

The Smalltalk Report

The International Newsletter for Smalltalk Programmers

May 1992

Volume 1 Number 7

IMPLEMENTATION OF OS/2 MULTI-THREADING SUPPORT IN SMALLTALK/V PM

By Doug Barbour

Contents:

Features/Articles

- 1 Implementation of OS/2 Multi-threading Support in Smalltalk/V PM
by Doug Barbour
- 7 SmallDraw—Release 4 Graphics and MVC, Part I
by Dan Benson

Columns

- 13 *Getting Real*: Class Instance variables for Smalltalk/V
by Juanita Ewing
- 16 *The Best of Camp.Lang.Smalltalk*: More frequently asked questions
by Alan Knight
- 19 *GUIs*: Separating the GUI from the application
by Greg Hendley and Eric Smith
- 23 *Smalltalk Idioms*: Why study Smalltalk idioms?
by Kent Beck

Departments

- 25 *Product Announcements*
- 26 *What They're Saying About Smalltalk*

Smalltalk/V PM (hereafter referred to as VPM) is an excellent OS/2 application development environment that provides most of the needed facilities (especially in release 1.3). When combined with a third-party window editing product, such as WindowBuilder from Cooper and Peters, applications can be created easily without the OS/2 Toolkit. In a multitasking system such as OS/2, however, complex applications frequently need to execute multiple tasks concurrently using system facilities. This article explores the implementation of VPM and discusses how the non-Smalltalk parts of an application can communicate with the Smalltalk parts, even when the non-Smalltalk parts are running in their own threads.

INTERFACING VPM TO OTHER PRODUCTS

One common requirement is an interface between application programming interfaces (APIs) provided by third parties and the VPM environment. Typically, these APIs are packaged as one or more dynamic link libraries. The most obvious way to provide the interface is to follow the VPM developer's guide and create a subclass of `DynamicLinkLibrary` (DLL). While this works for DLL calls that return control rapidly, it fails when control is kept for any length of time. The effect of this failure is to "hang" the workstation until the DLL call returns. To see this for yourself, evaluate the following with "Do it":

```
DosLibrary sleep: 10000
```

This will cause your entire workstation to hang for the 10 seconds it takes `DosSleep` to return control. Clearly, if you can't guarantee the DLL call will return control quickly, another implementation must be found. This causes particular problems for communications packages, since the time it takes to return control depends on the network and the partner program! In fact, all of the research for this article was done while developing an APPC interface for VPM.

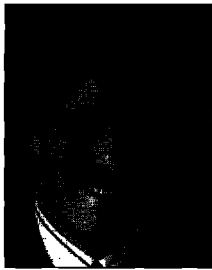
SOLVING THE INTERFACING PROBLEM

To an OS/2 C programmer, the solution to this problem would be simple: Create another thread of execution, and let it wait for the response from the DLL call. In the VPM environment, however, life is a little more complicated since it does not support OS/2 threads directly. An additional OS/2 thread is the best way to solve this problem. The only thing to be worked out is the means of communicating between VPM and the thread. OS/2 provides many ways to accomplish multi-thread communication, using PM messages is the easiest and most robust way because PM messages are already used extensively in VPM. To understand how to do this, some background information is needed.

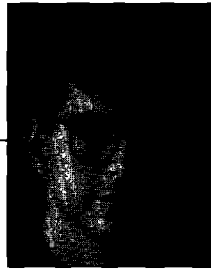
VPM IMPLEMENTATION

VPM provides support for multitasking via the `Process`, `ProcessScheduler`, and `Semaphore` classes. As long as all of the sub-tasks of the application are strictly Smalltalk code, this method works quite well. Most VPM application developers don't care if this multiprocessing is simulated and does not actually use OS/2's thread capability. This fact becomes

continued on page 3...



John Pugh



Paul White

EDITORS' CORNER

In the olden days, one of Smalltalk's most notorious drawbacks was its inflexibility with respect to interfacing with software developed in other languages. Application Program Interfaces (APIs) offer a solution to this problem, allowing developers to interface easily with existing DLLs, regardless of the language in which they were developed. However, as Doug Barbour points out in our lead article this month, it is still not a perfect solution. If Smalltalk is using a single thread processor, it in effect locks up while waiting for a return from an API call. Doug describes a technique for using existing PM messages to let Smalltalk/V PM communicate with OS/2 threads, thus providing Smalltalk programmers with more control over their applications' behavior.

This month marks the debut of a series of articles on Objectworks\Smalltalk by Dan Benson from the University of Washington. In Release 4, ParcPlace made many fundamental changes to the architecture of the graphics and user interface classes. As David Liebs and Kenny Rubin, described in last month's issue, the changes were necessary and overdue. Unfortunately, the in-depth explanations and good examples needed for programmers to comprehend the changes were sadly lacking. Over the coming months, Dan will develop a simple graphics application he calls SmallDraw. His first article introduces graphics concepts and application construction with the MVC architecture through the definition of a "minimal" SmallDraw. This simplified version demonstrates interactive creation of geometric shapes and display of graphics. Future articles will extend the functionality of SmallDraw by adding selection, translation, scaling, alignment (using a DialogView), and grouping of objects; vertical and horizontal scrolling of the view; a cut/copy/paste clipboard; and support for command keys.

It is our pleasure to welcome another well-known Smalltalk guru, Kent Beck, to the ranks of SMALLTALK REPORT columnists. Kent has been involved in many successful Smalltalk projects and is the co-inventor, with Ward Cunningham, of the popular Class-Responsibility-Collaborator (CRC) methodology for kick-starting the design of object-oriented systems. Kent's column will identify Smalltalk idioms: the "coding patterns" or "mechanisms" used frequently by experienced Smalltalk programmers in well-defined situations, but seldom written down or explained anywhere. If you have ever looked at existing Smalltalk code and wondered, Why is it done this way? Kent may have the answer.

Experienced Smalltalk programmers are well aware of the virtues of separating the model from the user interface when developing GUI applications. In this month's GUI column Greg Hendley and Eric Smith propose using a three-layer approach (Interface-Control-Model, or ICM) that further separates the UI component into an interface layer and a control layer. Simplistically, you can think of the interface layer as the code that would normally be generated by a window builder, and the control layer as the code that would respond to user interface interactions. Why separate interface from control? As Greg and Eric point out, one compelling reason is the speed with which systems can be ported across different hosts. The interface component largely isolates the code specific to the host GUI from the portable control component. As a result, only the interface component need be ported. They illustrate the approach by building a simple log-on dialog.

In her last column, Juanita Ewing described the differences between class variables and class instance variables, and lamented the fact that Smalltalk/V does not provide class instance variables. This month, she provides an implementation of such a facility for /V. Finally, Alan Knight provides us with his monthly round-up of the Smalltalk bulletin boards.

—The Editors

The Smalltalk Report

Editors

John Pugh and Paul White
Carnegie University & The Object People

SIGS PUBLICATIONS

Advisory Board

Tom Atwood, Object Technology
 Andy Brown, Inteltek
 George Acworth, Digitek
 Fred Long, American Gas Consulting
 Glenn Cook, The Volkswagen Group
 John DeWitt, ParcPlace Systems
 Tom Lane, Cambridge
 Bruce Johnson, IBM
 Peter Fagan, ParcPlace Systems
 Chuck Brown, General Electric
 V. Srinivasan, Sun Microsystems
 Robert Greenblatt, AT&T Bell Labs
 Dale Schaefer, Object Technology

The SMALLTALK REPORT

Editorial Board

John DeWitt, ParcPlace Systems
 Andy Brown, Inteltek
 Robert Phillips, Knowledge Systems Corp.
 John DeWitt, ParcPlace Systems
 Dale Schaefer, Object Technology International

Contributors

John DeWitt, ParcPlace Systems
 Glenn Cook, The Volkswagen Group
 Greg Hendley, Knowledge Systems Corp.
 Ed Smith
 Robert Phillips, Object Technology
 Eric Smith, Knowledge Systems Corp.
 Alan Knight, Digitek

SIGS Publications Group, Inc.

John DeWitt, ParcPlace Systems
 Glenn Cook, The Volkswagen Group
 Greg Hendley, Knowledge Systems Corp.
 Ed Smith
 Robert Phillips, Object Technology
 Eric Smith, Knowledge Systems Corp.
 Alan Knight, Digitek

Contributors

John DeWitt, ParcPlace Systems
 Glenn Cook, The Volkswagen Group
 Greg Hendley, Knowledge Systems Corp.
 Ed Smith
 Robert Phillips, Object Technology
 Eric Smith, Knowledge Systems Corp.
 Alan Knight, Digitek

John DeWitt, ParcPlace Systems
 Glenn Cook, The Volkswagen Group
 Greg Hendley, Knowledge Systems Corp.
 Ed Smith
 Robert Phillips, Object Technology
 Eric Smith, Knowledge Systems Corp.
 Alan Knight, Digitek



Smalltalk Report is published by SIGS Publications Group, Inc., 588 Broadway, New York, NY 10012 (212)274-0640. Copyright 1992 by SIGS Publications, Inc. 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 publishers. 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.

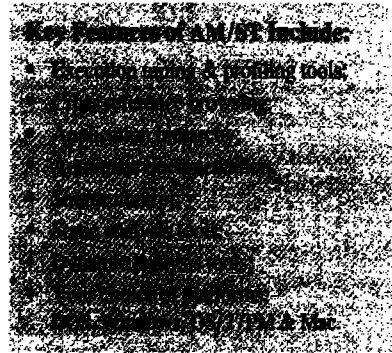
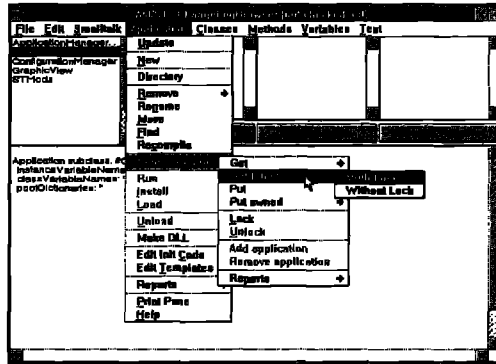
The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by SIGS Publications Group, 588 Broadway, New York, NY 10012 (212)274-0640. © Copyright 1992 by SIGS Publications, Inc. All rights reserved. Reproduction 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 publishers. 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.

ImageSoft

AM/ST™

AM/ST, developed by the SoftPert Systems Division of Coopers & Lybrand, enables the developer to manage large, complex, object-oriented applications. The AM/ST Application Browser provides multiple views of a developer's application.

AM/ST defines Smalltalk/V applications as logical groupings of classes and methods which can be managed in source files independent of the Smalltalk/V image. An application can be locked and modified by one developer, enabling other developers to browse the source code. The source code control system manages multiple revisions easily.



The original and still premier application manager for Smalltalk/V™

ChangeBrowser. As an additional tool available for Smalltalk/V PM and Smalltalk/V Windows, ChangeBrowser supports browsing of the Smalltalk/V change log file or any file in Smalltalk/V chunk format.

The addition of AM/ST to the ImageSoft Family of software development tools enhances and solidifies ImageSoft's position as — "The World's Leading Publisher of Object-Oriented Software Development Tools."



1-800/245-8840

ImageSoft™
The World's Leading Publisher of Development Tools

All trademarks are the property of their respective owners. ImageSoft, Inc., 2 Haven Avenue, Port Washington, NY 11050 516/767-2233; Fax 516/767-9067; UUCP address: mcduhup!image!info

continued from page 1...

critical, however, when interfacing VPM to other products that will be called from a lower-level language such as C.

The first thing to understand is how VPM uses OS/2 threads. In VPM 1.3, two OS/2 threads execute when the environment is running: a Presentation Manager (PM) message processing thread and a Smalltalk code executor thread. This design is based on a PM requirement that an application return control to it quickly after processing a message. Since a PM message might (and usually does) cause Smalltalk code to be executed, this PM requirement could not be guaranteed using a single OS/2 thread.

In the two-thread implementation, a PM message is processed by adding it to a global OrderedCollection named `CurrentEvents` by the PM message processing thread. This thread immediately returns control to PM, allowing other applications to process their PM messages. Some (typically very short) time later, the Smalltalk code executor thread checks the `CurrentEvents` collection to see if there are any messages. If any are pending, they are routed to their respective window objects. Class `NotificationManager` performs this service for the code executor thread. See its instance method `#run` for more details.

Since PM messages are identified by a unique message number, VPM must have a way to translate between message

numbers and method names. This translation is done by using two global objects, `PMEvents` and `PMEventsExtra`. `PMEvents` is an array of symbols, indexed by PM message number. That is, `(PMEvents at: 7)` contains the value `#wmSize:with:`. Seven is the message number assigned to the `WM_SIZE` message by PM. The `PMEvents` array is not large enough to map every possible

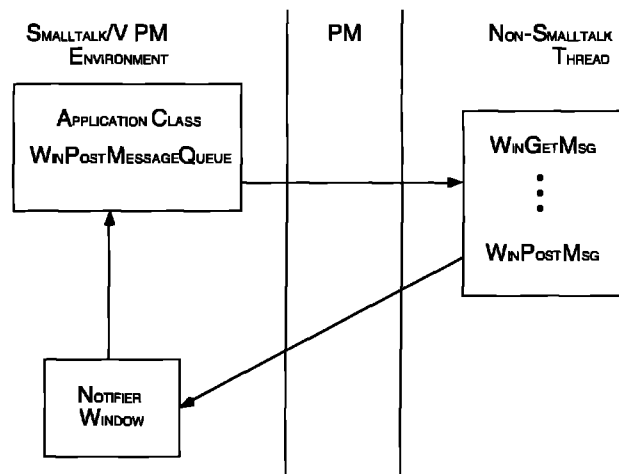


Figure 1. VPM communication with PM through additional OS/2 thread.

