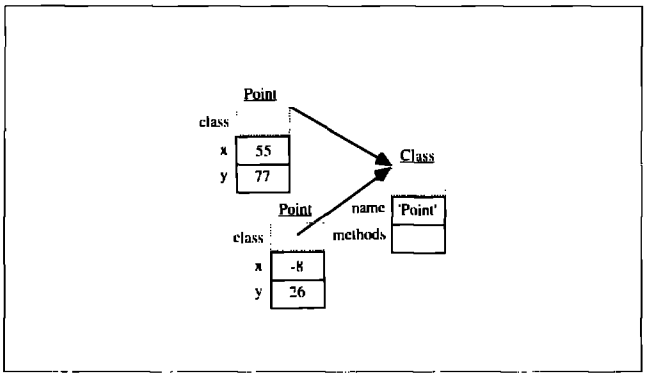


Figure 2a. A change for instance is a change for all.

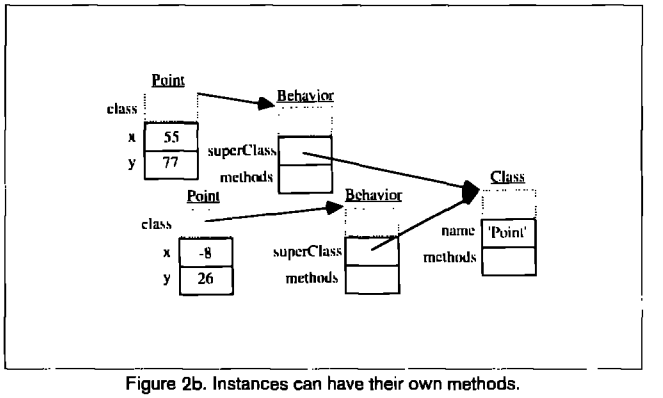


Figure 2b. Instances can have their own methods.

Conceptual model

The way you keep changes to one instance from affecting the others of its class is simple: They don't all have the same class. In Figure 1, all Points point to the same class object. To be separately specializable, they all need to point to different class objects, each of which inherits from the original class Point. That way, methods installed for one instance are installed only in that instance's personal class, not the one shared by all the other instances. Figure 2 summarizes this design.

Note that the class of the class of the instances is not Class, it is Behavior. (Isn't it grand to be working in a language that allows you to construct sentences like that and still have them mean something?) Classes are pretty heavyweight objects, so the system provides a simpler superclass, Behavior, which just has methods, subclasses, and superclasses. Unlike Classes, Behaviors are not expected to be named and put in a global dictionary, so they are able to be garbage collected when no one refers to them anymore. They do not introduce instance variables, so specializable instances implemented this way will only have private methods, but not state.

EXAMPLE

Creating instance (workspace version)

When I need to begin implementing a design like the one in Figure 2b, I always start in a workspace. After a bit of experimentation, here is the expression I came up with to create a specializable VisualPart:

```
| class instance |
class := Behavior new "Create a new Behavior"
    superclass: VisualPart; "Set its superclass"
    methodDictionary: MethodDictionary new;
        "Give it a clean MethodDictionary"
    setInstanceFormat: VisualPart format. "
        Give instances a reasonable format"
class compile: 'displayOn: aGC' notifying: nil.
        "VisualParts have to implement displayOn:"
instance := class new. "Make the specializable instance"
ScheduledWindow new "Create the window"
    component: instance; "Make the instance its component"
    open. "Open it"
instance inspect "Inspect the instance so we can compile new methods"
```

Then in the inspector I can execute expressions like:

```
self class compile: 'displayOn: aGraphicsContext
    aGraphicsContext displayString: 'Howdy' at: 100@100'
    notifying: nil
```

and refresh the window. Try inspecting self class in the instance to see that the structure built in the workspace matches the one in Figure 2b.

