The Smalltalk Report

The International Newsletter for Smalltalk Programmers

March–April 1994 **Volume 3 Number 6**

POOLS: AN ATTRACTIVE NUISANCE

by Juanita Ewing

Pools are a Smalltalk language construct for sharing data between class. Classes that share data using pools are not required to be hierarchically related. At first blush, pools sound attractive: pools allow functionally related classes to connect by sharing data.

However, pools are not without problems. They are poorly supported in most Smalltalk implementations and limit reusability. Hence, they are labeled an attractive nuisance.

What are Pools?

Pools are dictionaries of variables. The variables in a pool are called pool variables. Each variable has a value that is often a constant, but there is no language constraint that the variables be constants.

How do you use Pools?

When you define a class, you can instruct the Smalltalk compiler to use pool variable names when compiling instance and class methods. In some Smalltalk implementations, pool access is inherited by subclasses.

Show me

Suppose you have a class, called Stream, that is used for reading and writing data. You also have a pool called CharacterConstants, containing variables that describe commonly used characters. It would define variables such as Cr, Lf and Tab. You could use the CharacterConstants pool to implement methods for writing formatted data to the stream.

First, let's examine the Stream class definition:

```
Object
  subclass: #Stream
  instanceVariableNames: 'collection position size'
  classVariableNames: ''
  poolDictionaries: 'CharacterConstants'
```

In addition to instance variables and class variables, developers can customize the set of pools used by the class. Stream uses one pool, CharacterConstants.

Next, let's examine the method nextLine that uses variables from the pool CharacterConstants. Both Cr and Lf are pool variables from CharacterConstants.

```
Stream
nextLine
  "Answer a String consisting of the characters of the receiver up to the next line delimiter."

  | answer |
  answer := self upTo: Cr.
  self peekFor: Lf.
  ^answer
```

continued on page 4

Contents:

John Pugh

Paul White

EDITORS' CORNER

Last month, we spoke of the lack of discussion in major publications on the topic of managing object-oriented projects. Another related topic, rarely discussed, is the difficulty faced by organizations with respect to managing corporate software libraries. Of course, everyone can see the obvious potential of one "corporate" library managed across the organization, but very few are seeing it implemented.

We believe that there are some important changes that have to occur before library reuse will ever be possible on a large scale. First, we as software developers must change our distrust of software developed by others—the so-called "not invented here" syndrome. While we recognize this distrust is often warranted, it is an inhibitor to wide-scale reuse. Second, we need better tools for browsing our libraries. Today's tools don't provide adequate facilities for searching the library. The implementors feature, for example, is great if you know the name of the message but doesn't help with the "I wonder if there's a message that deals with so-and-so" queries. Third, the accounting procedures used by most organizations, in terms of budgeting time and resources, have to be changed. If it is my job to maintain a piece of software for other groups to reuse, I will require time away from my regular job to do this. Fourth, we need to better capture the design of the classes we intend to reuse—it is often difficult to understand the intent of a class without having up-to-date design information. We feel there are a few Ph.D. dissertations in this area of "library science" if anyone is interested.

Since we don't do it often enough, we would like to acknowledge the importance of our columnists. Their columns are always of the highest quality and cover a wide variety of interesting and informative topics. We look forward to learning more about Smalltalk from each column they write, and, from the comments we receive, we know you do too. This month's issue features three of our regulars. Juanita Ewing's "Getting Real" column discusses the use of PooledDirectories and where users may find some of the hidden problems. Kent Beck is back this month with a continuation of his "Where do objects come from?" series describing the tricky problem of returning more than one object from a method and the associated maintenance problems. And Alan Knight reports on the many ways one can obtain Smalltalk-related information from the "information superhighway" so widely talked about these days.

John Pugh Paul White

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 Inc., 588 Broadway, New York, NY 10012 212.274.0640.

Mailed First Class. Canada Post International Publications Mail Product Sales Agreement No. 290386. Subscription rates 1 year (9 issues): domestic, $79; Foreign and Canada, $94; Single copy price, $8.

POSTMASTER: Send address changes and subscription orders to: THE SMALLTALK REPORT, P.Box 2027, Langhorne, PA 19047. For service on current subscriptions call 215.785.5996.

To submit articles, please send electronic files on disk to the Editors at 509–885 Meadowlands Drive, Ottawa, Ontario K2C 3N2, Canada, or via Internet to pugh@scs.carleton.ca Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

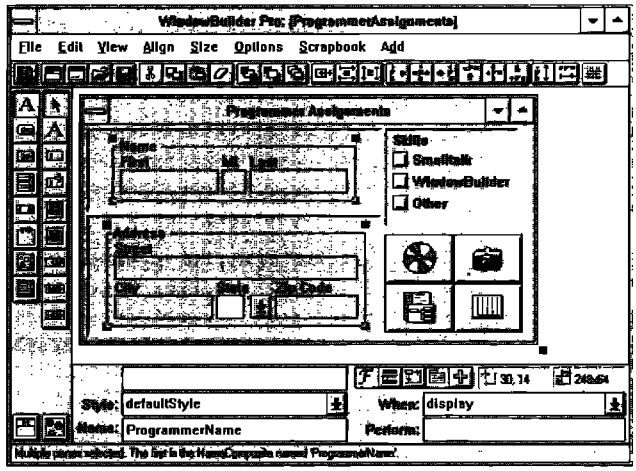
PRINTED IN THE UNITED STATES.