Listing 1. Creating OS/2 threads.

```
#define FUNCTYPE pascal far _loads

#define VMAPPC_THREAD_QUEUE 29503
#define VMAPPC_STOP_THREAD 29504

/* Function prototypes */
SHORT FUNCTYPE CreateThread(BYTE *stack, USHORT stackSize,
    HWND hwnd);
VOID FUNCTYPE Thread();

/* Global variables */
PBYTE stkbot;
SHORT rc;
TID threadID=0;
HWND notifyHwnd;
HAB hab;
HMQ hmq;

/* Individual Functions */

SHORT FUNCTYPE CreateThread(BYTE *stack, USHORT stackSize,
    HWND hwnd)
{
    notifyHwnd = hwnd;
    stkbot = &stack[ stackSize - 1 ];

    rc = DosCreateThread( (PFNTHREAD)Thread, &threadID, (VOID
        FAR *)stkbot );

    return( rc );
}

VOID FUNCTYPE Thread()
{
    QMSG qmsg;

    hab = WinInitialize( NULL ); /* Initialize PM */

    hmq = WinCreateMsgQueue( hab, 0 ); /* Create application
        msg queue */

    WinPostMsg( notifyHwnd, VMAPPC_THREAD_QUEUE,
        MPFROMLONG(hmq), 0L );

    WinGetMsg( hab, &qmsg, (HWND)NULL, 0, 0 );

    while( qmsg.msg != VMAPPC_STOP_THREAD )
    {
        .
        .
        .code to process each message...
        .
        .

        WinGetMsg( hab, &qmsg, (HWND)NULL, 0, 0 );
    }

    WinDestroyMsgQueue( hmq );
    WinTerminate( hab );
}

```

Listing 2. The AppDLL class.

```
DynamicLinkLibrary variableByteSubclass: #AppDLL
classVariableNames: "
    poolDictionaries: " !

!AppDLL methods !

createThread: stackAddress stackSize: stackSize notifyHwnd:
    notifyHwnd
    <api: 'CREATETHREAD' along ushort handle ushort>
    ^self invalidArgument!
```

PM message. Due to memory considerations, only about the first 478 are mapped here. Any message numbers received that are greater than the size of PMEvents are looked up in the global Dictionary PMEventsExtra. The PM message number (558, for example) is the key for this dictionary, and the method name symbol is its value (#hmmerror:with:). Once the PM message has been mapped to a method name via either PMEvents or PMEventsExtra, the PM message processing thread sends a Smalltalk message to the appropriate window object using the two arguments provided by PM. For more details look at any of the #wm... instance methods in the Window class.

EXTENSION OF THE VPM PM INTERFACE MODEL

The existing VPM message interface with PM is extended to support communications with the additional thread. This communication is carried out by having the thread create a PM message queue and passing its handle to the VPM environment. This handle is then used with the WinPostQueueMsg PM call to post messages to the thread. One of the message parameters should be the handle of the notifier window that is to receive notification when the request is complete. The thread performs whatever is requested, and posts a PM message to the notifier window. See Figure 1 for a diagram of this interaction.

The first issue to be addressed is the selection of message numbers for communications with the thread. A consecutive block of message numbers should be chosen for each application, avoiding conflicts. Message numbers must be unique within one VPM image. User-defined message numbers must be greater than 4,096 to avoid conflicts with PM messages. For the APPC interface, message numbers range from 29,500 to 29,505. Each message number must have an entry in PMEventsExtra that specifies the method name associated with it. For example, message number 29,501 is associated with the #vpmAppcVerbDone:with: method. The message definitions used in the APPC interface are:

```
PMEventsExtra
```

Listing 3. Creating a subclass of auxiliary window to handle PM messages.

<pre> InvisibleNotifierWindow subclass: #AppcNotifierWindow instanceVariableNames: 'vcbAddress threadQueue' classVariableNames: '' poolDictionaries: '' ! !InvisibleNotifierWindow methods! threadQueueEvent ----- Description: A message has been received. Cause the associated event to occur. Role: Private Parameters: none Returns: self " ^self event: #threadQueue! verbDone ----- Description: A message has been received. Cause the associated event to occur. Role: </pre>	<pre> Private Parameters: none Returns: self " ^self event: #verbDone! vpmAppcThreadQueue: mp1 with: mp2 ----- Description: A PM message has been received. Remember the parameter values, add the message to CurrentEvents, and return to PM as soon as possible. Role: Private Parameters: mp1 - PMLong first parameter from sender mp2 - PMLong second parameter from sender Requires: Instance Variables: threadQueue Returns: self " threadQueue := mp1 deepCopy. </pre>	<pre> ^self sendInputEvent: #threadQueueEvent! vpmAppcVerbDone: mp1 with: mp2 ----- Description: A PM message has been received. Remember the parameter values, add the message to CurrentEvents, and return to PM as soon as possible. Role: Private Parameters: mp1 - PMLong first parameter from sender mp2 - PMLong second parameter from sender Requires: Instance Variables: vcbAddress Returns: self " vcbAddress := mp1 deepCopy. ^self sendInputEvent: #verbDone! ! </pre>
--	--	---

Listing 4. An example of application class.

<pre> AsyncNotifier subclass: #AppcConversation instanceVariableNames: 'appcLibrary notifierWindow threadStack threadQueue sem' classVariableNames: '' poolDictionaries: 'AppcReturnCodes AppcConstants CharacterConstants' ! !AppcConversation methods ! initialize appcLibrary := AppcDLL open. notifierWindow := (AppcNotifierWindow new open; when: #threadQueue perform: #threadQueue: ; when: #verbDone perform: #verbDone: ; yourself). sem := Semaphore new. threadStack := PMAAddress copyToNonSmalltalkMemory: (ByteArray new: 4096). "Create the external thread" </pre>	<pre> appcLibrary createThread: threadStack asParameter stackSize: 4096 notifyHwnd: notifierWindow handle asParameter. "Fork a process to receive messages from the thread" [CurrentProcess makeUserIF. Notifier run] fork. "Wait for the #vpmAppcThreadQueue:with: message" sem wait. "Re-establish the main process as the receiver of PM messages" CurrentProcess makeUserIF.! threadQueue: anAppcNotifierWindow "Remember the thread's queue handle" threadQueue := anAppcNotifierWindow threadQueue. "Signal the main process that the message has been received" sem signal. "Terminate the message receiver process" Processor terminateActive.! </pre>
---	---

at: 29501 put: #vpmAppcVerbDone:with:;
at: 29503 put: #vpmAppcThreadQueue:with:.

To create an additional thread from the VPM environment, a DLL must be created to issue the `DosCreateThread` OS/2 call and to contain the code to be executed by the thread. In Listing 1, the `CreateThread` function performs this action. Parameters to `CreateThread` are the stack area to be used by the new thread, the size of the stack area, and the window handle to be notified when the thread is created and ready for work. The stack is passed to the thread from the VPM environment to ensure that it is properly freed after thread termination. The code in the `Thread` function initializes PM and creates a message queue. It then posts the `VPM_APPC_THREAD_QUEUE` message to the `AppcNotifierWindow` instance, passing the queue handle as a message parameter. The `Thread` function then loops until a `VPM_APPC_STOP_THREAD` is received, processing messages. After each message is processed, a `VPM_APPC_VERB_DONE` message is posted to the `AppcNotifierWindow` instance. After the `VPM_APPC_STOP_THREAD` message is received, the `Thread` function stops looping and posts the `VPM_APPC_THREAD_STOPPED` message, which allows us to be absolutely sure the thread has stopped before freeing the stack area.

A subclass of `DynamicLinkLibrary` must be created to allow the `CreateThread` function to be called. The `AppcDLL` class is shown in Listing 2.

To communicate with the thread using PM messages, there must be a PM window to receive them. Since this window will not perform any other functions, it should not be visible. This may be accomplished by making it a subclass of `DDEAuxWindow`. To facilitate reuse, an abstract superclass named `InvisibleNotifierWindow` has been created as a subclass of `DDEAuxWindow`. Notifier windows for various functions are subclasses of this class. `InvisibleNotifierWindow` implements the same `#when:perform:` interface as `SubPane`, allowing notifier windows to send messages to their owners when important events occur. Listing 3 shows some of the methods defined for the `AppcNotifierWindow`. As you can see, there is a method corresponding to each PM message number defined in `PMEventsExtra`. Companion methods are also defined for each. The `#vpmAppc...` methods are executed on behalf of the PM message processing thread. They copy the message parameters to instance variables if necessary and add the message event to `CurrentEvents` via the `#sendInputEvent:` method. The return from a `#vpmAppc...` method causes VPM to return control to PM, allowing other applications to perform window operations.

A REAL EXAMPLE

Let's trace through a complete interaction sequence between VPM and the thread. The interaction of interest is the thread notifying VPM that it has created its message queue and passing the queue handle as a message parameter. This is implemented in the `initialize` instance method of the `AppcConversation` class and is shown in Listing 4. Here are the steps:

- The instance of `AppcDLL` is created.
- The instance of `AppcNotifierWindow` is created and the `#threadQueue:` event is registered.
- A VPM Semaphore instance is created to allow waiting until the thread has been created.
- The thread's stack area is allocated as an instance of `PMAddress`.
- The thread is started by sending the `#createThread:stack-Size:notifyHwnd` message to the `AppcDLL` instance.
- A VPM process is forked to wait for the thread to respond with the `VPM_APPC_THREAD_QUEUE` message.
- The main VPM process waits for the semaphore to be signaled, thus causing the application to wait without disturbing PM operations.
- When the thread posts the `VPM_APPC_THREAD_QUEUE` message, it is processed by the forked process, which causes the `AppcNotifierWindow` instance to be sent a `#vpmAppcThreadQueue:with:` message.
- The `#vpmAppcThreadQueue:with:` method makes a copy of the thread queue passed by the thread, adds a `#threadQueueEvent` to `CurrentEvents`, and returns. At this point, VPM returns control to PM to allow other applications to perform windowing operations.
- The forked process removes the `#threadQueueEvent` from `CurrentEvents` and sends it to the `AppcNotifierWindow` instance.
- The `#threadQueueEvent` sends itself the `#event:` message with `#threadQueue` as the argument.
- The `AppcNotifierWindow` instance sends the registered method for the `#threadQueue` event (`#threadQueue:`) to the owner (the `AppcConversation` instance) with itself as the argument.
- The `#threadQueue:` method assigns the thread queue to an instance variable and signals the semaphore. This allows the waiting main process to continue.
- The forked process is terminated by the `#threadQueueEvent` method.

SUMMARY

Using the concepts explored in this article, interfaces may be written from VPM to any long-running application without adversely affecting other applications running in the system. ■

Doug Barbour is an Information Systems Engineer at Duke Power Company. He is also a partner at Barbour Enterprises, specializing in custom interfaces to Smalltalk/V PM as well as general purpose classes. Doug may be reached at Barbour Enterprises, 1058 D. Kelly Circle, Clover SC 29710, or by phone at 803.222.1363.

SMALLDRAW—

RELEASE 4

GRAPHICS AND

MVC

By Dan Benson

Graphics often play a big role in Smalltalk applications, but it is also one of the more difficult areas to grasp. This is particularly true of the current version of Objectworks for Smalltalk-80 Release 4. Even experienced users familiar with version 2.5 can be just as confused as newcomers to Smalltalk because of the major changes in the way graphics is handled.

One reason for this confusion is the lack of adequate in-depth explanations in the users' manuals that come with Objectworks. You will be pleased to know that this will soon be remedied by new chapters scheduled for the next release from ParcPlace. Another source of confusion is that there are few examples from which to learn. This series will describe a simple graphics application that might be instructive for those trying to get a better grip on Release 4 graphics and the MVC architecture.

What better way to experiment with rendering graphics than to build a structured graphics editor? Commercial drawing programs are now commonplace, and many are quite sophisticated. We'll borrow some ideas from these applications, but will keep things simple. Since we're using Smalltalk, let's call our application SmallDraw.

We'll start off with a "minimal" version of SmallDraw. We'll limit its capabilities to adding new graphic objects to the drawing and displaying them in a window. Additional features and functionality will be developed in future articles. The focus is primarily on rendering graphics on screen, so SmallDraw won't include such functions as saving drawings to files or printing to a printer.

GRAPHIC OBJECTS

SmallDraw is a structured graphics editor as opposed to a bitmap editor. In other words, each figure drawn in the win-

dow is treated as a separate object. Bitmap editors, on the other hand, work at the pixel level. We can think of the main entity of the application as a "drawing" that contains several graphic shapes. For simplicity, we'll limit the set of two-dimensional objects to: LineSegment, Rectangle, Polyline, Polygon, and Ellipse.

Each graphic object will have attributes that describe its shape. For instance, a LineSegment can be described by a start point and an end point, and a Polyline can be described by an ordered set of points. All objects will have an inside color (or none, in which case it will appear to be "hollow"), a border color, and a line width. A graphic object's behavior should include methods for accessing and modifying its attributes and for displaying itself at a given scale.