Working it into methods

Now that we see how to create specializable instances interactively, we need to be able to work the same concepts into permanent behavior. If all instances of a particular class are to be specializable, you can override the class message new:

```
new
    "Create a specializable instance"
    ^Behavior new
        superclass: self;
        format: self format;
        methodDictionary: MethodDictionary new;
        new
```

What if most instances are not specializable? You might only want to create the Behavior when you know the instance needs to be specialized. Here are a group of methods that implement lazy specialization:

```
specialize: aString
    "Compile aString as a method for this instance only"
    self specialize.
    self class compile: aString notifying: nil
specialize
    self isSpecialized ifTrue: [^self].
    class := Behavior new
        superclass: self class;
        format: self class format;
        methodDictionary: MethodDictionary new.
    self changeClassToThatOf: class basicNew
```

Note the strange method changeClassToThatOf:. It uses this interface, which requires us to waste an object, rather than changeClassTo: so that the primitive implementing it does not need to do complicated checks to make sure that the argument is a valid Behavior:

```
isSpecialized
    ^self class shouldBeRegistered not
```

continued on page 21

GUIs

Greg Hendley and Eric Smith

GUI-based application development: some guidelines

For this issue, we've decided to pull ourselves out of the muck of nuts-and-bolts details of GUI development in Smalltalk (however much we like to wallow in it) and talk about higher-level aspects of developing interactive applications in an object-oriented environment. In particular, we will be discussing some rules of thumb for GUI-based application development derived over the years.

The guidelines presented here have been gleaned from work on a wide variety of systems; they should be applicable not only to particular domains, but to nearly every GUI-based project. We hope you find them as useful in managing your projects as we do.

TERMS

But first a few quick definitions. Two terms we will use a great deal should be well understood. They are *domain model* (DM) and *user interface* (UI).

Domain Model

The domain model is defined by the set of classes that describe the model of the entities the user is trying to manipulate. For example, in an airline logistical system, the domain model would include classes representing aircraft, gates, fuel delivery systems, engine parts, maintenance workers, etc.

Ideally, the DM does not include any information about the UI. The DM does not know how the information it represents will be presented to the user, or how the user will manipulate that information.

Referring to two earlier columns on ICM architecture (SMALLTALK REPORT May 1992 and October 1992), the DM is identical to the Model layer in the ICM.

User Interface

The user interface is a much less cohesive entity. It consists of the sum of the code that gets the information in the DM to the user and maps the user's directives to changes in the DM. In terms of ICM application architecture, the UI is comprised of both the Interface and Control layers.

THE BASICS

Before talking about either the DM or the UI in detail, it is important to get these two efforts off on the right foot. How the division of the design efforts is established at the start of a project can have profound effects on its outcome.

Getting things started

Assume that a new project has just started, and an initial meeting about the new system's overall goals has taken place. The results are probably some initial forays into DM design (almost certainly dead wrong) and a few sketched storyboards for the UI. Neither of these will prove valuable.

First, there should be some measure of understanding among the designers and developers. A common vocabulary should come out of the process so that developers can communicate without talking at cross purposes. The most important outcome is an understanding of what the user wants to do with the system. Write that goal on the board in big letters before beginning the meeting, and make sure you can come up with a one-page summary on that subject after the meeting. Then make sure everyone has a copy.

Don't let your UI design drive your DM design

If you have proceeded as described above, your users, and probably your designers, have a much better idea of how the system will look from the outside rather than the inside. There are probably pages of sketches of screens and maybe a flash prototype.

The immediate temptation is to start driving your entire design from these storyboards. Starting from the UI sketches is a good way to generate requirements for the DM. You'll be able to come up with a minimal list of things the DM must be able to do to allow users to accomplish their tasks. However, if the DM design is derived from the UI design, then the DM design will suffer badly. It will not reflect the real-world relationships of the entities it contains. It will be more tightly coupled to the current project than if it were pursued independently, and so will be less reusable.

There is a danger in looking at a screen drawing and naming domain objects from it. Instead, look at users' mental models of what they do to find DMs. Then teach the interface to manipulate them.

Don't let your DM design drive your UI design

The reverse problem occurs if the initial effort goes into producing a complete model of the real-world system the user

wants to manipulate. From the DM's perspective, the UI is laid on top of the DM. This is sometimes as extreme as simply producing a browser for any and every object the user might want to edit. It results in an explosion of windows whose relationships to each other are, from the user's perspective at least, mysterious at best.

The UI and DM designs should proceed in parallel

The solution to both of these problems is to allow both design efforts to proceed in parallel, with each having only the required level of influence over the other. For example, if as part of UI design, a completely new task is identified, then new functions will be required of the DM. The designers of the DM would no doubt like to be apprised of this sort of change.

Similarly, if things thought to be impossible for the DM to represent are found to be tractable, then the UI designers should be informed that they may include access to them in the UI. Both sides of the design effort must be kept up to date with the requirements and capabilities of the DM.