SHARE OBJECT SYSTEMS INC.

WINDOWBUILDER PRO!

The New Power in Smalltalk/V Interface Development

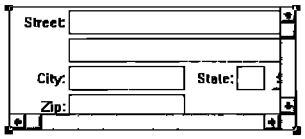
Smalltalk/V developers have come to rely on WindowBuilder as an essential tool for developing sophisticated user interfaces. Tedious hand coding of interfaces is replaced by interactive visual composition. Since its initial release, WindowBuilder has become the industry standard GUI development tool for the Smalltalk/V environment. Now Objectshare brings you a whole new level of capability with WindowBuilder Pro! New functionality and power abound in this next generation of WindowBuilder.



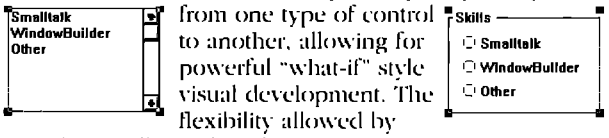
WindowBuilder Pro/V is available on Windows for $295 and OS/2 for $495. Our standard WindowBuilder/V is still available on Windows for $149.95 and OS/2 for $295. We offer full value trade-in for our WindowBuilder customers wanting to move up to Pro. These products are also available in ENVY/*Developer* and Team/V™ compatible formats. As with all of our products, WindowBuilder Pro comes with a 30 day money back guarantee, full source code and no Run-Time fees.

Some of the exciting new features...

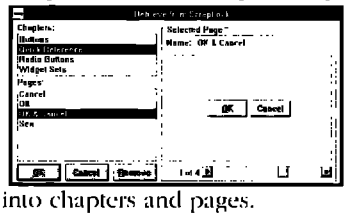
• CompositePanes: Create custom controls as composites of other controls, treated as a single object, allowing the developer higher leverage of reusable widgets. CompositePanes can be used repeatedly and because they are Class based, they can be easily subclassed: changes in a CompositePane are reflected anywhere they are used.



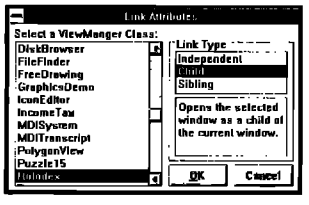
• Morphing: Allows the developer to quickly change from one type of control to another, allowing for powerful "what-if" style visual development. The flexibility allowed by morphing will greatly enhance productivity.

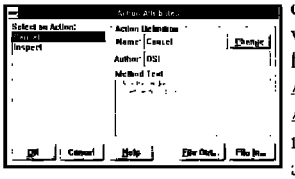


• ScrapBook: Another new feature to leverage visual component reuse, ScrapBooks provide a mechanism for developers to quickly store and retrieve predefined sets of components. The ScrapBook is a catalog of one's favorite interface components, organized into chapters and pages.



• Rapid Prototyping capabilities: With the new linking capabilities, a developer can rapidly prototype a functional interface without writing a single line of code. LinkButtons and LinkMenus provide a powerful mechanism for linking windows together and specifying flow of control. ActionButtons and ActionMenues provide a mechanism for developers to attach, create, and reuse actions without having to write code. These features greatly enhance productivity during prototyping.





• ToolBar: Developers can Create sophisticated toolbars

just like the ones in the WindowBuilder Pro tool itself.

• Other new features include: enhanced duplication and cut/paste functions, size and position indicators, enhanced framing specification, Parent-Child window relationship specification, enhanced EntryField with character and field level validation, and much more...

• Add-in Manager: Allows developers to easily integrate extensions into WindowBuilder Pro's open architecture.

Catch the excitement, go Pro!
Call Objectshare for more information.

(408) 727-3742

Objectshare Systems, Inc
Fax: (408) 727-6324
CompuServe 76436,1063

5 Town & Country Village
Suite 735
San Jose, CA 95128-2026

What's wrong with that?
Nothing's wrong so far. Now let's try some important operations like defining a new pool. The traditional way to define a new pool in Smalltalk is to create a global variable whose value must be a dictionary. Then the user must populate the dictionary with keys that will be interpreted as variable names when the dictionary is used as a pool. Some implementations of Smalltalk require the keys in the dictionary to be Strings, others require Symbols. The code to create a pool looks like this:

```
"Declare the variable"
Smalltalk at: #MyUIConstants put: nil.

"Create the dictionary."
MyUIConstants := Dictionary new.

"Populate"
MyUIConstants at: 'BackgroundColor' put: Color paleYellow.
MyUIConstants at: 'ForegroundColor' put: Color blue.
MyUIConstants at: 'TextColor' put: Color black.
MyUIConstants at: 'TextHighlightColor' put: Color darkYellow.
```

How do you use that pool?
In this example, we created a pool called MyUIConstants, and filled it with color values. Now we can use this pool in the definition of the class TextWidget:

```
Widget
  subclass: #TextWidget
  instanceVariableNames: 'contents'
  classVariableNames: ''
  poolDictionaries: 'MyUIConstants'
```

Why did we use a dictionary to define a pool?
You'll notice that creating a pool didn't involve any expressions of the form Smalltalk createPoolNamed:#MyUIConstants. Instead, we create a global variable and set its value to a dictionary.