The following class hierarchy defines the graphic objects used in SmallDraw. I've prefixed each class name with SD to distinguish it as a SmallDraw object and to avoid any naming conflicts when filed in to an image:

```
Object ()
  SDGraphicObject ('insideColor' 'borderColor' 'lineWidth')
    SDLineSegment ('start' 'end')
    SDPolyline ('vertices')
    SDPolygon ()
    SDQuadrangle ()
    SDEllipse ()
```

The common attributes of all graphic objects are collected in the superclass SDGraphicObject. SDPolygon is identical to SDPolyline except its boundary is closed. Rather than use Smalltalk's Rectangle class, I defined a separate class with a more general name, SDQuadrangle, which inherits from SDPolyline, but is constrained to four vertices and 90 degree angles. Representing a rectangle by an origin point and a corner point assumes it is always aligned with the x-y axis. A more general representation allows rectangles to be oriented at any angle. I've defined SDEllipse as a subclass of SDQuadrangle more for convenience of representation than semantics since an ellipse can be represented by its bounding rectangle and, except for display, it acts just like a quadrangle.

DISPLAYING GRAPHIC OBJECTS

In Release 4, graphics are displayed on two-dimensional graphic media, all of which are subclasses of DisplaySurface. With the three types of display surfaces (Window, Pixmap, and Mask), there are two approaches one can take: display graphics directly on-screen using a Window, or display graphics off-screen using a Pixmap or Mask before final on-screen presentation. Off-screen rendering can result in smoother looking updates and eliminate "flashing," but is a bit more involved, so we'll stick with on-screen rendering for SmallDraw.

From an object-oriented point of view, it seems reasonable to have the graphic objects display themselves since they should know best how to do that. However, we should not expect these objects to know Bresenham's line drawing algo-

rithm or how to turn on a red pixel, etc. The object that deals with these primitive operations is an instance of `GraphicsContext`. Each instance of a display surface has its own graphics context that keeps track of parameters such as line width, color, font, clipping rectangle, etc. The graphics context object acts as the intermediary between objects to be displayed and display surfaces. It maintains its local origin for its display surface and can display everything from images, to text, to geometric shapes.

The `GraphicsContext` instance methods for displaying geometric shapes include line segments, polylines, filled polygons, rectangular borders, filled rectangles, arcs, and wedges (filled arcs). It is through these methods that the `SmallDraw` graphic objects are able to display themselves. In `SmallDraw`, when an object is asked to display itself, it is sent an instance of a graphics context along with a `Point` that specifies the scale to be used. Where applicable, each type of object draws its interior, if it has an inside color, and then its border, if it has a border color. It will first tell the `GraphicsContext` what color to use and then what shape to draw. `SDPolygon`, for instance, does the following:

```
displayOn: aGraphicsContext scale: aScalePoint
| pts |
pts := self vertices collect: [:p | p * aScalePoint].
self insideColor isNil
  iffFalse: [aGraphicsContext
            paint: self insideColor;
            displayPolygon: pts].
self borderColor isNil
  iffFalse: [aGraphicsContext
            paint: self borderColor;
            lineWidth: self lineWidth;
            displayPolyline: pts]
```

Note that the `displayPolygon:` method displays a filled polygon defined by the set of points, whereas the `displayPolyline:` method displays only the boundary. `GraphicsContext` has similar methods for circular shapes as seen in the display method for `SDEllipse`:

```
displayOn: aGraphicsContext scale: aScalePoint
| bb |
bb := self boundingBox scaleBy: aScalePoint.

self insideColor isNil
  iffFalse: [aGraphicsContext
            paint: self insideColor;
            displayWedgeBoundedBy: bb startAngle: 0
            sweepAngle: 360 at: 0@0].
self borderColor isNil
  iffFalse: [aGraphicsContext
            paint: self borderColor;
            lineWidth: self lineWidth;
            displayArcBoundedBy: bb startAngle: 0
            sweepAngle: 360 at: 0@0]
```

THE MVC ARCHITECTURE

`Smalltalk` applications are constructed using the Model-View-Controller (MVC) architecture. A brief overview of the MVC paradigm will help in putting together our application. The process begins by dividing the application into two parts: the information model (the part that manages data storage and processing) and the user interface (the part that handles input and output). The user interface is divided further into `View`, which is responsible for visual output, and `Controller`, which is responsible for user input such as from the mouse or keyboard.

Separating the user interface components from the information model makes it easier to “plug-in” other interfaces to the same model, connect multiple interfaces to a single model, or reuse interfaces for other models. The `Smalltalk` system is full of examples of using the same kinds of interface components for different sorts of models. For instance, take a look at the `SystemBrowser` or the `FileList` utility.

A model can have any number of views, whereas views are usually attached to a single model. Each view has only one controller, or none if it doesn't require user input, and each controller has only one view. While the model is indirectly connected to its view through its dependents' instance variable, the view and controller have instance variables tying them directly to each other and their model as shown in Figure 1.

The model communicates with its interface components through a dependency mechanism. Each model maintains a list of its dependents and notifies them whenever changes are made to the model's state. These dependents can be any kind of object, but are usually one or more views. A view keeps an eye out for certain changes in its model. When it detects any of those changes, it will update itself by displaying certain aspects of its model.

Whenever the model changes its state (in a way that may be of significance), it sends itself one of the following messages:

```
self changed
self changed: aSymbol
self changed: aSymbol with: anArgument
```

Which message is used depends on how much detail is necessary. For instance, the `changed` message is the most general and simply informs the dependents that the model has changed in some way, but it doesn't tell them which aspect of the model has changed. Some views might only be interested in a particular aspect of the model, so additional information can determine whether or not they will respond to the change.

The `changed` methods are found in the `Model` instance methods. Each `changed` message is eventually transformed into an appropriate `update` message that is broadcast to the list of dependents. The dependents must have a corresponding `update` method in which they redisplay themselves or take some other action:


```

update: aSymbol with: anArgument from: aModel
update: aSymbol with: anArgument
update: aSymbol
update

```

Upon receiving an `update` message, a view will usually redisplay all, or a portion, of its model. For instance, in our SmallDraw application, when we add a new graphic object to a drawing, the view should be notified so that it will redisplay the drawing. Our model would do something like the following:

```

addObject: anObject
... code to add anObject ...
self changed: #add

```

Our view would be set up to look for that particular aspect (`#add`), invalidating itself to be refreshed when found while ignoring all other aspects:

```

update: anAspect
#add = anAspect
ifTrue: [self invalidate]

```

Controllers come into play whenever input comes from the mouse or keyboard. A controller usually takes control of input events whenever the cursor is in its view. It can be set up to check for keyboard events or mouse clicks, taking action when appropriate. Some controllers know about menus and how to process them for their respective views. A controller dealing with a menu will often direct the menu selection to itself, its view, or its model.

The MVC notification mechanism is automatically inherited when we define our application components to be subclasses of Model, View, and Controller. Let's begin our application by describing the model where the information of interest is stored and processed. In this case, the model will be the SmallDraw object itself. The primary information it will keep track of is the set of graphic objects that are drawn. Additionally, it will maintain a current inside color, border color, and line width that each new object will be assigned. These values will be initialized when the application is invoked but will be able to be modified through menu selections. The SmallDraw model will need to provide methods for adding new objects to the drawing and for accessing and modifying its other attributes.

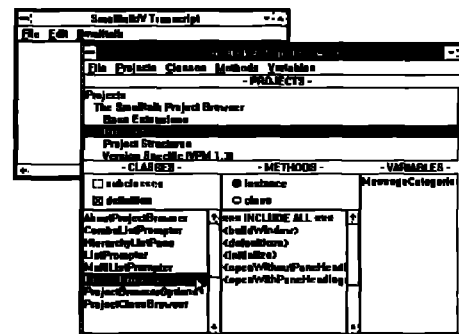
The Smalltalk system classes do not have a ready-made graphic view that will do everything that we want, so we'll define our own SmallDrawView as a subclass of View. The SmallDrawView is responsible for displaying the graphic objects of its model, an instance of SmallDraw. We'll give the SmallDrawView the ability to change the scale of the scene so that we'll be able to "zoom-in" or "zoom-out." It therefore will need to keep track of its current scale and provide some manner of changing the scale. When the SmallDrawView displays itself it will obtain the set of objects to be displayed from its model. Then, the view will ask each object to display itself using the view's

The Smalltalk Project Browser

Source Code Management System for Smalltalk/V

The Smalltalk Project Browser, from Empower Software, allows developers to track and manage changes to the Smalltalk image, and enables simplified code sharing and software project organization. Designed as a logical extension of the Smalltalk environment, this system defines hierarchical projects as collections of classes, methods, and global variables, upon which various operations can be performed.

Two tools are included: The Project Browser is used to define and maintain projects (or generate from image changes), file projects in and out, create object libraries, and generate documentation (summary, encyclopedia of classes); The Project Class Browser is an enhanced Class Hierarchy Browser which supports tracking of class and method changes for a project, and adds several productivity enhancements.



Version 1.0 supports Smalltalk/V PM & Windows, and is available now for \$99.95, + \$5 shipping (\$20 outside the U.S.; in Ca. please add 8.25% sales tax). Includes source code, object libraries (PM), and user manual.



Empower Software
 279 S. Beverly Drive, Suite #217
 Beverly Hills, Ca. 90212
 Voice: (213) 878-2327 CIS: 71031,2640

graphics context, scaled by the view's display scale. As an aside, if we wanted to, we could design our application window to contain two SmallDrawViews, with one at normal size and the other enlarged or reduced. Each would be a dependent of the same model, but would provide different perspectives of that model.

We'll also need to define our own SmallDrawController. It will be responsible for handling all user interaction from the mouse and keyboard. The two main functions of the controller in this application are handling menus and drawing new objects. If we make it a subclass of ControllerWithMenu, it will inherit the ability to handle menus. When the operate button is pressed, the controller usually obtains the menu from its view and processes it. For simplicity's sake, we'll choose the type of object to draw from a menu. Since the controller handles the drawing of new objects, we'll have it add its own menu selections to its view's menu before processing. By contrast, most commercial applications provide a palette of drawing-tool buttons for selecting the type of shape to draw.

So far, here is the hierarchy of SmallDraw MVC application classes we've described:

```

Object ()
  Model ('dependents')
    SmallDraw ('objects' 'insideColor' 'borderColor' 'lineWidth')
  VisualComponent ()
    VisualPart ('container')
      DependentPart ('model')
        View ('controller')

```

```

SmallDrawView ('scale')
Controller ('model' 'view' 'sensor')
    ControllerWithMenu ()
        SmallDrawController ()

```

MVC INTERACTION

Because views and controllers are often designed to work closely together, views often specify, and even create, their own controllers. When the SmallDrawView needs to create its controller, it will ask its **defaultControllerClass** for one. All we need to do is provide a SmallDrawView instance method that specifies the correct controller class:

```

defaultControllerClass
^SmallDrawController

```

Connecting our MVC triad together is accomplished by specifying the model when the view is created:

```

aSmallDrawView := SmallDrawView model: SmallDraw new.

```

From this single message the view knows its model and the model's dependents include the view. When the view creates its controller, the connections are completed.

It may be interesting to look at the message selector interactions between our MVC components. We can see that the only interaction initiated from the model is in the update notification mechanism. Other than that, there are four methods that the view and controller rely on from their model. If we had a completely different model, but one whose object interface included the same four selectors as SmallDraw (and notified dependents with #add), we could use the same view and controller with no changes.

Responsibilities in the application are distributed among the MVC components. As such, each component has something to offer the user by means of the menu. The SmallDraw model presents the user with options to change the default inside color, border color, and line width. The SmallDrawView allows the user to change the scale of the displayed objects, and the SmallDrawController offers a choice of objects to be drawn. Each has its own independent menu, but we need a way of combining them into a single menu for presentation on screen and a way to determine who will process the resulting menu selection.

The operate menu is activated by pressing the operate button or by clicking in the menu bar. The SmallDrawController senses this and asks itself for its menu. The SmallDrawController **menu** method asks its view for its menu. If one is returned, a new menu is constructed combining the controller's menu and its view's menu. In a similar fashion, the SmallDrawView **menu** method asks its model for a menu, and answers the resulting combination of the two menus.

Determining the responsible party of a menu selection requires the controller and view to keep track of the selectors each responds to. When the SmallDrawController obtains the

user's menu selection, it asks itself whether the menu selection is one of its local menu items. If so, it responds to the selection. If not, it asks its view the same thing. If so, the view is asked to perform the selection, otherwise the model is asked to perform the selection.

RUBBER BANDING

A common technique for drawing shapes interactively on screen is to use "rubber banding." Through rubber banding, figures appear to be stretched into shape as the cursor moves across the screen. For this to happen effectively, the rubber band lines must be alternately drawn and erased in rapid succession without damaging the existing contents of the screen. There are different ways to accomplish this in Smalltalk, the simplest is to use the following Screen instance method:

```

displayShape: shape lineWidth: lineWidth at: aPoint
forMilliseconds: milliseconds

```

where:

```

shape - Array of points defining the rubber band line(s)
lineWidth - width of line(s) to be drawn
aPoint - origin for shape in screen coordinates
milliseconds - length of delay before erasing

```

Depending on the length of delay used, the lines can appear to shimmer as they are quickly drawn and erased, indicating their temporary or dynamic status. From my experience, this technique is very effective on a Macintosh but, depending on the background color of the window, it can be difficult to see on the IBM RS/6000 platform. I don't know the quality of its visual appearance on other platforms. I've found that a line width of 1 and a 25 millisecond delay works well on a Macintosh IIx. Since several methods make use of rubber banding, and it may be necessary to modify these parameters for different platforms, the following SmallDrawController instance methods are defined:

```

rubberBandLineWidth
^1

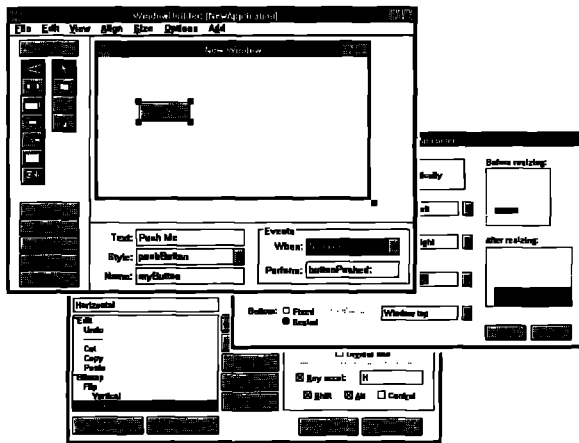
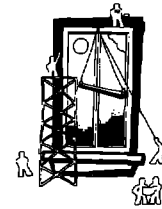
rubberBandDelay
^25

```

Let's take a look at a very common rubber banding shape, the rectangle. For now, this shape will be used in drawing SDQuadrangles and SDEllipses, but eventually we'll use it for selecting groups of objects in the window. For this reason, we define it as a separate SmallDrawController instance method called **rectangleFromScreen**. Rectangles will be drawn on screen by pressing the mouse button, which defines one corner of the rectangle, dragging the mouse across the screen, and releasing the mouse button at the opposite corner of the rectangle. When the method is entered, it will be assumed that the mouse button has just been pressed. The method's job is to rubber band a rectangular shape from that corner point to the cursor point as it is being dragged across the screen. The

WINDOWBUILDER

The Interface Builder for Smalltalk/V



"... this is a potent rapid application development tool which should be included in any Smalltalk/V developer's environment."