Both UI and DM efforts must be working from the same requirements

Without this fairly minimal level of communication, the project will come to grief at an early stage. Though it seems obvious, it is worth stating that if the DM and UI efforts are not kept working on the same idea of what the user wants and what the DM can provide, grave disorder will result at the final integration. In the unlikely event users change their minds, the guiding document of what they want must be updated and members of both efforts must be given the new information.

THE DM DESIGN

Fortunately for UI designers, the basic kinds of objects they will be dealing with and the relationships between them already are often determined by the structure of the development environment. The object orientation of their design is ensured. Their primary goal is the quality of the UI according to the user.

DM designers, in comparison, start with a blank slate. They often must create the DM from the very basic classes provided by the development environment. As a result, they are free to make the resulting design highly object oriented or as much like a FORTRAN program as they wish. Unfortunately, there are more pressures toward the latter than the former.

Preserve the object orientation of your DM design

Time pressure combined with the speed at which requirements change is the greatest destroyer of good object-oriented design. DM designers must balance getting a product out on time with ensuring that the design stays clean enough to be reasonably maintainable, extendable, and reusable. Unfortunately, all of these benefits come at the expense of time.

There is as yet no panacea for this conflict of design goals. The most consistently successful compromise is to run through a prototype of the DM design and implementation as early in the project lifecycle as possible. Then heave the whole thing into the dust bin and do it right. Most of the requirements changes and design problems will be identified during the prototype. The resulting final DM design will benefit from the experience of implementing the prototype and shaking out the requirements and design changes. If the interface between the UI and DM has been suitably bounded, the impact on the UI of this midstream change of DM will be minimal.

Don't let external dependencies corrupt your DM design

Another threat to the object orientation of the DM is the requirement that it interface with non–object-oriented systems. If the interfaces to these systems are built into too high a level of the DM, their effects will be felt throughout the whole DM and even may be visible outside of it. The result will be a poor object-oriented model of the user's domain. There will also be an unacceptable degree of coupling with the external system.

The way to prevent this sort of damage is to isolate the offending, non–object-oriented system within a layer of classes that define an object-oriented model of it. Rather than providing only an interface to the function calls in the external system, expend the extra effort and build an object-oriented

model of the external system. A couple of layers of classes that hide the non–object-oriented nature of the external system will provide the rest of your DM with clean objects to work with and will keep the DM's interactions with the external system at an object-oriented level.

As a quick example, your system may be called upon to send requests out of an RS-232 port to some device to request status. You could simply wrap the system calls that send bytes down the wire. But this does not model what your DM is really doing. It is sending requests for status to a remote device. Defining objects that represent the remote device and its status would be much more useful and would hide the communications mechanism used to talk to the device.

Put off information systems interfacing as long as possible

A special case of interfacing with non–object-oriented systems is the information system. Commonly, this will be a relational database or perhaps an even more primitive data store. Often there will be an existing schema in the database for representing the DM entities. The temptation is to base the DM design on this schema. Don't do it. Ignore the schema until forced bodily to store and retrieve information using the database.

DM designs derived from database schemata often result in classes that know a great deal of information, but don't do anything. There are often no obvious candidate classes to which important behaviors may be assigned. The result is that the design ends up with a collection of very passive classes derived from the database schema and another group of classes representing processes performed using classes from the first group. The resulting design is nonintuitive and difficult to convey to anybody outside the initial design team.

The best policy is to pursue a pure, object-oriented design for the DM and put in the extra work to map this design to and from the database schema. The extra work will pay off in improved quality of the DM.

THE UI DESIGN

Since the structure of UI classes is often determined by the development environment, the degree of object orientation of the UI is already determined. However, a UI design can be very clean and object oriented, and still be terrible. We'll quickly cover a couple of the most common culprits for poor UIs in an object-oriented setting.

Provide user-centered views in your DM

This problem is related to the problem of letting the DM drive the UI design covered above. To keep the UI centered on the user and not the DM, keep the details of the DM out of any discussions of the UI. Go through the tasks as a user and treat the DM as a single big object. Later, during implementation, UI builders can allocate the various dependencies upon the DM to the objects best able to address them.