Historically, pools were never formally defined as first-class elements of the Smalltalk language. There is no syntax for defining pools or pool variables. Instead, the exact implementation of the pool language constructs known and relied upon by developers. This isn't a good idea because it prevents vendors from improving the implementation of pools—future versions of Smalltalk may not even use dictionaries to implement pools. It also makes it difficult for developers to move their code to different Smalltalk platforms that have a different implementation of pools. The worst thing, though, is that developers write code that treat pools as dictionaries.

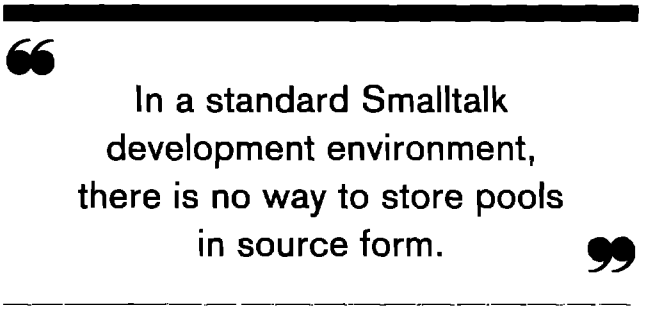
What problems result from treating pools as dictionaries?
Because pools are globals and available from every method, and their implementation is known, developers are very tempted to write code like this:

```
MyUIConstants at: 'TextColor' ifAbsent:[^nil]
```

The problem with this code is that the compiler does not detect it as a pool variable reference. It is just a message send to a global variable. Thus, the Smalltalk programming environment cannot reason about this expression as a pool variable reference.

This type of reasoning would be important if you were designing your program, and were considering eliminating the pool variable TextColor. If you asked the programming environment to search for all references to the pool variable TextColor, it would not find this one.

Another problem related to the public implementation of pools and the availability of a pool in the global name scope, is inappropriate access of pool variables. Pool variables can be accessed in any method, not just methods in classes that define usage of the pool. If the pool is treated like a dictionary, you can send it a message to access its contents, which are pool variables. For example, the expression ColorConstants at: 'ClrBlue' provides access to the pool variable ClrBlue.



How do I store pools?
A source form of a Smalltalk application is more than just a rarely used archiving artifact. It is a necessity for serious developers. For a more complete discussion on the benefits of storing source, see my SMALLTALK REPORT column on "How to Manage Source Without Tools, " (Volume 2, Number 3).

The typical way developers create a source form of their application is by filing classes out of an image. When developers file classes in and out of an image, they encounter another problem with pools. A class that references a pool can file out without a problem, but its pools are not filed out. Because pools are shared between classes, it would be inappropriate to file them out with any single class. Instead, pools should be independently stored in source form.

In a standard Smalltalk development environment, there is no way to store pools in source form. Most Smalltalk environments don't even define a source form.* The pools must be present, however, when you file your class back in.

What lessons have developers learned?
If developers have defined and used pools before, they have learned to save the code they used to create the pool, and execute it again to recreate pools when rebuilding their development environment. This is typically some workspace code, and it is usually saved in a file.

To rebuild their development environment, developers must manually track which classes require which pools, and rebuild their development environment with a combination of source code to recreate classes and executable code to recreate

global and pools.

Developers have also learned to write their pool definition code carefully because pool definition code is fragile. If you overwrite an existing pool by creating a new dictionary, any existing code using pool variables will be disconnected from the pool. Changes to the pool will not be tracked by the compiled code.

Suppose you have a class, TextWidget, that uses the pool MyUIConstants. The method initialize uses two pool variables, TextColor and TextHighlightColor.

```
TextWidget
initialize
    "Initialize the receiver for standard look and feel."

    self setTextColor: TextColor.

    self setHighlightColor: TextHighlightColor
```

You can redefine the pool with this expression:

```
Smalltalk at: #MyUIConstants put: Dictionary new.
```

But, the redefinition breaks the connection between the pool and existing references from methods. This is because you have created a new pool that happens to have the same name as the old pool. You are not redefining the old pool. Even if you populate the new dictionary with identical variables you cannot reestablish the connections:

```
MyUIConstants at: 'BackgroundColor' put:Color black.
MyUIConstants at: 'ForegroundColor' put: Color blue.
MyUIConstants at: 'TextColor' put: Color white.
MyUIConstants at: 'TextHighlightColor' put: Color green.
```

The original value of TextColor was black. In the new pool MyUIConstants, its value is white. The initialize method still contains a reference to the old pool variable TextColor, and initializes TextWidgets to have the a black text color. Examining the source of the method gives no clue about the current state of the compiled code. Recompiling the method will allow the compiler to rebind the reference.

To avoid redefining existing pools, developers usually place conditionals around pool creation expressions (requiring further assumptions about the implementation of pools):

```
(Smalltalk includesKey: #MyUIConstants)
    ifFalse: [Smalltalk at: #MyUIConstants put: Dictionary new]
```

Accidental pool redefinition is another reason why it is dangerous to allow the implementation of pools to be known.

What impact do pools have on reusability?