- Jim Salmons, *The Smalltalk Report*, September 1991

The key to a good application is its user interface, and the key to good interfaces is a powerful user interface development tool.

For Smalltalk, that tool is WindowBuilder.

Instead of tediously hand coding window definitions and rummaging through manuals, you'll simply "draw" your windows, and WindowBuilder will generate the code for you. Don't worry — you won't be locked into that first, inevitably less-than-perfect design; WindowBuilder allows you to revise your windows incrementally. Nor will you be forced to learn a new paradigm; WindowBuilder generates standard Smalltalk code, and fits as seamlessly into the Smalltalk environment as the class hierarchy browser or the debugger.

Until March 31st, WindowBuilder/V PM will be available at an introductory price of \$295, \$100 off the list price of \$395. WindowBuilder/V Windows sells for \$149.95. Both include an unconditional 60 day guarantee.

For a free brochure, call us at (415) 855-9036, or send us a fax at (415) 855-9856. You'll be glad you did!

COOPER & PETERS, INC. [FORMERLY ACUMEN SOFTWARE]

2600 EL CAMINO REAL, SUITE 609 PALO ALTO, CALIFORNIA 94306

PHONE 415 855 9036 FAX 415 855 9856 COMPUSERVE 71571,407

method will return the resulting rectangle scaled to the view's coordinate system:

rectangleFromScreen

"Answer the resulting rectangle obtained from the user in the view's coordinate system. Assume the mouse is already pressed."

```
| origin rectangle polygon screen lastPoint start newPoint |
screen := Screen default.
```

```
start := lastPoint := self sensor cursorPoint.
```

```
origin := self sensor globalOrigin.
```

```
rectangle := Rectangle origin: start corner: lastPoint.
```

```
polygon := Array new: 5 withAll: start.
```

```
[self sensor anyButtonPressed]
```

```
whileTrue:
```

```
  [screen
```

```
    displayShape: polygon
```

```
    lineWidth: self rubberBandLineWidth
```

```
    at: origin
```

```
    forMilliseconds: self rubberBandDelay.
```

```
  (newPoint := self sensor cursorPoint) = lastPoint
```

```
  ifFalse:
```

```
    [rectangle := Rectangle vertex: start vertex:
```

```
      (lastPoint := newPoint).
```

```
    polygon
```

```
      at: 1 put: rectangle topLeft;
```

```
      at: 2 put: rectangle topRight;
```

```
      at: 3 put: rectangle bottomRight;
```

```
      at: 4 put: rectangle bottomLeft;
```

```
      at: 5 put: rectangle topLeft]].
```

```
^rectangle scaleBy: self view displayScale reciprocal
```

The rectangle is rubber banded as long as the mouse button is pressed, however, the shape only needs to be updated whenever the mouse moves. Creating the rectangle with the **vertex:vertex:** method allows the rectangle on screen to be stretched in any direction from the initial corner point. This is demonstrated below showing four different snapshots of rubber banded rectangles superimposed on each other:

You may notice that the Screen instance method used for rubber banding applies to the entire screen and is not clipped to the controller's view. This can be useful for animating objects being "dragged" between windows or for drawing objects that extend beyond a view's bounds, but it can also possibly cause some confusion or give an impression of inconsistency since most other commercial applications do not behave that way.

PUTTING IT ALL TOGETHER

In order to get our application to appear on the screen in its own window we use an instance of ScheduledWindow. The "Scheduled" part of the name indicates that its instances are scheduled with ScheduledControllers, the control manager. With Release 4, ScheduledWindows take on the "look-and-feel" of the host windowing system of the specific platform on which it runs. This applies to only the outer portions of the window, such as the title bar, close, and zoom boxes. The interior of the window maintains the "Objectworks Smalltalk" look and will appear the same across all platforms. Creating a ScheduledWindow and giving it a label that will appear in its title bar is straightforward:

```
aWindow := ScheduledWindow new.
aWindow label: 'SmallDraw'.
```

Now that we have a window that knows how to render itself on the screen, we can fill it with our `SmallDrawView`. However, we must first place a `Wrapper` around the view. `Wrappers` add decoration to the views they contain such as color, borders, layout, scroll bars, and menu bars. There are several types of `Wrappers`, beginning with the “plain brown” `Wrapper` (no frills) to the most decorative, `EdgeWidgetWrapper` (all the frills). We’ll use the `EdgeWidgetWrapper` so that we can have a menu bar, but we’ll need to turn off the vertical scroll bar as each `EdgeWidgetWrapper` comes with a vertical scroll bar by default:

```
aWrappedView := (EdgeWidgetWrapper on: aSmallDrawView)
noVerticalScrollBar.
```

We can now place our wrapped view in the window by specifying it as a component of the `ScheduledWindow`:

```
aWindow component: aWrappedView.
```

Our application will be invoked by first creating an instance of `SmallDraw` and then asking it to “open” itself using the following `SmallDraw` instance method:

```
open
| aWindow aSmallDrawView aWrappedView |
aWindow := ScheduledWindow new.

aWindow label: 'SmallDraw'.
aSmallDrawView := SmallDrawView model: self.
aWrappedView := (EdgeWidgetWrapper on: aSmallDrawView)
noVerticalScrollBar.
aWindow component: aWrappedView.
aWindow openWithExtent: 200@200
```

This, of course, could be simplified to a single statement:

```
open
(ScheduledWindow new)
label: 'SmallDraw';
component: (EdgeWidgetWrapper on:
(SmallDrawView model: self)) noVerticalScrollBar;
openWithExtent: 200@200
```

Now, to start the `SmallDraw` application we simply do the following:

```
SmallDraw new open
```

Incidentally, if we wanted our application window to have more than one subview, we would need to place our views in an instance of `CompositePart`, which would then be made the component of our window. `CompositeParts` can contain any number of views or other `CompositeParts`. Relative and absolute placement of subviews within composite parts is specified through `Layouts`. The following `SmallDraw` instance method opens a window containing two `SmallDrawViews` each occupying half of the window vertically:

```
openWithTwoViews
| window composite |
window := (ScheduledWindow new) label: 'SmallDraw'.
composite := CompositePart new.
window component: composite.
composite
“The left hand view.”
add: (EdgeWidgetWrapper on:
(SmallDrawView model: self)) noVerticalScrollBar
in: (LayoutFrame new
leftFraction: 0;
rightFraction: 0.5;
topFraction: 0;
bottomFraction: 1);

“The right hand view.”
add: (EdgeWidgetWrapper on:
(SmallDrawView model: self)) noVerticalScrollBar
in: (LayoutFrame new
leftFraction: 0.5;

rightFraction: 1;
topFraction: 0;
bottomFraction: 1).
window openWithExtent: 200@200
```


CONCLUSION

In this article, I’ve presented a structured-graphics editor, `SmallDraw`, albeit a minimal version. I admit it’s not much of an editor yet, since the user can merely draw objects in the window. Future articles in the series will extend the functionality of `SmallDraw`. We’ll add selection, translation, scaling, alignment (using a `DialogView`), and grouping of objects, vertical and horizontal scrolling of the view, a cut/copy/paste clipboard, and support for command keys. ■

Dan Benson is a Ph.D. candidate in the Department of Electrical Engineering at the University of Washington where he is developing a 3-D spatial database for human anatomy using `Smalltalk` and the GemStone OODBMS. He may be contacted at: Department of Electrical Engineering, FT-10, University of Washington, Seattle, WA 98195, by phone at 206.685.7567, or email: benson@ee.washington.edu.

Universal Database OBJECT BRIDGE™

This developer's tool allows Smalltalk to read and write to:
ORACLE, INGRES, SYBASE, SQL/DS, DB2, RDB, RDBCDD,
dBASEIII, Lotus, and Excel.



Intelligent Systems, Inc.

506 N. State Street, Ann Arbor, MI 48104 (313) 996-4238 (313) 996-4241 fax

Class instance variables for Smalltalk/V

In my last column, I described the effect of class variables and class instance variables on class reusability and concluded that classes implemented with class instance variables are more reusable than classes implemented with class variables. Smalltalk-80-derived versions of Smalltalk have class instance variables, but Smalltalk/V versions do not. This column contains the code to add class instance variables to Smalltalk/V Windows.

All objects in Smalltalk/V have instance variables, even class objects. The code in this column just makes the facility apparent for classes and allows users to define new class instance variables.

HOW TO DEFINE CLASS INSTANCE VARIABLES

Ordinarily, users see a class definition in a browser like the example in Listing 1. After the code from this column is added to an image, users will see an extended class definition in the browser. The extended class definition consists of two messages, one to the class and one to the metaclass. Listing 2 is an example of an extended class definition with no class instance variables. The message argument to the metaclass is an empty string.

Adding a class instance variable is just like adding an instance variable. The user modifies the argument to the message `instanceVariableNames:`. The argument is a string containing names of class instance variables. Then the user saves the class definition through the menu. The system redefines the class and recompiles as needed. Listing 3 is an extended class definition with a class instance variable named `defaultDirection`.

IMPLEMENTING CLASS INSTANCE VARIABLES

The code to add class instance variables to Smalltalk/V Windows consists of five methods, four of which are funda-

mental and one that is a modification to the class hierarchy browser. A complete listing of the code is included at the end of this column.

Other versions of Smalltalk/V have different implementations, and a different version of the code is necessary to implement class instance variables.

```
MetaClass class subclassOf: aClass
Modified
```

This is the instance creation method for MetaClass and is a private method. It is modified so new instances of metaclass have the structure of their superclass. In the original version of this method, each metaclass was created with the structure of the Class class.

```
MetaClass methods instanceVariableNames: stringOfInstVarNames
New
```

This is a new method representing the public interface for class instance variables. This method is used to redefine the instance variables for a class (class instance variables). The argument to this method is a string containing names of class instance variables. The argument is the same format as for instance variables and class variables.

```
Class fileOutOn: aStream
Modified
```

This method has been modified to write the definition for class instance variables. The result of this method is also used to print the definition of a class in the browser. The string-defining class instance variable always prints even if there are

Listing 1. Class Definition for AnimatedObject.

```
Object subclass: #AnimatedObject
instanceVariableNames:
'position oldPosition jumpIncrement direction ... goCount'
classVariableNames: ''
poolDictionaries: ''
'WinConstants'
```

Listing 2. Extended Class Definition for AnimatedObject.

```
Object subclass: #AnimatedObject
instanceVariableNames:
'position oldPosition jumpIncrement direction ... goCount'
classVariableNames: ''
poolDictionaries: ''
'WinConstants'
AnimatedObject class instanceVariableNames: ''
```

silence
a collection of tools for project management and code delivery

- full multi-user project management
- source code version control
- automatic change documenting
- release packaging
- ship compiled code without source
- reconfigurable installation tool
- change log browser and restorer
- code performance profiling

\$99.95

introduitory pricing
 • on Windows version until March 31st, 1992
 • on OS/2 version until May 31st, 1992

digamma solutions
 Unit 6, 387 Spadina Avenue, Toronto, Ontario, Canada, M5T 2G6
 Phone: (416) 551-5833 Fax: (416) 409-2850

no class instance variables. This is necessary because the evaluation of a class definition in the browser must return the same result.

Class recreate: numberOfExtraFields
New

This new private method is used to recreate the class object when the number of class instance variables has changed. It deals with a number of implementation details, such as storing the new class in the Smalltalk dictionary and the global variable TableOfClasses, and inserting the new class into the class inheritance hierarchy.

ClassHierarchyBrowser acceptClass: aString from: aPane
Modified

Listing 3. Extended Class Definition for AnimatedObject with a Class Instance Variable.

```

Object subclass: #AnimatedObject
instanceVariableNames:
    'position oldPosition jumpIncrement direction ... goCount '
classVariableNames: "
poolDictionaries:
    'WinConstants '
AnimatedObject class instanceVariableNames: 'defaultDirection'
    
```

This method has been modified to update the reference to the selected class after saving a new definition of a class. If the number of class instance variables has changed, then a new class object will be created and the browser needs to be updated. This is a private method.