Don't over-instrument your UI

Beware of feature creep. This oft-repeated caveat still bears emphasizing. If the DM has been designed on its own, it is quite likely capable of modeling aspects of the real-world system it represents, about which the user is quite uninterested. Avoid the temptation to show the user, in a single application, every capability built into the DM. The result will look much like a 747 cockpit.

Stick to providing what was originally asked for. If the user asks for more, you can always be smug about how easy it is to add it.

KEYS TO SUCCESS

All of the above assume that you will be dividing the work of DM and UI development. Even if there is a single developer, these two tasks should be conceptually divided. To make the whole process work, the two design and development efforts must be kept reading the same script.

Continuous communication

As mentioned above, keeping the level of communication between the two design efforts is crucial to avoiding big surprises come integration time. Both sides of the development effort must be kept up to date regarding changes generated by the user, management, and each other.

This communication is so important that clear structures to support it should be put in place early in the project lifecycle. Have one or more people on both teams. Have members of both teams in the same room constantly. Anything that keeps the two efforts in contact will help.

Continuous integration

As implementation proceeds, integrate the developing UI and DM components frequently—preferably daily or more often. Putting off integration until just before project milestones will simply lead to more and bigger embarrassing surprises.

By keeping the two teams in communication and by constantly testing the fit of their work, both wheels of the project can be kept on the same track and schedule slips can be identified and planned for before they become dangerously large.

And finally...

As a last word, no matter how small the project seems to be, go through the analysis and design before delving into implementation. At the very least, you will gain important documentation of how you were thinking when you began implementation. This will ease maintenance and reuse even if it doesn't ease implementation. ■

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

THE BEST OF comp.lang.smalltalk

Alan Knight

Reflection

At its lowest level, Smalltalk is implemented by primitive operations, usually written in C or assembler. The number of primitives is surprisingly small, however, and most of Smalltalk's functions are implemented in Smalltalk. This includes many basic aspects of the compiler, the windowing system, and the language itself.

Because of this we can use Smalltalk code to examine or alter the Smalltalk system. This property is called *computational reflection* and is a very powerful, dangerous, and confusing feature.

WHAT YOU CAN DO WITH REFLECTION

In the simplest case we can use reflective capabilities to write code that reasons about other code (or itself). Examples of this are the Smalltalk debugger, inspectors, and the senders/implementors feature. All of these are written entirely in Smalltalk, which is made possible by reflection.

These capabilites are benign, since they only examine the code, but reflection also allows us to alter the system in almost any way we choose. This is where it starts to get dangerous.

Having the ability to change the system gives us an enormous amount of power. We can make drastic changes to the environment or add very sophisticated features. A favorite target, at least in Smalltalk-80, is the compiler. Several of the components we use in our lab make significant changes to it. We have the constraint engine from ThingLab II, which adds features for compiling constraints and makes them execute at reasonable speed. We have ENVY/Developer, which hooks the compiler into a database for version control and configuration management. We have VisualWorks, which also makes changes to the compiler, although I haven't figured out why it needs to.

System changes allow these products to do things that would otherwise be difficult or impossible. In the cases of ENVY and VisualWorks, the changes are so significant that the products are distributed as images rather than source code to be filed in.

WHY YOU SHOULDN'T DO IT

Unfortunately, these changes cause major problems for reuse. Smalltalk and object-oriented programming are being sold as tools for building software components that can be combined to easily build systems. Combining components that modify the compiler (or other basic system components) in incompatible ways is a nightmare. Bringing ThingLab into ENVY was not easy, even though we had one of OTI's ENVY gurus doing the hard parts. Bringing in VisualWorks appeared even more difficult. Fortunately, there are enough other people interested in VisualWorks that OTI is doing all the work for us. We just have to wait a little longer for the product. That solves our immediate problem, but having to tinker with all our components every time we bring in a new one doesn't bode well for reuse.

I'm certainly not arguing that changes to the system are always a bad thing, but they are something that should be examined carefully and avoided if there are reasonable alternatives. Just because you can change the system doesn't mean you should.

The other problem with making major changes to the system is that it's hard to do it right. Making changes to the system is as likely to cause your image to crash as adding a new feature. In fact, it's surprisingly easy to make fatal changes without even realizing it. I know I have. Often enough, their effect is delayed so that I've saved my image before realizing it's wounded.

TRAPS

There are many potential difficulties in attempting to change basic features of Smalltalk. The virtual machine may make assumptions about the structure of a few classes. This probably will not be well documented. If changes can be made, the order of changes may be critical, creating code that runs but cannot be filed in and other interesting phenomena. Getting lost in the maze of circular definitions is always dangerous.