Pools have a negative impact on the sacred cow of Smalltalk, reusability. Let's examine our definition of pools again: a construct for sharing *data* between classes. In other words, pools contain data and do not define behavior. The two main mechanisms for reuse in Smalltalk are inheritance and polymorphism. Both are focused on behavior. They rely on behavior functioning on encapsulated data–exactly the opposite of what pools provide.

Let's look at an example with the pool MyUIConstants. The class TextWidget uses the pool to access user interface constants. The method initialize is implemented as follows:

```
TextWidget
initialize
    "Initialize the receiver for standard look and feel."

    self setTextColor: TextColor.
    self setHighlightColor: TextHighlightColor
```

Suppose we make a variation of TextWidget that has a different highlight color. We don't want to change the original class, so we create a subclass of TextWidget that uses the MyUIConstants pool. And, we add a new pool variable to represent the new highlight color, called MarkupTextHighlightColor.

```
MyUIConstants at: 'MarkupTextHighlightColor' put: Color darkYellow.
```

Because the inherited initialize method contains a direct reference to the pool variable, we are forced to override the entire initialize method instead of just overriding the color specification for text highlight. Here is the new initialize method for the subclass:

```
MarkupTextWidget
initialize
    "Initialize the receiver for mark up look and feel."

    self setTextColor: TextColor.
    self setHighlightColor: MarkupTextHighlightColor
```

A better way to write this code is to isolate and encapsulate references to the constants in this method. Then subclasses can override the encapsulating methods if necessary. Remember though, that these constants are used in several classes. It may be better, depending on the usage, it the constants are encapsulated in methods from a stand-alone class.

A new class, WidgetUIConstants, could function as the encapsulator for all user interface constants. It would respond to messages like foregroundColor and textHighlightColor. Straightforward use of this class would look like this:

```
TextWidget
widgetConstantClass
    "Return the class containing user interface
      constants."

    ^WidgetUIConstants

initialize
    "Initialize the receiver for mark up look
      and feel."

    self setTextColor: self widgetConstantClass
      textColor.
    self setHighlightColor: self widgetConstant
      ClasstextHighlightColor
```

A new subclass of WidgetUIConstants could contain the variations appropriate for the subclass MarkupTextWidget. MarkupTextWidget now needs to override the specification of the user interface constants class, but does not need to override the initialize method.

```
MarkupTextWidget
widgetConstantClass
    "Return the class containing user interface constants."

    ^MarkupWidgetUIConstants
```

This example illustrates that it is not straightforward to override references to pool variables in a subclass. The override usually results in multiple methods that specify the same constants, which leads to maintenance problems. This is typical of the extensibility problems found in cases of direct variable references.

The added benefit of a stand-alone class alternative is that the class can be stored in source form and managed by ordinary Smalltalk tools. It can also be subclassed to provide variations of the constants. Pools have neither of these capabilities.

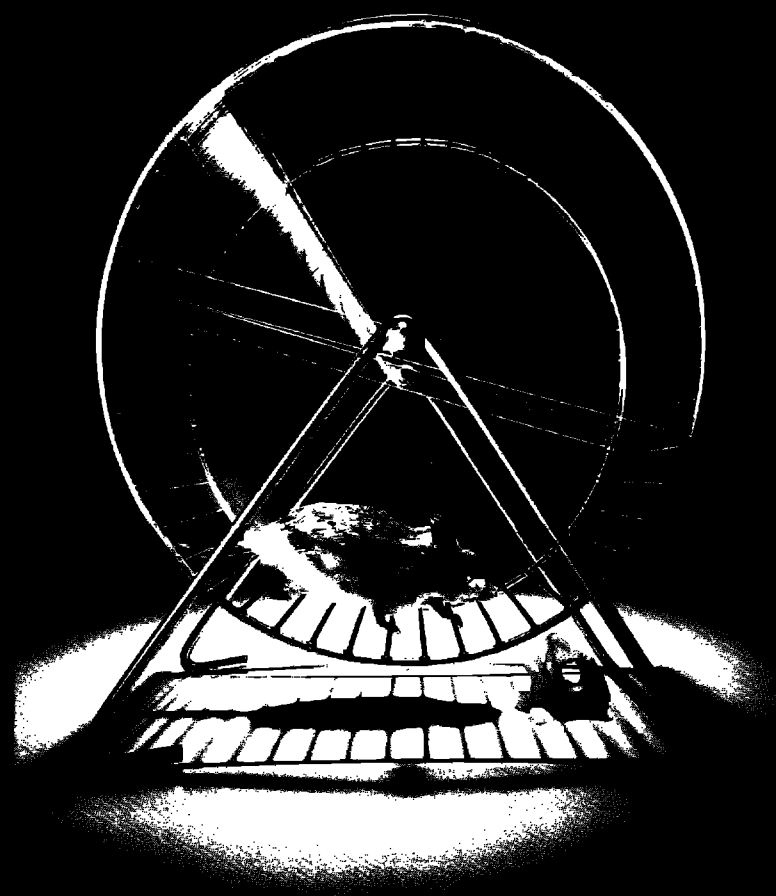
Bottom Line

Behavior is better than data. Smalltalk reuse mechanisms work on behavior.