COMPLETE LISTING

MetaClass class
subclassOf: aClass

"Private - Answer a new metaclass that is a subclass of the metaclass for aClass."

```

| newMeta |
newMeta := self new.
newMeta
    assignClassHash;
    structure: aClass class structure;
    superclass:
        (aClass == Class
         iffTrue: [Class]
          iffFalse: [aClass class]);
    methodDictionaries:
        (Array with: (MethodDictionary newSize: 2)) ,
        newMeta superclass methodDictionaries.
^newMeta
    
```

MetaClass

instanceVariableNames: stringOfInstVarNames

"Define (or redefine) the set of class instance variables for the class which is an instance of this metaClass. The number of class instance variable may be increased only if there are no existing instances of the class."

```

| theClass oldSize newSize aStream theClassName |
theClass := self instanceClass.
theClassName := theClass symbol.
oldSize := self instVarNames size.
newSize := stringOfInstVarNames asArrayOfSubstrings size.
oldSize < newSize
    iffTrue:
        [" if the size of the class object needs to increase
         there must be no instances"
         theClass withAllSubclasses do:
             [:aClass | aClass allInstances notEmpty
                 iffTrue: [^self error: 'Has instances']]].
self instVarNames: stringOfInstVarNames.
oldSize < newSize
    iffTrue:
        [theClass recreate: newSize-oldSize
         "recreate the class object"].
theClass := Smalltalk at: theClassName.
aStream := WriteStream on: (String new: 64).
theClass fileOutOn: aStream.
Smalltalk logSource: aStream contents forClass: theClass.
self compileAll.
self allSubclasses do:
    [:aClass | aClass compileAll].
^theClass
    
```

Class

fileOutOn: aStream

"Append the extended class definition message for the receiver to aStream. Include the statement for the definition of class instance variables."

```

| aString |
aStream cr;
nextPutAll: self superclass printString; space;
nextPutAll: self kindOfSubclass; space;
nextPutAll: name storeString; cr; space; space.
self isBits
iffalse:
  [aStream nextPutAll: 'instanceVariableNames: '.
  (aString := self instanceVariableString) isEmpty
  iffalse: [aStream cr; nextPutAll: ''].
  aStream
  nextPutAll: aString storeString;
  cr; space; space].
aStream nextPutAll: 'classVariableNames: '.
(aString := self classVariableString) isEmpty
iffalse: [aStream cr; nextPutAll: ''].
aStream
nextPutAll: aString storeString;
cr; space; space;
nextPutAll: 'poolDictionaries: '.
(aString := self sharedVariableString) isEmpty
iffalse:[ aStream cr; nextPutAll: ''].
aStream nextPutAll: aString storeString.

"Include class instance variable definition."
aString := self class instanceVariableString.
aStream nextPut: $.; cr.
aStream nextPutAll: self class name.
aStream nextPutAll: 'instanceVariableNames: '.
aStream nextPutAll: aString storeString.
aStream cr; space; space

```

Class

recreate: numberOfExtraFields

"Private - Replace this class object with an identical object with additional fields for class instance variables."

```

| newInstance mySuperclass myName oldId |
myName := self symbol.
newInstance := self class basicNew.
oldId := self id.
1 to: self class instSize - numberOfExtraFields
do:
  [:i |
  newInstance instVarAt: i put: (self instVarAt: i)].
mySuperclass := self superclass.
mySuperclass removeSubclass: self.
mySuperclass addSubclass: newInstance.
Smalltalk at: myName put: newInstance.
newInstance methodDictionary do:
  [:m |
  m classField = self
  ifTrue: [m classField: newInstance]].
newInstance subclasses copy do:
  [:sub |
  sub superclass: newInstance.
  sub recreate: numberOfExtraFields].
TableOfClasses at: oldId + 1 put: newInstance.
newInstance id: oldId.
self become: DeletedClass

```

ClassHierarchyBrowser

acceptClass: aString from: aPane

"Private - Accept aString as an updated class specification and compile it. Notify aPane if the compiler detects errors."

```

| result isClass |
result := Compiler
evaluate: aString
in: nil class
to: nil
notifying: aPane
iffail: [^true].
Smalltalk logEvaluate: aString.
isClass := result isKindOf: Class.
isClass
  ifTrue: [selectedClass := result].
self changed: #instanceVars:.
^isClass not

```

Juanita Ewing is a senior staff member of Instantiations Inc., a software engineering and consulting firm that specializes in developing and applying object-oriented software projects, and is an expert in the design and implementation of object-oriented applications, frameworks, and systems. In her previous position at Tektronix Inc., she was responsible for the development of class libraries for the first commercial-quality Smalltalk-80 system. Her professional activities include Workshop and Panel Chairs for the OOPSLA conference.

VOSS

Virtual Object Storage System for Smalltalk/V

Seamless persistent object management with update transaction control directly in the Smalltalk language

- Transparent access to Smalltalk objects on disk
- Transaction commit/rollback
- Access to individual elements of virtual collections and dictionaries
- Multi-key and multi-value virtual dictionaries with query by key range and set intersection
- Class restructure editor for renaming classes and adding or removing instance variables allows incremental application development
- Shared access to named virtual object spaces
- Source code supplied

Some comments we have received about VOSS:

"...clean ...elegant. Works like a charm."

-Hal Hildebrand, Anamet Laboratories

"Works absolutely beautifully; excellent performance and applicability."

-Raul Duran, Microgenics Instruments

**logic
ARTS**

VOSS/286 \$595 (\$375 to end of February 1992) + \$15 shipping.
VOSS/Windows \$750 (\$475 to end of February 1992) + \$15 shipping.
Quantity discounts available. Visa, MasterCard and EuroCard accepted.
Logic Arts Ltd. 75 Hemingford Road, Cambridge, England, CB1 3BY
TEL: +44 223 212392 FAX: +44 223 245171

More frequently asked questions

This month I continue describing and trying to answer some of the frequently asked Smalltalk questions posted on USENET. I must be off to a bad start, because much of this column is taken up with additions and changes from last month's information. I guess it's the price we pay for working in an area of rapid change. First change: I said there was no official list of frequently asked questions. One has now been established, and I'll be incorporating information from it. The list is maintained by Craig R. Latta (latta@con.berkeley.edu), and is available for ftp from [xcf.berkeley.edu](ftp://xcf.berkeley.edu). Many thanks to Craig for taking on the maintainer's job.

MORE FREE STUFF

Last issue I listed some sources of freely available Smalltalk code. With a few exceptions, most of that code comes in the form of small "goodies." These can be system enhancements, bug fixes, or utilities, but are seldom large enough to be called applications. This is only natural. Many people are willing to freely contribute their own small fixes and favorite enhancements to the community. It takes a much greater commitment to contribute a large project. Generally, large projects come from university research projects.

LARGER-SCALE APPLICATIONS

The first of these is the T-Gen parser generator package, written by Justin Graver (graver@ufl.edu) at the University of Florida. The package is described as "a general-purpose object-oriented tool for the automatic generation of string-to-object translators" and is available by ftp from [bikini.cis.ufl.edu](ftp://bikini.cis.ufl.edu).

The second item is a group of packages by Stephen T. Pope and others, related to the Smallmusic project. This is "a project to discuss and develop an object-oriented system for music." There is an electronic mailing list for discussions of the project, and several implementations and documents. If you have access to Internet electronic mail, you can join the mailing list by sending a request to smallmusic-request@xcf.berkeley.edu. The implementations and documents are available by ftp from [ccrma-ftp.stanford.edu](ftp://ccrma-ftp.stanford.edu). The most recent implementation is called MODE (not to be confused with the MoDE user interface toolkit, available from the University of Illinois archive at [st.cs.uiuc.edu](ftp://st.cs.uiuc.edu)). MODE

runs under ParcPlace Smalltalk Release 4.0 and combines the functions of several earlier applications running under ParcPlace Smalltalk 2.X.

If you're used to commercial software, it pays to be aware of a few differences when dealing with "free" software. First, there are no guarantees and no toll-free customer support hotlines. The code may be badly written, poorly documented, or nonportable. You may be able to get someone (possibly even the author) to help you with any problems, but you may not. I haven't personally used any of the packages described above, so don't take this mention as an endorsement of any kind.

Second, there may be restrictions on how the code may be used. Many authors retain copyright on the packages. Using all or part of the package in a commercial product may require an arrangement with the author or may not be allowed at all. Carefully read anything concerning copyright or licensing agreements. Large packages are more likely to reserve rights than goodies. The authors of T-Gen and the Smallmusic packages retain the copyright on the software.

MANCHESTER GOODIES BY FTP

One of the archives I mentioned last issue is at the University of Manchester in England. Recently, this archive became accessible by ftp. The machine is called [mushroom.cs.man.ac.uk](ftp://mushroom.cs.man.ac.uk) (an alias for 130.88.13.70) and the files are available in the directory `pub/goodies`. This machine and the archive server at the University of Illinois should now contain exactly the same material. Questions about the archive can be addressed to lib.manager@cs.man.ac.uk.

SMALLTALK CHAT SESSION

Internet Relay Chat, or IRC, is a real-time computer conferencing program. It allows people with direct access to the Internet to conduct conversations without the delays of electronic mail. Anyone on the Internet can participate, regardless of location. The drawback is the requirement for a direct network connection, meaning that the number of people capable of participating is much lower than in electronic mail or USENET forums.

On March 3, Martin Brown (mjb@netcom.com) organized an IRC conference of Smalltalk users. Even though it started at 8:30 p.m. Pacific Time, making it awkward for eastern

North America and ridiculous for Europe, the conference attracted quite a few participants.

IRC runs almost exclusively on UNIX machines, so almost all of the participants were Smalltalk-80 users. A number of people from ParcPlace participated, including VP Engineering Richard Dellinger and CEO Adele Goldberg. Highlights included a preview of features in the next ParcPlace release (in beta test as I write) and the opportunity to give feedback on features we'd like to see in future versions. It's hoped that this will be the first of many such conferences.

If you'd like to participate in these conferences, you will need an account on a machine with a direct Internet connection and a copy of the IRC program. Contact your system administrator for details.

LANGUAGE WARS

Like so much about computers, languages are a religious issue. I won't say that this is especially bad for OOP languages, but it's certainly no better than average. Many of the people on the net are reasonable, unprejudiced, and willing to accept differences of opinion. Unfortunately, a lot of them aren't, and they seem to be the ones who enjoy long debates on the relative merits of different languages.

Which language they are attacking or defending doesn't make too much difference. Each community has its own points of snobbery and defensiveness. Smalltalk advocates like to talk about pure OOP languages and about being one of the original sources of OOP. They become very defensive when anyone calls their language "slow" or "academic." C++ advocates like to talk about running fast and being the most popular language. They get defensive when their language is called "impure" or "a hack." Eiffel advocates like to talk about software engineering principles in a pure OOPL. They get defensive about being called "obscure" or "proprietary." There is some substance under the rhetoric, mostly concentrated in a few basic issues.

MULTIPLE INHERITANCE

Many current OOP languages make extensive use of multiple inheritance. Users of these languages tend to consider it an important part of OOP, and naturally ask why Smalltalk is so backward as to not support it. From the perspective of many in the Smalltalk community, multiple inheritance has been tried and judged more trouble than it's worth. Besides, if you really want it, you can always write it yourself. For example, someone named Terry (terry@galaxia.newport.ri.us) writes:

I would like to open a discussion about multiple inheritance in Smalltalk. To begin with, could someone who knows the history explain why Smalltalk does not have m.i.? Here are my suggestions for multiple inheritance....

Ralph Johnson (johnson@cs.uiuc.edu) replies with a summary of the history:

The folks at Tektronix claimed to have fixed lots of bugs, but they still kept running into problems, and finally decided that it wasn't worth it. Implementing m.i. this way can be done entirely in the image. You don't need to know anything about how the v.m. is implemented. So, anybody out there who wants to implement m.i. can just go ahead. If you are successful and can make something that people want to use then you will be famous, though probably not rich!

STRONG/STATIC TYPING

Another significant difference of opinion on programming languages is the matter of static vs. dynamic typing. In fact,

“
Many of the people on the net are reasonable, unprejudiced, and willing to accept differences of opinion. Unfortunately, a lot of them aren't.”

the existence of this argument represents one of the most remarkable achievements of C++. Almost overnight it turned more C programmers than I would have believed possible into resolute defenders of strong typing. They, along with Eiffel programmers, will argue that strong static typing is essential for good software engineering. The argument asks What if your air traffic control system pops up a "does not understand" dialogue in the middle of a forced landing? They also argue that strong typing enforces better design principles. Almost as an afterthought, they add that it makes programs run faster.

On the other hand, Smalltalk, Objective-C, and other programmers argue that (at least with current technology) static typing systems either excessively restrict what programs can be written or else don't really eliminate the possibility of run-time type errors. Further, the errors that it catches are mostly those that would be trivially detected in testing. Better to worry about what happens when your air-traffic control system divides by zero. Further, they argue, the flexibility that you lose with static typing inhibits reuse and makes programming harder.

In a lot of languages, static typing is also intimately bound up with multiple inheritance. In languages like C++ and Eiffel, variables can have values of different types, but only if all of those types are subclasses of the declared type of that vari-

SIXGRAPH™ Smalltalk/V users: the tool for maximum productivity

- Put related classes and methods into a single task-oriented object called application.
- Browse what the application sees, yet easily move code between it and external environment.
- Automatically document code via modifiable templates.
- Keep a history of previous versions; restore them with a few keystrokes.
- View class hierarchy as graph or list.
- Print applications, classes, and methods in a formatted report, paginated and commented.
- File code into applications and merge them together.
- Applications are unaffected by compress log change and many other features..