Fortunately, systems like USENET allow us to draw on the knowledge of those who have already encountered such problems. In the remainder of this column, I print excerpts from a discussion on one particular type of system change, adding instance variables to the class Behavior. Although this thread originally started with a question about Smalltalk/V, most of the discussion concerns Smalltalk-80. That's because Smalltalk/V doesn't allow changing the definition of classes that have instances (or whose subclasses have instances). Since Class, which is a subclass of Behavior, always has instances, we can't directly change these definitions in ST/V.

Even in Smalltalk-80, which supports modifying classes with instances, it is not a trivial matter. We encounter all of the difficulties described above, and very experienced programmers can make mistakes.

ADDING INSTANCE VARIABLES TO BEHAVIOR

Most of what follows is from two very long posts. The first is from Ralph Johnson, a professor at the University of Illinois, one of whose research projects is an optimizing compiler for a statically typed Smalltalk (see September 1991 SMALLTALK REPORT). The other post is from Mario Wolczko, who maintains the Smalltalk archive at the University of Manchester. Both of these people obviously have spent a lot of time hacking the internals of Smalltalk.

Ralph Johnson (johnson@cs.uiuc.edu) begins.

> **Smalltalk-80 lets you add an instance variable to a class with instances, but you still can't add instance variables to Behavior or Class. When we tried, the system started recompiling every class and, after an hour or so, just froze. I mentioned this to Peter Deutsch, then ParcPlace chief implementor and general guru. He stared off in the distance for several minutes, and then nodded his head, "Yes, you can't do it."**
>
> **First, you can't change Behavior, and we didn't try. Behavior defines the part of a class and a metaclass that the virtual machine knows about. . . . Thus, you can only change Behavior if you change the virtual machine. (This is not exactly right; you can probably add instance variables at the back as long as you don't mess with the ones at the front.)**
>
> **It is easy to make new subclasses of Behavior, and to add instance variables to them. However, if you try to change Class then you run into a circularity, because Class ultimately defines itself. More precisely, Class class is a subclass of ClassDescription class, which is a subclass of Behavior class, which is a subclass of Object class, which is a subclass of Class. (For a very good reason, I might add. :-))**
>
> **Thus, if you change Class, you are also changing the superclass of its class, i.e., you are changing its own implementation. I haven't figured out exactly what breaks, but it would be surprising if something didn't. Welcome to the navel staring world of reflection.**
>
> **If you really, really, really want to add an instance variable to Class, it can be done. First, you have to have a program that can copy an image and write it out to disk. You then change this program so that when it writes out a Class, it adds the extra instance variables. While you are at it you modify Class so that it knows about these extra instance variables. Run this program and you have a new image that gives extra instance variables to Classes.**
>
> **I have thought long about this program, but the difficulty in getting it all to work has never seemed worth the trouble. There are so many easier ways to accomplish the same thing. The easiest is to have a dictionary indexed by the name of the class, whose value is an object that represents the extra instance variables you wish you could add. This is a little slower than real instance variables, but is otherwise nearly indistinguishable, especially if you like to access instance variables with accessing methods. You just have to write two accessing methods for each variable foo:**

```
foo
    ^(ExtraClassVariables at: self name) foo
foo: anObject
    (ExtraClassVariables at: self name) foo: anObject
```

> **and that is all it takes. If you want to be complete, make sure that deleting a class removes the entry for the class from the dictionary.**
>
> **That is usually what we do, but once we wanted to do something better. Our compiled Smalltalk lets each class define its own method lookup routine. When the compiled code finds that it needs to do method lookup, it fetches the method lookup routine from an instance variable in the class of the receiver. This technique doesn't work very well unless we can add an instance variable. Long ago, we hacked around the problem of not being able to add an instance variable to Class by adding an instance variable to a component of Behavior, i.e., to MethodDictionary. Or rather, we made a subclass of MethodDictionary and added it there. Unfortunately, ParcPlace has made MethodDictionary not subclassable. In an orgy of fiendish hackery, I subclassed set and replaced the set of subclasses in a Behavior with a SetThatHasAMethodLookupRoutine. Now the compiled code just has to do another level of indirection to find the method lookup routine, while the normal Smalltalk image thinks everything is the same as it always was.**

The unusual punctuation :-) is a USENET convention called a "smiley." It's intended to compensate for the lack of any tone-of-voice cues in text, and indicates humor or something not to be taken too seriously.