Because pools are data, avoid pools whenever possible. Instead, create a class that encapsulates the data in the pool and replace existing pool variable references with messages. Your code reuse and ability to store your application in source form will improve. Send your feedback on this discussion to juanita@digitalk.com. ■

Juanita Ewing is a senior staff member of Digitalk Professional Services, 921 SW Washington, Suite 312, Portland, OR 97205, 503.242.0725.

Creating those new client and server applications would be far more rewarding if you could reuse existing code instead of rewriting it. And now that goal becomes reality with object-oriented programming. Especially when you can rely on VisualWorks™, the ParcPlace Smalltalk™ Applications Development Environment that creates applications that are instantly portable between Windows, OS/2, Macintosh and UNIX. True OOP, it provides a robust set of tools to build sophisticated graphical applications with access to a wide variety of relational databases. Fully armed with superior flexibility, dynamic compilation for impressive performance and the world's largest set of tried and tested class libraries, VisualWorks is scalable from enterprise to department and back. Call 1-800-759-7272 ext. 400 for our Solution Pack. You'll see why so many forward-looking Fortune 1000 companies have selected VisualWorks for client and server development. And stopped rewriting history.



VisualWorks

ParcPlace®

DEVELOPERS WHO DO NOT REMEMBER HISTORY ARE CONDEMNED TO REWRITE IT.

CREATING IPF HELP PANELS FOR SMALLTALK/V OS/2 APPLICATION

by Marcos Lam and Susan Mazzara

Part I of this article discussed IPF and some of its requirements that play an important role in linking help panels to a Smalltalk application. Part II explains Digitalk's implementation of its help classes and some enhancements and extensions you can make to them.

Digitalk's Smalltalk/V for OS/2 provides two classes for linking IPF help panels to a Smalltalk application: PMHelpLibraryDLL and HelpManager. PMHelpLibraryDLL wrappers all six APIs provided by the system dynamic link library called HELPMGR.DLL, which provides the interface to IPF. HelpManager models IPF and is the class that Smalltalk applications interact with to access IPF.

As explained in Part I of this article (THE SMALLTALK REPORT, Vol. 3, No.5), IPF interfaces with applications through a help subtable. Help subtables store control IDs (identifiers for menus, menu items, subpanes, and so on) and associate help panel resource IDs with each control. A resource ID is one of the IDs that you tag in a help panel header to identify it for online help requests and internal hypertext or hypergraphic links. When an application receives a help request it passes the control identifier to IPF. IPF looks at the help subtable to determine the corresponding help panel resource ID and then displays the panel tagged with that resource ID.

Chapter 8 of the SMALLTALK/V FOR OS/2 PROGRAMMING REFERENCE, "Help Manager," makes no mention of help subtables.[1] Indeed, HelpManager always passes an empty help subtable to IPF whenever a help instance is to be created. The manual recommends that applications use menu titles, window titles, and menu item selectors as IDs (id=) for help panels. It does not recommend using resource IDs, which IPF expects in order to find the correct help panel. So how does IPF decide which help panel to display when it receives a help request? Since it does not have a help subtable to refer to, it always sends the HM_HELPSUBITEM_NOT_FOUND message in response to a help request. ApplicationWindow >> hmHelpsubitemNotFound:with: uses the two parameters passed in to determine the name of the control, or selector. In the case of a menu item, that you have requested help for. It then sends IPF a message to display the help panel by the same name. This process is illustrated in Figure 1.

TAGGING HELP PANELS

To create help panels that suit this process of interacting with IPF, Digitalk prescribes the following ways of tagging your help panels:

Extended Help

To create an extended help panel for a window or dialog, you give it an instance of HelpManager and then tag the extended help panel with resource ID 1. The code that creates an instance of HelpManager for a view looks like this:

```
HelpManager for: self mainView
    title: 'Help'
    file: 'library.hlp'
```

In the message #for:title:file:, the title is how you place a title on the coverpage for your help panels. The file name points to your compiled help library. The #for:title:file: creation method sends HelpManager the message #for:title:file:dialogs:aboutDlgClass:.

```
for: aWindow title: aString file: aFileName dialogs: aCollection
aboutDlgClass: aboutDlg
    ^self new
        pszTutorialName: ''
        phtHelpTable: nil
        hmodHelpTableModule: nil
        helpWindowTitle: aString
        pszHelpLibraryName: aFileName
        applWindow: aWindow
        aboutDlgClass: aboutDlg
        extHelp: 1
        keysHelp: 2
        dialogs: aCollection.
```

In this message, HelpManager has the value for the extended help panel resource ID hard coded to 1 and the keys help panel resource ID hard coded to 2. To get an extended help panel or keys help panel to work according to Digitalk's requirements, you have to tag them as shown here.

```
:h1 res=1.Panel Title
:h1 res=2.Keys Help
```

Menus and Menu Items

HelpManager's support for IPF bypasses IPF's help subtables. As shown in the previous illustration, when IPF receives a help request for a menu or menu item and cannot find the control ID in a help subtable, IPF notifies the application. The application, in turn, sends IPF an alternate help panel ID. To provide these alternate IDs, you tag help panels for menus and menu items as follows.

Menus:

```
:h1 id='~Menu Title'.Help Panel Title
```

Menu items:

```
:h1 id=menuItemSelector.Help Panel Title
```

Were Digitalk to follow IPF's requirements, these help panels would be tagged as follows.