CodeIMAGER™ V286, VMac \$129.95
VWindow & VPM \$249.95
 Shipping & handling: \$13 mail, \$20 UPS, per copy
 Diskette: 3^{1/2} 5^{3/4}

SixGraph™ Computing Ltd.
 formerly ZUNIQ DATA Corp.
 2035 Côte de Liesse, suite 201
 Montreal, Que. Canada H4N 2M5
 Tél: (514) 332-1331, Fax: (514) 956-1032
CodeIMAGER is a reg. trademark of SixGraph Computing Ltd.
 Smalltalk/V is a reg. trademark of Digital, Inc.

Plan ahead for the largest object-oriented conference and exhibition on the East Coast in 1992.

Object

EXPO
 THE NATIONAL CONFERENCE AND EXHIBITION
JUNE 1-5, 1992
THE SHERATON NEW YORK

*For a detailed brochure call 212•274•9135
 or Fax: 212•274•0646*

able. Thus multiple inheritance is essential if the system is to have any flexibility at all.

GARBAGE COLLECTION

I'm sure you've all heard this one. What if your nuclear power plant has to do a garbage collect in the middle of a meltdown service routine? One side argues that garbage collection is thus a bad thing. The other says that one would have to be careful in using garbage collection in a hard real-time system with stringent response-time requirements, but that this does not describe most programming.

WHO'S RIGHT?

Ultimately, the more sensible participants will admit there are few, if any, universally right answers. Languages are designed to meet different goals. Criticizing them for not being something they were not intended to be is pointless.

To their credit, many of the better-known figures, including those closely associated with a particular language, have tried to defuse this sort of partisanship. There have been many patient explanations of the reasoning behind language features, and calm appeals to not try to use one tool for every job. Occasionally, though, you have to fight sarcasm with sarcasm, and I'd like to reproduce a particularly good example which appeared in comp.lang.eiffel.

An announcement had been posted for the Tenth Eiffel User Conference, to be held in Dortmund, Germany. John

Nagle (nagle@netcom.com) commented:

This is the TENTH conference? And still nobody uses it? Maybe there's something wrong.

Bertrand Meyer (bertrand@eiffel.com), who invented the Eiffel language, responded:

A small clarification may be useful here. As Mr. Nagle so competently points out, almost no one uses Eiffel; in fact until recently there were only nine users. But now a tenth person just started, so we are holding a conference, appropriately titled the TENTH EIFFEL USER conference, to celebrate.

The new user is in Canada, hence the word "international"; this is like "world" in "world series" for baseball.

We hope this helps clarify the issue, and sincerely apologize for any confusion the posting may have caused. ■

Alan Knight is a researcher in the Department of Mechanical and Aerospace Engineering at Carleton University, Ottawa, Canada, K1S 5B6. He currently works on problems related to finite element analysis in ParcPlace Smalltalk, and has worked in most Smalltalk dialects at one time or another. He can be reached at +1 613 788 2600 x5783, or by e-mail as knight@mrco.carleton.ca.

Separating the GUI from the application

Consider your garden-variety Smalltalk application running under a GUI. Of what major parts does it consist? In most cases, somewhere down in the depths is the Domain Model. This is made up from classes that represent the real things users think they are manipulating (e.g., transistors, diodes, connections, etc.). All of these classes are commonly implemented to know as little about the user interface as possible. Many would agree that ideally the domain model objects don't even know that such a thing exists. This is all old hat, and we'll assume that you're familiar with the concept.

The other common component of a GUI-based application is, not surprisingly, the user interface. Here is found all of the knowledge of what kind of widget is used to present which information, where it goes on the display, and what happens when the end user does something to it. Unfortunately, this part of most GUI-based applications comes in a lump. We'll cover how and why the UI component should be further divided.

When the user interface for a GUI-based application is examined, at least two broad categories of function can be found. The most apparent we'll call, for lack of a better term, the interface. The choice of display elements to be used falls into this category. For example, a certain set of choices may be offered as a list box, set of radio buttons, or pull-down menu. Other things in this category are the size and locations of such screen elements, how they respond to user input, what color they are, what fonts they use and so on. Anything pertaining directly to presentation or the first (or lexical) level of user input handling belongs in this category.

The other broad category involves control of the application. The semantic components of an application belong in this category. Examples include things like "list B must be refreshed if the user makes a new selection from list A." Other things that belong here are the handlers for the various commands (e.g., cut, copy, and paste) available to the user, or at least those that don't simply affect the display.

The control category contains all of the things that define what it means to be the application except for the last stages of presentation and the first stages of input. If it involves knowledge of the domain model or the relationships among the various pieces of information presented, it belongs in this category.

ICM ARCHITECTURE

Most "standard" Smalltalk application architectures of the past have had just two layers: a model and an interface. Using the approach presented here results in a three-layer application. We'll call this structure the Interface Control Model (ICM).

INTERFACE LAYER

All of the information pertaining directly to what shows up on the end-user's screen is part of the Interface layer. This includes the choice of display widgets, colors, fonts, menus vs. radio-buttons, as well as all of the first-level input handlers. They should normalize the input so that the Control layer doesn't have to know what kind of screen widget it's coming from.

If you use a window builder of any kind for this component, you'll be done in minutes. Except for the automatically generated methods used for setting up the various panes, buttons etc., nearly all of the methods given here will be one liners. They just mediate between the GUI-independent world of the control layer and the very GUI-dependent collection of user interface classes provided by Smalltalk.

CONTROL LAYER

This layer is where all of the work involved in producing a high-quality user interface ends up. This is where selected items mean something and where the smarts to translate user commands into action on the domain model live. The amount of thought that goes into this level will make or break your user interface.

The Control layer receives messages from the Interface layer that inform it about what the user is doing. Some of these messages identify selection of options or objects. Others notify the Control layer of the user's request that a command be executed. The Control layer, in turn, sends messages to the Interface layer to tell the interface what information is out of date (e.g., `updateListA`).

Though the Control layer knows that the Interface layer exists, has a pointer to it, and even knows a few messages it can send it, it does not by any means know everything. The Control layer should send only a very limited set of messages to the Interface layer. The Control layer should not even be aware of what class or classes the objects in the Interface layer are.

DOMAIN MODEL LAYER

As mentioned above, the good old Model layer is a well-beaten horse. It should suffice to say that objects living down here know little or nothing about the fact that they even have an interface. They may, if they're nice, notify unknown objects above when important information has changed.

ISOLATION

An important aspect of the ICM approach to application design is that the lower layers have as little knowledge about the layers above them as possible. They communicate only via a carefully designed protocol. The Model layer knows virtually nothing about the application. The Control layer knows all about the model, but very little about the Interface layer. It knows that it has an interface and knows a few messages it might send it (e.g., `updateListA`). The Interface knows nothing at all about the Model layer and only a little about the Control layer. For example, it knows what messages to send to the Control layer when the user has selected something or requested that a command be executed.

A QUICK EXAMPLE IN SMALLTALK/V PM

Listing 1 provides a quick example of some of the principles we've just discussed. It is an implementation of the user name/password requestor shown in Figure 1. The first class implements the Interface layer.

When the user executes a command, for example presses the "Ok" button to execute the "log-on" command, the interface does nothing more than tell the control layer that the user has chosen that command. At this point, the Interface layer is not even aware that the operation might result in a failure. Though the interface will be responsible for informing the user of such a failure (see the method for `#notifyInvalid` in Listing 1), it does not know where the failure occurs.

Note that none of the application state is maintained in the Interface layer. The current values for the user name and password are all kept in the Control layer as shown in Listing 2.

The class `SystemLogonControl` implements the control layer for our example application. To continue the example shown

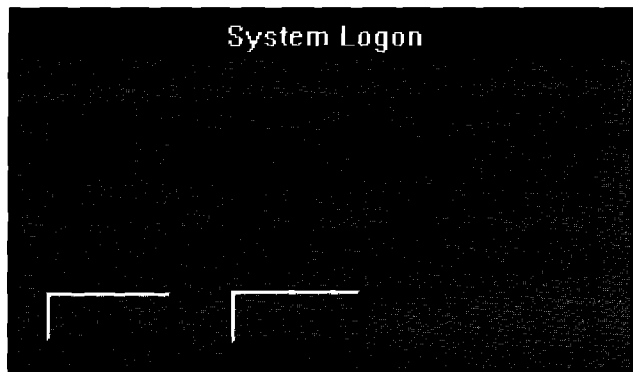


Figure 1. User name/Password Requestor.

Listing 1. An implementation of the user name/ password requestor.

```

WindowDialog subclass: #SystemLogonDialog
instanceVariableNames: 'control '
classVariableNames: ''
poolDictionaries: 'WBConstants '

bcCancel: aPane

    "Generated by WindowBuilder for a pane callback."

    "Button Command — Tell control that the user has
    chosen the 'cancel' command."
    self control cmdCancel

bcOk: aPane

    "Generated by WindowBuilder for a pane callback."

    "Button Command — Tell control that the user has
    chosen the 'logon' command has been chosen. (My
    button says 'Ok' but control and I have agreed that this
    represents the 'logon' command)."
    self control cmdLogon

control

    "Private — Answer the object who represents the
    control layer."

    control isNil ifTrue: [control := self defaultControl].
    ^ control

defaultControl

    "Private — Answer the default value for control."

    ^ self defaultControlClass new interface: self

defaultControlClass

    "Private — Answer the class of my default control layer."

    ^ SystemLogonControl

getUserName: aPane

    "Generated by WindowBuilder for a pane callback."

    "Ask control for his idea of what the user name is."
    aPane contents: self control userName

notifyInvalid

    "Sent only by my control layer. Tell the user that
    his log on failed."

    MessageBox
    notify: "
    withText: 'Logon Failed!!'

open

    "The usual, large method for opening all of the views. Only the
    creation code for the important subpanes is included."
    
```

Listing 1. (cont'd)

```

.
.
.
addSubpane: (
  .EntryField new
  owner: self;
  framingBlock: (...);
  paneName: 'userNameField';
  startGroup;
  tabStop;
  when: #getContents perform: #getUserName;;
  when: #textChanged perform: #setUserName;;
  yourself
);
addSubpane: (
  EntryField new
  owner: self;
  framingBlock: (...);
  paneName: 'passwordField';
  startGroup;
  tabStop;
  when: #textChanged perform: #setPassword;;
  yourself
);
addSubpane: (
  Button new
  owner: self;
  framingBlock: (...);
  paneName: 'okButton';
  defaultPushButton;
  startGroup;
  tabStop;
  when: #clicked perform: #bcOk;;
  contents: 'Ok';
  yourself
);
addSubpane: (
  Button new
  owner: self;
  framingBlock: (...);
  paneName: 'cancelButton';
  startGroup;
  tabStop;
  when: #clicked perform: #bcCancel;;
  contents: 'Cancel';
  yourself
);
.
.
.
openOn: aModel
  "Open me up on the given model."

  self control model: aModel.
  self open

setPassword: aPane

  "Generated by WindowBuilder for a pane callback."

  "The user has changed the password. Tell control
  about the change."
  self control password: aPane contents

setUserName: aPane

  "Generated by WindowBuilder for a pane callback."

  "The user has changed the user name. Tell control
  about the change."
  self control userName: aPane contents

```

in Listing 2, if the user requests the "log-on" command, the Interface will send the message #cmdLogon to the control layer. As can be seen in the method for this message above, the control layer then packages up the information necessary for the model to validate the log-on. Should the log-on be successful, the control layer decides that the application is at an end and closes things up. Otherwise, it decides the user needs to be notified of the failure. The actual notification is a presentation detail and is, therefore, left to the interface.

Admittedly, this is a rather simple example. However, we have used the ICM architecture, with very satisfying results, to implement much larger and more complex applications. Scalability is not a concern with this technique. In fact, as applications grow in size, this sort of division becomes commensurately more important.

SEPARATING THE HOST GUI FROM YOUR APPLICATION

Adding yet another layer to the all your Smalltalk applications may not sound just like what you were shopping for.

However, we're not making these suggestions just to get you to draw another box and arrow on all your slides. There are at least three considerable advantages to the ICM architecture. These advantages fall in the areas of maintenance, project management, and portability.

MAINTENANCE

In our experience with implementing highly interactive applications, each of the three layers is associated with a different level of code volatility. Once a reasonably solid Model layer has been implemented, it changes very little from version to version of the application. The code in the Control layer often changes only a little more frequently. Several more trials may be required to arrive at a solid Control layer, but once obtained it also changes little between versions. It is usually extended rather than modified.

On the other end of the scale, the presentation aspects of an application can change very rapidly. These are more often modified than extended. The presentation also bears the brunt of keeping up with changes in the host GUIs. By sepa-

Listing 2. The Control layer.

```

Object subclass: #SystemLogonControl
  instanceVariableNames: 'interface userName password model '
  classVariableNames: ''
  poolDictionaries: ''

cmdCancel
  "Command — The only thing that happens here is that
  the application closes up and goes away."

  self interface close

cmdLogon
  "Command — The user wants to log on. We'll give it
  a try. If we succeed, then the application should close
  up and go away. If we fail, then the interface should
  confront the user with the problem."

  (self model verifyPassword: self password forUser: self userName)
    ifTrue: [self interface close]
    ifFalse: [self interface notifyInvalid]

interface
  "Answer the object which implements the interface
  layer for this application."

  ^ interface

interface: anObject
  "Set the object which implements the interface
  layer for this application."

  interface := anObject

model
  "Answer the domain model (ie. the guy who knows
  about accounts, passwords and logon verification)."

  ^ model

model: aSystemLogonManager
  "Record the object I can ask to verify logon requests."

  model := aSystemLogonManager

password
  "Answer the password. If it's nil, it should
  default to an empty String."

  password isNil ifTrue: [password := ''].
  ^ password

password: aString
  "Set the password used to verify user logon."

  password := aString

userName
  "Answer the userName. If it's nil, it should
  default to an empty String."

  userName isNil ifTrue: [userName := ''].
  ^ userName

userName: aString
  "Set the account name of the user loggin on."

  userName := aString

```

rating the Control from the Interface, the more volatile aspects of the application are isolated from the more stable portions. This protects the stable code from the ravages of constant change.