Although this post is extremely informative, its central premise is no longer valid, as Mario Wolczko (mario@cs.man.ac.uk) points out:

> **It's with trepidation that I dare to contradict both Ralph Johnson and Peter Deutsch, but here goes... The "blue book" definition of the virtual machine mentions only the first three instance variables of Behavior, namely superclass, methodDict and format, and assumes they are instance variables 1, 2 and 3. If you change the structure or position of any of these, e.g. by adding any instance variables anywhere in the list before 'format', the system will surely break. The blue book VM makes no assumptions about any other instance variables in**

Behavior; it doesn't assume the existence of **Class**, **ClassDescription**, **Metaclass**, etc., or know anything about their structure in addition to those three instance variables inherited from **Behavior**. (This may have changed with more recent versions of the VM, but I don't think so—I'll get to that shortly.)

This alone doesn't mean that you can add an instance variable. Another problem arises during the recompilation of methods. When compiled, a method references instance variables by their offsets within objects. So, if you add an instance variable to a class, all the instance methods in that class and its subclasses should be recompiled. (Actually, if you add only at the end of the list, only methods in subclasses need be recompiled, providing the subclass has its own instance variables, but the system has never taken advantage of this fact, and always recompiles everything.) Also, all existing instances have to be mutated to reflect the new structure. In older versions of the system, the mutation and recompilation were not synchronised, so when adding an instance variable to the end of **Behavior**'s list, some of the methods actually performing the recompilation and mutation got recompiled before their instances were mutated (or possibly vice versa, I forget), and the system fell over. Since the addition of **ClassBuilder** in 2.4, this doesn't seem to happen any more. Try adding an instance variable to **Behavior** after **'format'**—it should work fine in any 2.4, 2.5 or 4.x system. Whether this was by design or accident, I don't know.

An alternative solution (which I dabbled with in 2.4 days) is to implement "lazy" mutation—build a completely new class, delete all the methods from the old class's dictionary except **doesNotUnderstand** (and a few others needed for mutation), and have **doesNotUnderstand** mutate the object when it receives a message. I sweated over this for a few days and got the basics working, but then 2.5 came along...

Ralph Johnson replies:

It is certainly amusing to find out that I have been avoiding adding instance variable to Class all these years for no good reason. I don't know what versions I originally tried it on, but it was quite a few years ago. This is a classic example of superstitious behavior—to keep on doing something because it worked a particular way once. Thanks to all those who set me straight, and I'm going to go eliminate some ugly code that has apparently not been necessary for several years.

Alan Knight is currently working in V/Windows on contract for The Object People. He can be reached at 613.225.8812 or by email at knight@mrco.carleton.ca.

■ SMALLTALK IDIOMS

continued from page 15

Only classes that have a name return true for shouldBeRegistered. If we already have specialized an instance using the above algorithm, this test will be correct, while other reasons for creating unnamed classes would render it wrong.

CONCLUSION

You have seen what instance-specific behavior is, why you would choose to use it, and how to implement it in VisualWorks. In the next column I will describe how to implement it in Smalltalk/V PM 2.0, Digitalk's most technically advanced product. The differences in implementation reveal some of the differences in philosophy between the two companies as engineering organizations. These differences will be important to you as you move between systems. ■

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

Highlights

Excerpts from industry publications

IMPLEMENTATION

. . . "Among those projects I have found to be successful in object orientation, there has always been a collaboration of management (and) developers," said [Rational Inc.'s Grady Booch]. He said failures result either when management imposes object orientation on an unwilling or unprepared technology group, or conversely, where the programmer tries to implement object-oriented systems without the blessing from or understanding by management. . .

OOPSLA conference is object-oriented, David Tanaka, COMPUTER DEALER NEWS, 11/16/92

CULTURAL ISSUES

. . . [Taligent's Mark Vickers:] . . . "We have to change people's focus from having to rebuild the world from scratch; with objects, it's easier and faster to deliver new ideas and make money by leveraging on others' work. [We've] got to get the industry into a mode of being able to leverage its previous products rather than having to throw things away. . . "

Objects for end users, Carry Lu, BYTE, 12/92

CLIENT-SERVER

. . . OOP and enterprise model design are the two core building blocks upon which the next generation information systems must be built if they're to provide manufacturers the kind of flexibility required to be competitive in the 1990s. And the prevailing architecture so aptly suited to exploit the potential of these twin building blocks is client-server architecture. . .