Menus:

```
:h1 res=2000.Help Panel Title
```

Menu items:

```
:h1 res=2001.Help Panel Title
```

DRAWBACKS OF HelpManager

The advantage of HelpManager's technique of interacting with IPF is that you are not burdened with determining control IDs and passing them to IPF, but the drawbacks are numerous:

- A multiview application can have only one extended help panel, since all resource IDs must be unique.
- Menus and menu items in the same main window must have unique labels.
- A popup menu must have the exact same items as its title bar menu counterpart.
- It is not possible to provide help for popup menus on a window that does not have a title bar.
- This technique conflicts with IPF's requirements for external hypertext and hypergraphic links.

Extended help for multiview applications

HelpManager's implementation of extended help requires you to place the help panels for each window in a separate library because each extended help panel must have the resource ID of 1 and resource IDs within a library must be unique. But suppose you have an application with outboard windows that relate to the main window or control the contents of the main window. In a mail order application, for example, you may have one main window for processing orders. From this window you can open other windows for processing customer information and maintaining your inventory database. Such an application is illustrated in Figure 2.

To provide on-line help for this application, you would have to write three separate help libraries and use external hypertext or hypergraphic links among them to retrieve information. But if a user is working in the order processor, for example, and asks for extended help, he or she cannot display a comprehensive table of contents for the entire application, because the table of contents is compiled within each separate library.

For a multiwindow application, such as this order processing application, having each window documented in a separate help library means not having a single place (such as a table of contents) where users can look for an overview of the tool and see what other tools are a part of it. This restriction makes information retrieval difficult and reduces the usefulness of on-line help systems.

Unique menu names

What if your application needs to reuse menu names? What if the mail order application, for example, has two menus, one called Orders and one called Accounts, and each has a submenu called Open? How do you tag the help panel for the Open submenus (Figure 3)?

According to Digitalk guidelines, you tag the help panel for each menu as follows:

```
:h1 id='~Open'.Opening Orders
:h1 id='~Open'.Opening Accounts
```

When you attempt to compile these help panels, IPF returns a compiler error, because identifiers must be unique. To solve this problem, you must either rename one of the submenus or write one help panel that is vague enough to suffice for both menus. This requirement makes it impossible to maintain consistency in naming conventions in the interface.

Popup menus

When there is a help request for a popup menu or its items, ApplicationWindow >> hmHelpsubitemNotFound:with: uses the identifier of the menu or menu item to determine its counterpart in the title bar menu. The help request is processed as if it were for a menu or menu item in the title bar menu. This approach has the following drawbacks:

- A popup menu must have a counterpart in the title bar menu.
- A popup menu must have the exact same items as its title bar menu counterpart (text pane popup menu).

CUA '91 guidelines provide for windows without title bar menus. If you want to apply these guidelines to your application, it is not possible to provide help for popup menus on a window that does not have a title bar menu. These guidelines also provide for users to customize popup menus.[1] But in Digitalk's implementation of HelpManager, if a user changes a popup menu so that it no longer matches its title bar menu counterpart, you can no longer provide help for it.

Hypertext and Hypergraphic Links

Another drawback to HelpManager's support for menu and menu-item help requests is that it conflicts with external links. To link to a help panel from another help library, you tag the heading with a "global" attribute, as follows:

```
:h1 res=2000 id=2000 global.Help Panel Title
```

When you add this attribute to the help panel for a menu or menu item, IPF enforces its rule that the resource IDs (res=) are to be used for internal access to the help panel and IDs (id=) are to be used for external access. If you try to ask for a global help panel by its ID from any other source besides an external hypertext or hypergraphic link, IPF cannot find the panel.

Since HelpManager bypasses help subtables, which associate application resources with IPF resource IDs, and uses IDs to get help for a menu or menu item, you cannot make both a context-sensitive help request and an external link for a panel.