PROJECT MANAGEMENT

The design of the presentation of an application and of its control, though related, involves different skills. In some organizations, different developers would be given responsibility for these areas. If the application is designed with a monolithic user interface, the two developers' work will be hopelessly mixed and mingled. By using the ICM architecture, their work would be cleanly divided. Also, the protocol by which their two components communicated could be easily defined.

PORTABILITY

One of the strongest reasons for using the ICM architecture is that it leads to applications that can be ported from one dialect of Smalltalk to another very rapidly. All of the knowl-

edge pertaining to the local host's GUI is kept isolated in a small, hopefully mostly automatically generated layer. Dialects of Smalltalk vary most widely in how they describe their user interface mechanisms. In applications designed using the ICM approach, the portable Control code and the nonportable, GUI-specific code are not all run together in a single layer. This allows applications to be moved between Smalltalks very rapidly. ■

Greg Hendley is a member of the technical staff at Knowledge Systems Corporation. His OOP experience is in Smalltalk/V(DOS), Smalltalk-80 2.5, Objectworks Smalltalk Release 4, and Smalltalk/VPM.

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, North Carolina 27511, or by phone 919.481.4000.

Why study Smalltalk idioms?

My dictionary defines an *idiom* as “a phrase whose meaning cannot be predicted from its words.” While learning Smalltalk (a task that continues daily) I have often been puzzled by a fragment of code. Only upon reflection do I understand the author's intent. About a year ago I began collecting examples of idioms I encountered, and asked my friends to tell me about ones they found. This article is an introduction to the material I have collected.

Many programmers new to Smalltalk spend most of their time just reading code. Studying idioms can accelerate this process. Knowing what to expect, or at least having somewhere to turn when you are baffled by a piece of code, is important to new Smalltalkers.

Another meaning for *idiom* is “a style of speaking of a group of people.” As with spoken language, Smalltalk has several dialects. The two most prominent are the Digitalk and ParcPlace dialects. There were also two distinct Tektronix dialects, easily distinguished from one another. Xerox Special Information Systems (the Analyst folks) also had their own distinctive style. New offshoots arise anywhere Smalltalk has taken root for several years.

Being conscious of the collective idiom of a body of code can also help more advanced programmers. Code that adheres to a shared idiom is easier to maintain, as there are fewer gratuitous surprises for new readers. Idioms also speed development through a kind of pattern-matching process. Once you have identified a circumstance in which an idiom is applicable, coding proceeds much faster than if you always have to invent new mechanisms from scratch. Standing on the brink of a new column, I look forward to exploring the range of idioms available to Smalltalk programmers. From time to time I'll be joined by prominent Smalltalkers who will describe their favorite idioms. We will also explore the subtle differences between the Digitalk and the ParcPlace schools.

This column will present idioms at many levels of complexity and scope. Rather than present all 50 or so of the idioms I have identified so far, I have chosen a smattering to get things going. The first few are small in scale and likely to trip up programmers new to Smalltalk. The concluding design idioms are more likely to interest more advanced programmers.

CONDITIONALS AS EXPRESSIONS

In most procedural languages, conditional statements do not return values. In Smalltalk, however, the result of sending `ifTrue:iffalse:` to the Boolean “true”, for example, is the value of the last expression in the block which is the first argument. This fact can be used to advantage to simplify some methods considerably. While you could write:

```
| result |
foo isNil
  ifTrue: [result := 5]
  iffFalse: [result := 7].
^result
```

It is shorter (and, after you get used to it, easier) to write:

```
| result |
result := foo isNil
  ifTrue: [5]
  iffFalse: [7].
^result
```

Once you've gone that far, you can get rid of the temporary variable entirely and simply write:

```
^foo isNil
  ifTrue: [5]
  iffFalse: [7]
```

and: AND or: VERSUS & AND |

There are two methods each for conjunction and disjunction in Smalltalk. `and:` and `&` both return true only if both the receiver and the argument are true, and `or:` and `|` both return true if either the receiver or the argument are true. The difference is that the keyword versions (`and:` and `or:`) take a block as an argument rather than a Boolean. The block is evaluated only if the result of the message is not determined by the receiver. For instance, you should use the keyword version of conjunction if evaluating the argument would cause an error if the receiver was false. For instance, if you wrote:

```
anArray size >= 10 & (anArray at: 10) isNil
```

you would get an error if `anArray` held less than ten elements.

In this case you would use the keyword version:

```
anArray size >= 10 and: [(anArray at: 10) isNil]
```

This way the `at:` message is not sent if `anArray` is too small. The Objectworks\Smalltalk release 4 image uses `or:` to determine if operating system resources (such as pixmaps) that do not survive over snapshots need to be reinitialized. It is common to see code like this:

```
(pixmap isNil or: [pixmap isOpen not]) ifTrue: [pixmap := Pixmap extent...
```

The other reason to use the keyword versions is for optimization. If the second part of a conjunction is expensive and the receiver is often false, using `and:` instead of `&` can result in a considerable savings. Why would anyone ever use the binary message versions of conjunction and disjunction? *Style*, baby. The keyword versions often introduce extra parentheses (as in the pixmap example above). They use far more characters. And since they are a little unusual, they require a moment of thought every time you encounter them.

DEFAULT PARAMETERS

Many programming languages provide the ability to not specify certain parameters for a procedure call and have them set to a default value. Smalltalk provides this facility through a programming idiom. A displayable object, for instance, might implement a message `display` as follows:

```
display
  self displayOn: Display
```

which in turn is implemented as:

```
displayOn: aDisplayMedium
  self displayOn: aDisplayMedium at: 0@0
```

and so on, until all the parameters needed to display the object have been collected. As the user of this object, you can specify as many or as few parameters as you need to get the job done.

The downside of implementing default parameters this way is the combinatorial explosion in the number of methods that can result. If you are creating default parameters for a method that has five parameters you could potentially create $5! = 120$ different methods. If you write all the possible combinations you obscure the purpose of the original method. If you don't write them all, you run the risk of not providing the combination that someone needs.

A common idiom for organizing default parameters is to choose a priority order. Create one method that defaults the most important parameter, another which specifies that parameter but defaults the next most important, and so on until you specify all parameters. In the example above, the destination for `display` is the most important parameter and the location, the next most important. This approach limits the number of methods, but ensures that the most commonly used combinations are available.

ABSTRACT SUPERCLASSES

Some classes are not meant to be instantiated. They exist only as repositories for interesting related bits of behavior. The

most powerful of these abstract superclasses reduce a set of related messages to one or two methods that each concrete subclass is required to implement. Both Smalltalks provide `Collection` as a good example. If you create a subclass of `Collection`, you need only implement `do:`. You get the rest of the enumeration methods without further effort.

Identifying candidates for abstraction is not easy. I got the following strategy for using this idiom from Ken Auer of Knowledge Systems. If reusability is ever going to be an issue for a class divide it into two parts at the beginning: an abstract part that contains only methods, and few or no variables, and a concrete part that holds the state necessary to actually compute. The example he used had an abstract `FinancialInstrument` and a concrete `Bond`. As you go along, only allow state to move into the superclass if you can't reasonably put it in the subclass. By pushing implementation decisions (state) down to the concrete class, you have a better chance of finding what is truly common to the implementation of all such objects by examining what is left in the abstract superclass.

Another strategy for finding abstract superclasses comes from Ward Cunningham. He suggests beginning an implementation without using inheritance at all. Only when you get tired of manually copying and pasting methods from one class to another do you factor their commonality into a superclass for both. This strategy has the advantage that it identifies commonality from concrete examples. The best use of inheritance for code sharing is often not apparent until far into the design.

VALUES MASQUERADING AS OBJECTS

One of the glories of objects is the ease with which they can be passed around. But this easy mobility can become a nightmare if you have passed off an object and it begins to change without your knowledge. There is a suite of idioms for dealing with these aliasing problems. The one described here is the simplest, but it can have the greatest performance impact. If once you have created an object you never change its state you cannot possibly have aliasing. I call objects used in this way "*values*" because of their similarity to numbers. In fact, numbers in Smalltalk are implemented in just this way. If you have the object 10 and you add 5 to it, you don't change 10, you get a new object, 15, instead. You don't have to worry about giving away your 10 and having it turn into a 15 behind your back.

`Points` and `Rectangles` are implemented much the same way. After you have created a `Point` with `Number>>@` all other operations (`+`, `*`, `translateBy:`) return new `Points`. Unfortunately, `Points` can have their coordinates changed directly via `x:` and `y:` and `Rectangles` also offer methods for directly changing their values.

The simplicity of value objects comes at a price. Their indiscriminate use can result in excessive memory allocation. If you must side-effect an otherwise functional object, do so only with a freshly allocated one in a small, well-defined scope (preferably a single method). As with all optimizations, pillaging a value object for speed should only be done when the

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 offices, Product Announcements Dept., 91 Second Ave., Ottawa, Ontario K1S 2H4, Canada.

Object Technology International Inc., (OTI) has announced a major development agreement with International Business Machines Corporation. The new agreement with IBM's Applications Business Systems (ABS) will enhance the use of graphical user interfaces and objects in application development.

ABS will work with OTI, a leading object-oriented developer, to create an object-oriented environment for developing cooperative applications. OTI will combine the object-oriented ENVY/Developer technology with AS/400 cooperative processing support to provide a client-server application development environment for Smalltalk/V programmers. Smalltalk/V is an object-oriented development language provided by AD/Cycle International Alliance Member Digitalk Inc.

This new development agreement is intended to enhance the capabilities of AS/400 programmers using graphical user interfaces, cooperative programming and object-oriented programming.

For more information, contact Object Technology International Inc., 1785 Woodward Drive, Ottawa, Ontario, Canada K2C 0P9; (613) 228-3535.

ExperTelligence has announced a new version of its intelligent interactive graphical application development system **Action!** for version 1.3 of Digitalk's Smalltalk/V PM.

The new version extends Action! V1.2. It adds new CUA 91 objects like the Spin Button and the Slider. It also gives more control to developers with the powerful Properties Editor and Coordinate Windows. Finally, it allows interface compatibility with the Macintosh Lisp versions of Action!

Using Action!, an interface developed in Macintosh common Lisp or Procyon common Lisp on a Macintosh can be instantaneously ported to an Intel board machine running Smalltalk/V PM.

Action! V 1.3 for Smalltalk/V PM is now available directly from ExperTelligence.

For more information, contact ExperTelligence Inc., 5638 Hollister Avenue, Suite 302, Goleta, CA 93117; (805)967-1797.

Digitalk Inc., the developer of the Smalltalk/V object-oriented programming system and a member of the IBM International Alliance for AD/Cycle, today announced the acquisition of **Instantiations Inc.** Instantiations provides a wide range of services to Fortune 500 companies that are developing applications using Smalltalk object-oriented technology.

Instantiations, led by object technology veteran Michael Taylor, is com-

posed of some of the industry's leading object technology experts. Principal among these are Allen Wirfs-Brock, a well-known Smalltalk expert, and Rebecca Wirfs-Brock, noted author and object-oriented design methodologist.

For further information, contact Digitalk Inc., 9841 Airport Boulevard, Los Angeles, CA 90045; (310) 645-1082.

Digitalk Inc. has announced that it is developing a 32-bit version of its Smalltalk/V development environment for UNIX to be delivered by year-end. The first platform for the company's new UNIX technology will be IBM's RS/6000 RISC (Reduced Instruction Set Chip) machine which runs AIX, IBM's version of UNIX.

The new UNIX version of Smalltalk/V is based on Digitalk's 32-bit Smalltalk/V technology for OS/2 2.0. Developers can develop their applications on either Smalltalk/V for OS/2 or Smalltalk/V for Windows, and these applications will run unmodified on the new UNIX release.

For more information, contact Digitalk Inc., 9841 Airport Boulevard, Los Angeles, CA 90045; (301) 645-1082.

Object Technology International Inc. has announced the immediate availability of its object-oriented product development environment, **ENVY/Developer R1.30**, for Smalltalk/V PM V1.3 and Smalltalk/V Windows V1.1.

This product provides a powerful concurrent software engineering environment for systems and applications development. Team support, version control and configuration management are seamlessly integrated with Smalltalk/V's programming environment.

Release 1.30 includes ENVY/Packager, OTI's tool for delivering small, standalone Smalltalk applications. Developers are provided with fine-grained control over the inclusion and placement of objects included in the final product.

This new release supports two Smalltalk language implementations: Smalltalk/V Windows V1.1 for Windows 3.0 and Smalltalk/V PM V1.3 for OS/2. Supported networks include Novell NetWare, LAN Server and LAN Manager. Configurations are available from three-user systems up to site or special corporate licenses.

For further information, contact Object Technology International Inc., 1785 Woodward Drive, Ottawa, Ontario, Canada K2C 0P9; (613) 228-3535.

performance of the finished applications is a problem for real users, never on mere speculation.

CONCLUSION

A good grasp of Smalltalk's many idioms can speed assimilation of the language and its class libraries, improve the productivity of new development, and accelerate understanding of legacy code. This article has only scratched the surface of known Smalltalk idioms, all of which were present in Smalltalk-80 as it escaped from Xerox. The dispersion of

Smalltalk will fuel the growth of many new idioms.

I am still collecting idioms. If you identify one you would like to share, contact me. ■

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPars Computer. He is also the founder of First Class Software, which develops and distributes re-engineering products for Smalltalk. He can be reached at P.O. Box 226, Boulder Creek, CA 95006 or kentb@maspar.com.