CIM II: The integrated manufacturing enterprise, Peter F. Lopes, INDUSTRIAL ENGINEERING, 11/92

DISTRIBUTED OBJECTS

. . . Only distributed objects can talk to each other across the network through ORBs—a simple, but important, differentiation from programming-language objects, such as those created from C++. Distributed objects differ from programming-language objects in two fundamental ways: they can be in different locations (why they're called distributed) and can use ORBs. Distributed objects can be confused with programming-language objects because the distributed objects may be implemented using an OO programming language (also used to create programming-language objects) such as C++, SmallTalk, and Objective C.

Objects everywhere: Sun, Object Design work together on object-oriented file system, Shalini Chatterjee, SUNWORLD, 11/92

·CALENDAR·

March 17-19, 1993
UNIFORUM '93
San Francisco,CA
800.323.5155

April 19-23, 1993
OBJECT EXPO NY
New York, NY
212.274.0640

May 4-6, 1993
Dev Con '93
Costa Mesa, CA
800.531.2344, ext 912

●

March30-April1,1993
OBJECT TECHNOLOGY '93
Cambridge, UK
44.491.41022

●

May 3-7, 1993
DB EXPO
San Francisco, CA
415.941.8440

●

October 25-28,1993
IAKE '93
Amsterdam, The Netherlands
301.926.4243

PRODUCT ANNOUNCEMENTS

Product Announcements are not reviews. They are abstracted from press releases provided by vendors, and no endorsement is implied. Vendors interested in being included in this feature should send press releases to our editorial offfices, Product Announcements Dept., 91 Second Ave., Ottawa, Ontario K1S 2H4, Canada.

A new charting product for Objectworks\Smalltalk has been introduced by East Cliff Software. The project, EC-Charts, allows Smalltalk programmers to easily include scatter plots, bar charts, and line charts in their application windows. EC-Charts is a widget, or view; the application simply provides the numbers, and the EC-Charts widget plots them and displays the chart.

East Cliff Software, 21137 East Cliff Dr., Santa Cruz, CA 95062, 408.462.0641, fax: 408.462.0441

Digitalk has announced the availability of the PARTS COBOL Wrapper, a component for PARTS Workbench, the first technology to wrap COBOL into reusable parts. This client/server integration enables very rapid visual application construction from prefabricated software components. Digitalk also has announced the availability of the PARTS Relational Database Interface, a component for PARTS Workbench, which allows integration with relational databases.

Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045, 310.645.1082, fax: 310.645-1306

THE TOP NAME IN TRAINING IS ON THE BOTTOM OF THE BOX.

Where can you find the best in object-oriented training?

The same place you found the best in object-oriented products. At Digitalk, the creator of Smalltalk/V.

Whether you're launching a pilot project, modernizing legacy code, or developing a large scale application, nobody else can contribute such inside expertise. Training, design, consulting, prototyping, mentoring, custom engineering, and project planning. For Windows, OS/2 or Macintosh. Digitalk does it all.

ONE-STOP SHOPPING.

Only Digitalk offers you a complete solution. Including award-winning products, proven training and our arsenal of consulting services.

Which you can benefit from on-site, or at our training facilities in Oregon. Either way, you'll learn from a staff that literally wrote the book on object-oriented design (the internationally respected "Designing Object Oriented Software").

We know objects and Smalltalk/V inside out, because we've been developing real-world applications for years.

The result? You'll absorb the tips, techniques and strategies that immediately boost your productivity. You'll reduce your learning curve, and you'll meet or exceed your project expectations. All in a time frame you may now think impossible.

IMMEDIATE RESULTS.

Digitalk's training gives you practical information and techniques you can put to work immediately on your project. Just ask our clients like IBM, Bank of America, Progressive Insurance, Puget Power & Light, U.S. Sprint, plus many others. And Digitalk is one of only eight companies in IBM's International Alliance for AD/Cycle—IBM's software development strategy for the 1990's. For a full description and schedule of classes, call (800) 888-6892 x411.

Let the people who put the power in Smalltalk/V, help you get the most power out of it.

100% PURE OBJECT TRAINING. DIGITALK