ENHANCEMENTS AND EXTENSIONS TO HelpManager

Extended help

With a simple extension to HelpManager, you can code a

Listing 1

```
HelpManager variableSubclass: #MyHelpManager
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries:
  'PMHelpConstants PMConstants '   !


!MyHelpManager class methods !

for: aWindow title: aString file: aFileName
  dialogs: aCollection aboutDlgClass: aboutDlg
  extendedHelpPanelID: anInteger1
  keysHelpPanelID: anInteger2

  ^self new
    pszTutorialName: ''
    phtHelpTable: nil
    hmodHelpTableModule: nil
    helpWindowTitle: aString
    pszHelpLibraryName: aFileName
    applWindow: aWindow
    aboutDlgClass: aboutDlg
    extHelp: anInteger1
    keysHelp: anInteger2
    dialogs: aCollection.!

for: aWindow title: aString file: aFileName extendedHelpPanelID: anInteger1
  keysHelpPanelID: anInteger2

  ^self for: aWindow title: aString file: aFileName dialogs: #()
aboutDlgClass: nil extendedHelpPanelID: anInteger1
keysHelpPanelID: anInteger2! !


!MyHelpManager methods !

applicationWindow

  ^applicationWindow!

buildHelpManager

  | a1 a2 helpLibrary helpTables helpTable helpSubTable |
  helpLibrary := PMHelpLibraryDLL open.
  helpLibrary isInteger"An error has occurred."
    ifTrue: [MessageBox inform: 'Library file not found.'
          title: 'PMHelpLibraryDLL'.
        ^self ].
  self pmHelpLibrary: helpLibrary.
  addresses := Set new.
  helpTableModule notNil
    ifTrue: [
      helpInit
        phtHelpTableId: helpTableId;
        hmodHelpTableModule: helpTableModule ]
    ifFalse: [
      helpSubTable := self helpSubtableStructure.
      helpTables := OrderedCollection new.
      helpTables
        add: ((SelfDefinedStructure named: 'HELPTABLE' )
        idAppWindow: (PMWindowLibrary
            queryWindowUShort: applicationWindow frameWindow
              asParameter
                index:
                QwsId );
          phstHelpSubTable: (addresses add:
            (MemoryBlockAddress
              copyToNonSmalltalkMemory:
                helpSubTable asParameter ) ) asParameter;
          idExtPanel: extHelpPanelId ).
      dialogs do: [:id |
        helpTables add: ((SelfDefinedStructure named: 'HELPTABLE' )
            idAppWindow: id;
            phstHelpSubTable: (addresses add:
              (PMAddress
                copyToNonSmalltalkMemory: helpSubTable asParameter) )
                    asParameter;
            idExtPanel: id ) ].
      helpTable := ByteArray new.
      helpTables do: [:ht |
        helpTable := helpTable, ht contents ].
      helpTable := helpTable, ((SelfDefinedStructure
        named: 'HELPTABLE' ) contents ).
      helpInit phtHelpTable: (addresses add:
        MemoryBlockAddress
          copyToNonSmalltalkMemory: helpTable ) ) asParameter ] .
  a1 := MemoryBlockAddress
    copyToNonSmalltalkMemory: helpWindowTitle asParameter .
  a2 := MemoryBlockAddress
    copyToNonSmalltalkMemory: helpLibraryPath asParameter.
  helpInit
    pszHelpWindowTitle: a1 asParameter;
    pszHelpLibraryName: a2 asParameter.
  helpInstance := self pmHelpLibrary
    createHelpInstance: PM hab
    pHelpInit: helpInit asParameter.
  a1 free. a2 free.
  self pmHelpLibrary associateHelpInstance: helpInstance
    hwndApp: applicationWindow handle.!

destroyHelpInstance

  pmHelpLibrary notNil
    ifTrue: [
    pmHelpLibrary destroyHelpInstance: helpInstance.
    pmHelpLibrary close.
    addresses do: [:a | a free] ].
  helpInstance := pmHelpLibrary := nil. !

displayHelp: id

  ^id isString
    ifTrue: [(PMWindowLibrary sendMsg: helpInstance
        msg: HmDisplayHelp
        mp1Struct: id asParameter
        mp2: HmPanelname ) asPMLong asBoolean ]
    ifFalse: [(PMWindowLibrary sendMsg: helpInstance
        msg: HmDisplayHelp
        mp1: id
        mp2: HmpaneltypeNumber ) asPMLong asBoolean ] !

helpSubtable

  | answer helpSubtable |
  answer := OrderedCollection with: 2.
  helpSubtable := (self applicationWindow respondsTo: #helpSubtable)
      ifTrue:  [self applicationWindow helpSubtable]
      ifFalse: [IdentityDictionary new].

  helpSubtable associationsDo: [:association |
    answer
      add: association key;
      add: association value ].
  ^answer
    add: 0;
    add: 0;
    yourself!

helpSubtableStructure

  | answer helpSubtable |
  helpSubtable := self helpSubtable.
  answer := PMStructure new: helpSubtable size * 2.
```

```
  1 to: helpSubtable size do: [:index |
    answer uShortAtOffset: (index - 1) * 2
      put: (helpSubtable at: index) ].
  ^answer!

pmHelpLibrary

  ^pmHelpLibrary!

pmHelpLibrary: aPMHelpLibraryDLL

  pmHelpLibrary := aPMHelpLibraryDLL! !

!ViewManager methods !
helpSubtable

  ^Dictionary new!

provideOnlineHelp
  Provide online help. Default is to do nothing. Typically subclasses over-
ride this to instantiate HelpManager."
! !



!TopPane methods !
helpSubtable

  ^self owner helpSubtable! !
```

Listing 2

```
!ApplicationWindow methods !
allPopupMenus

  ^(self allChildren collect: [:subPane | subPane popupMenu ] )
      select: [:each | each notNil]! !


!Menu methods !
buildPopupIn: aWindow

  popup isNil
    ifTrue: [
      popup := MenuWindow new.
      ((aWindow mainWindow menuWindow allMenus size 2 )
        + ((aWindow mainWindow allPopupMenus
          indexOf: (aWindow mainWindow allPopupMenus
            detect: [:menu | menu title = self title]
            ifNone: [] ) ) * 4 ) )
          timesRepeat: [popup allMenus
                    add: (self class new title: 'Dummy'; yourself ) ].
      popup addMenu: self.
      popup buildWindow: aWindow ]!

isDummy

  ^self title = 'Dummy'!

popUpAt: aPoint in: aWindow
  "Popup the receiver menu at aPoint in aWindow. Answer the action bar
MenuWindow."

  self buildPopupIn: aWindow.
  popup popUpAt: aPoint in: aWindow.
  ^popup! !


!MenuWindow methods !
allMenus

  ^allMenus! !
```