WHAT THEY'RE SAYING ABOUT SMALLTALK

Excerpts from industry publications

...Changing the way you develop software is a nontrivial decision. You are not going to take seasoned FORTRAN or C programmers and turn them into Smalltalk wizards overnight. You are likely, however, to find one or more programmers in your development group who are highly interested in object technology. These self-motivated, early adopters are good candidates for a core transition team within your development group. But they must be given training in object-oriented analysis and design as well as Smalltalk programming. (Important tip: A Smalltalk programmer with a structured, procedural mindset is not really a Smalltalk programmer)...

Technology: Smalltalk speaks to users' needs, Jim Salmons and Timlynn Babitsky, OPEN SYSTEMS TODAY, 2/17/92

...Practical studies based on function point analysis show that a 4GL solution to a typical problem is 50% simpler than 3GL solution, but only 15% simpler than an OOPL solution. This is probably because the OOPL's user-defined types are counted, while those built into the 4GL are not. The size and complexity of a 4GL solution grows incrementally with the system. With an OOPL, you just add components to meet each new challenge. Fourth-generation and other specialized languages will stick around, if only for cultural reasons (most of today's programmers cut their teeth on them). The world is unlikely to rally around a single OOPL. Technical and financial developers use C++. MIS shops prefer Smalltalk or are waiting for object-oriented Cobol. Ironically, object-oriented 4GLs now accompany some OO databases to help them integrate objects from multiple OOPLs.

Software & Systems: Fourth generation heyday at an end, Craig Hubley, COMPUTING CANADA, 2/17/92

...At press time [Sequent's Larry] Evans didn't want to elaborate on Ptx/Object, but according to documents obtained by UNIX WORLD, the product will combine a parallel version of the ParcPlace Objectworks/Smalltalk development environment and a parallel implementation of Versant Object Technology Corp.'s object database management system. This will enable users to tap information from legacy systems and convert it to objects...

Because the object-oriented market is in its infancy, Ptx/Object won't generate any short-term income. As the object-oriented market becomes more mature, however, Sequent's early entrance will help it, say analysts...

Sequent's software cure, Gary Andrew Poole, UNIX WORLD, 3/92

...But we [Microsoft] also expected (and predicted) that "OOP" would become a vendor buzzword long before great OOP solutions were generally available. Our concern was to ensure that the press did not treat OOP as an end in itself, but merely as a means: to make software easier to design and maintain for developers; to make applications easier to use and more functional for end users. If OOP does not provide these benefits, then it remains only an acronym, not a solution....Our concerns about OOP being over-sold, however, remain. Object-oriented tools and systems are the most sophisticated and complicated of any software products ever built. Object orientation is not something that can be tacked onto

an existing system; it must be designed in. It is very difficult to do right, technically. And the focus must always remain on providing real value to the consumer...

Object insider: Bill Gates, OBJECT MAGAZINE, 3-4/92

...There is a creative tension building in the computer industry between the "way we used to do it" and the "way we should do it." Object technology is causing this tension, as the vision of the way we should do it clashes with current reality. There is only one way to resolve this tension we find ourselves in: either pull the vision of object technology towards reality, or pull reality toward the vision...

Object Request Broker—the end of the beginning, Chris Stone, OBJECT MAGAZINE, 3-4/92

...In the longer term, it will be up to content specialists to define the basic services that classes should provide in a way that makes them more independent of their implementation context...

...In addition to warranties and certification processes, I believe that class vendors should provide testing facilities with every class they ship. Ideally, these facilities would include a complete test bed for automatically sending a full range of messages to the class and checking the correctness of its responses. Providing developers with these testing tools would allow them to double check the performance of any class. More importantly, it would help them assure that any modifications or subclassing they performed had not violated the basic functionality of the class...

...Ultimately, the solution to compensating class creators will be determined by market forces, which usually have a way of defying the most well-intentioned attempts at prediction...

Easing into objects: developing the object components industry, David A. Taylor, OBJECT MAGAZINE, 3-4/92

...The changing nature of systems, however, including a move to distributed development and deployment, the increased use of graphical user interfaces, improvements in language and environment technology, and widespread availability of classes and libraries, will speed the adoption of object-oriented programming languages. Large systems of the future will evolve on a project or functional basis rather than as multi-year phased systems. As the world migrates to reusable components and the development of organic systems, small methods will once again dominate. A method like IE, based on the mainframe-centric master enterprise model view of the world, will ultimately collapse under its own weight. When objectified versions appear, view them with caution, and ask if the problems they are solving will be relevant to you by the time you complete your enterprise model.

Methodology: objectified information engineering—the method time bomb?, Adrian Bowles, OBJECT MAGAZINE, 3-4/92

...Relational vendors believe that by extending the capabilities of their database servers, or the capabilities of some front-end tools, many of the benefits of object technology can be achieved without

adopting a new database model. Sybase's [VP of marketing, Stewart] Schuster referred to object technology as a "natural extension" rather than a "fundamental paradigm shift." [Mary] Loomis [VP of technology for Versant Object Technology] noted that RDBMSes have stored procedures and BLOBS, but claimed that was not adequate. "They are baby steps," Loomis said. "Some RDBMSes have stored procedures which begin to put some actions in the databases, but they're in a very [limited] way. They really don't couple the data with the action. Object databases are much more than BLOBS.

SPARC databases square off, Barry D. Bowen, OPEN SYSTEMS TODAY, 2/17/92

...According to the January/February 1992 issue of PC AI, the value of the relational database market in 1980 was \$2 million, but in 1990 it was \$2.5 billion. Similarly, the value of the object-oriented market today is \$10 million. By 1995, PC AI estimates the value will skyrocket to \$235 million...

Quick statistics, COMPUTER EDGE, 2/7/92

...Now they're scrambling to recreate their [CASE] programs to write software for desktop machines and local area networks. Many of the big mainframe CASE vendors, including IBM, Texas Instruments Inc., and Andersen Consulting, are already coming out with new products for these markets...Otherwise, there will be a reapportioning of IS dollars in favor of smaller CASE vendors...Such as object-oriented programming environments. AT&T, for instance, is using an object-oriented CASE program called Teamwork from Cadre Technologies Inc. of Providence, RI, to build a piece of its 911 service. The object-oriented technology combined with CASE programming capabilities, says AT&T systems engineer Michael Krass, lets the company write code that can be easily reused and maintained—unlike traditional CASE...

The case against CASE, Robert Moran, INFORMATION WEEK, 2/17/92

...[Sun Microsystems's R.G.G.] Cattell made a couple of observations on choosing among the available object databases: "For the object database in question, look at the power of the query languages for associative retrieval or queries across sets of objects. Second, how well are they integrated with a programming language?" In some areas, the differences among commercial offerings are minor. "Based on the [Cattell] benchmarks, there is not much difference in performance among the object databases. Further more, the size of the benchmark database is not important. There is no storage-requirement penalty." He said he would choose a database largely on the basis of the company and its stability. "The technical differences are small." What about writing your own? "I would lean toward buying," Cattell said. "You can always build something faster relative to what you can get off the shelf, but the performance of the commercial object databases is quite impressive."

Guidelines for choosing the best database technology, Robert H. Blissmer, ELECTRONIC ENGINEERING TIMES, 2/17/92

...[Gartner Group's David Stein]"Object technology is probably the most significant development in software technology in 40 years. Like all major baseline technology shifts, this one won't be felt immediately, but over a 10-year period. A massive amount of development has to be done, but this is so far-reaching in its effects that it will impact literally everything that's being done with software. You now have the

ability to standardize, and that changes the economics of software development, which is where the big bucks are spent..."

[Merrill Lynch & Co.'s Anthony Pizi]"...Multimedia is the future for training. It allows people to learn at their own pace. This technology is very good for training people in a standardized way. A firm's staff might be spread out among five different offices, but they can all use the same training program. Investment bankers might use multimedia to give a presentation to a client and make changes on the fly. But one of the biggest problems is portability. You need either tremendous amounts of memory or a U-haul to lug all the hardware like CD player, CPU and monitor. Live video is on its way, but any time you incorporate video and sound, huge files are created, which have to be compressed. More "affordable" boards are coming out, so end users don't have to worry about losing any data..."

[Andersen Consulting—Financial Markets Industry Group's Robert Gach]"...Rule-based processing and object-oriented development are technologies that firms on the Street could use to steer clear of large applications, which have millions of lines of code and don't lend themselves to flexible new product development. Object-oriented technology and maximizing code reusability—changing something, copying it or adding a variant—as opposed to starting from scratch, would save the Street a lot of development costs and time. Applications would get to traders faster if programmers could reuse packets of logic that are two to three commands long. Applications with only 30% to 40% of the code are easier to maintain, document, understand and fix. In addition, by using both these technologies, programmers stay in the end user's world and define systems from a business standpoint..."

Hot technologies for the 1990s, Ann Goodman and Jenna Michaels, WALL STREET & TECHNOLOGY, 2/92

...[Computer Associates' Dominique Laborde:]"The industry is beginning to pay attention to object-oriented programming... IBM does not have any products in this category but needs to say something about an object-oriented DBMS. So the company published a broad list of specifications. Right now, those specifications are not mature enough for us or anyone else to support..."

Data access solution now has IBM road map, Paul Korzeniowski, SOFTWARE MAGAZINE, 2/92

...The constant refrain heard among the audience [at PC Forum] was this: We shouldn't talk about objects because nobody agrees what they are, users don't see them, and nobody really has them. The notion was that objects are sort of the industry's dirty laundry, so they really shouldn't be hung out in decent company...We disagree. Objects should not be swept under the rug. We know that the notions of object orientation have been around the industry since the early 1970s. But we believe that the technology infrastructure is only now becoming sufficiently powerful and sophisticated so those notions can be implemented in everyday systems. And we think that development will change everything about computing for our readers. And we mean everything, including what skills you need to be successful, what kind of products you should invest in, the methods you use to evaluate those products, the expectations you have (and set for your management) about how fast you can implement new applications, and the approaches you choose to adopt in designing your systems...A key benefit of object-based systems, for instance, should be to move the locus of responsibility for applying technology from the vendor...to the customer...

Editorial: Object-oriented technology needs to be thrashed out, Stewart Alsop, INFO WORLD, 3/2/92



WINDOWS AND OS/2: PROTOTYPE TO DELIVERY. NO WAITING.

In Windows and OS/2, you need prototypes. You have to get a sense for what an application is going to look like, and feel like, before you can write it. And you can't afford to throw the prototype away when you're done.

With Smalltalk/V, you don't.

Start with the prototype. There's no development system you can buy that lets you get a working model working faster than Smalltalk/V.

Then, incrementally, grow the prototype into a finished application. Try out new ideas. Get input from your users. Make more changes. Be creative.

Smalltalk/V gives you the freedom to experiment without risk. It's made for trial. And error. You make changes, and test them, one at a time. Safely. You get immediate feedback when you make a change. And you can't make changes that break the system. It's that safe.

And when you're done, whether you're writing applications for Windows or OS/2, you'll have a standalone application that runs on both. Smalltalk/V code is portable between the Windows and the OS/2 versions. And the resulting application carries no runtime charges. All for just \$499.95.

So take a look at Smalltalk/V today. It's time to make that prototyping time productive.

Smalltalk/V

Smalltalk/V is a registered trademark of Digitalk, Inc. Other product names are trademarks or registered trademarks of their respective holders.

Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045
(800) 922-8255; (213) 645-1082; Fax (213) 645-1306

LOOK WHO'S TALKING

HEWLETT-PACKARD

HP has developed a network troubleshooting tool called the Network Advisor. The Network Advisor offers a comprehensive set of tools including an expert system, statistics, and protocol decodes to speed problem isolation. The NA user interface is built on a windowing system which allows multiple applications to be executed simultaneously.

NCR

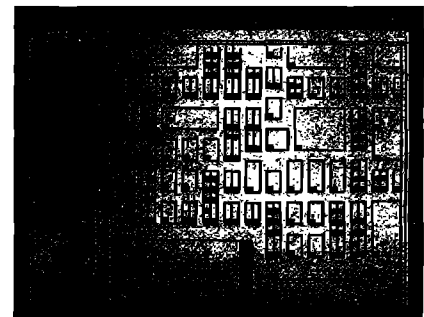
NCR has an integrated test program development environment for digital, analog and mixed mode printed circuit board testing.

MIDLAND BANK

Midland Bank built a Windowed Technical Trading Environment for currency, futures and stock traders using Smalltalk V.

KEY FEATURES

- World's leading, award-winning object-oriented programming system
- Complete prototype-to-delivery system
- Zero-cost runtime
- Simplified application delivery for creating standalone executable (.EXE) applications
- Code portability between Smalltalk/V Windows and Smalltalk/V PM
- Wrappers for all Windows and OS/2 controls
- Support for new CUA '91 controls for OS/2, including drag and drop, booktab, container, value set, slider and more
- Transparent support for Dynamic Data Exchange (DDE) and Dynamic Link Library (DLL) calls
- Fully integrated programming environment, including interactive debugger, source code browsers (all source code included), world's most extensive Windows and OS/2 class libraries, tutorial (printed and on disk), extensive samples
- Extensive developer support, including technical support, training, electronic developer forums, free user newsletter
- Broad base of third-party support, including add-on Smalltalk/V products, consulting services, books, user groups



This Smalltalk/V Windows application captured the PC Week Shootout award—and it was completed in 6 hours.



Smalltalk/V PM applications are used to develop state-of-the-art CUA-compliant applications—and they're portable to Smalltalk/V Windows.