Listing 3

```
!TopPane methods !
allPopups

  ^(self allChildren collect: [:subPane | subPane popup ] )
      select: [:each | each notNil] !

getMenuItemID: aString

  | isPopup menuItemName menuNames readStream spec
  topLevelMenu |
  readStream := ReadStream on: aString.
  isPopup := readStream peek == $^
    ifTrue: [readStream upTo: $^.
      true ]
    ifFalse: [false].
  menuNames := (readStream upTo: $*)
    asArrayOfSubstringsDelimitedBy: $>.
  menuItemName := readStream upToEnd.
  topLevelMenu := isPopup
    ifTrue: [self allPopups
          detect: [:each | each title = menuNames first] ifNone: [] ]
    ifFalse: [self menuTitled: menuNames first].

  ^topLevelMenu isNil
    ifTrue: [-1]
    ifFalse: [spec := menuNames asOrderedCollection
      removeFirst; yourself.
      menuItemName notEmpty
        ifTrue: [spec add: menuItemName].
      spec isEmpty
        ifTrue: [topLevelMenu id]
        ifFalse: [topLevelMenu getItemIDFromFullSpec: spec ] ] ! !

!Menu methods !
getItemFromFullSpec: aCollection

  | index label submenu |
  label := aCollection first.
  index := self getIndex: label ifAbsent: [].

  ^index isNil
    ifTrue: [self hasSubMenu
      ifTrue: [
        submenu := self subMenus
          detect: [:each | each title = label] ifNone: [].
        submenu isNil
          ifTrue: [nil]
          ifFalse: [aCollection size == 1
            ifTrue: [submenu]
            ifFalse: [submenu getItemFromFullSpec:
              (aCollection copyFrom: 2 to: aCollection size ) ] ] ]
      ifFalse: [nil] ]
    ifFalse: [aCollection size == 1
      ifTrue: [items at: index]
      ifFalse: [(selectors at: index) getItemFromFullSpec:
        (aCollection copyFrom: 2 to: aCollection size) ] ] !


getItemIDFromFullSpec: specification

  | item |
  item := self getItemFromFullSpec: (specification isString
    ifTrue: [specification asArrayOfSubstrings]
    ifFalse: [specification] ).

  ^item isNil
    ifTrue: [-1]
    ifFalse: [item id] ! !
```

specific help panel resource ID when you create an instance of HelpManager for a window or dialog. The extension, shown in Listing 1, alters the HelpManager creation method #for:title:file:dialogs:aboutDlgClass: to add two new arguments: the extended help panel resource ID and the keys help panel resource ID.

This method is sent to HelpManager by the #for:title:file:extendedHelpPanelID:KeysHelpPanelID: method, shown below:

```
HelpManager
        for: self mainView
        title: 'Help Library Title'
        file: 'library.hlp'
        extendedHelpPanelID: 2000
        keysHelpPanelID: 2010
```

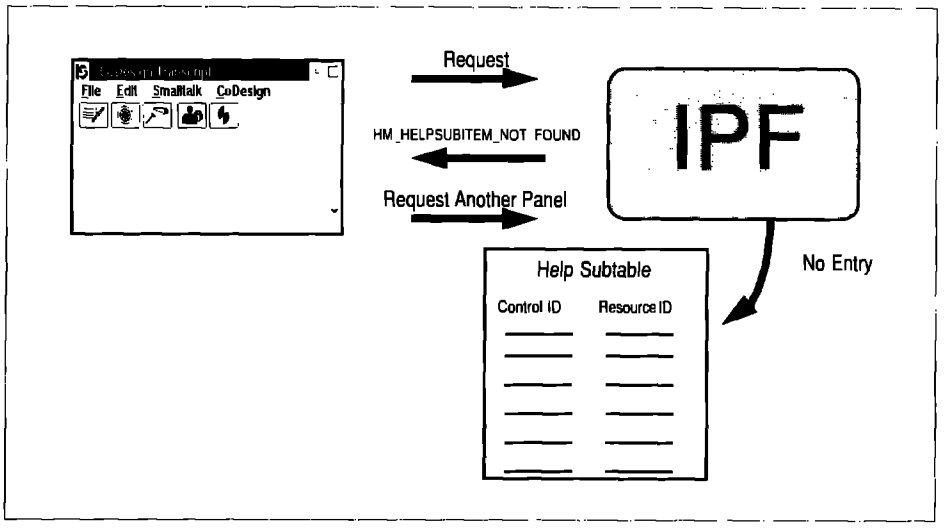


Figure 1. Processing help requests with an empty help subtable.

By allowing you to code the resource ID of your extended help panel and keys help panel, you can create as many of them as you need and place all help panels for an application in a single help library. This enables you to have a single source for online help with a comprehensive table of contents. With this extension, each window in the mail order application, for example, can be compiled in a single help library so that you can see, simply by looking at the table of contents, what kind of information is available for each tool.

Help subtables

Sending a help subtable to IPF to find help panels for menus and menu items can solve the problem that HelpManager has with external hypertext links and the problem with reusing menu names. The following is an example of a help subtable for the mail order application menus previously illustrated:

```
^IdentityDictionary new
        at: 512 put: 2000;
        at: 768 put: 2001;
        at: 769 put: 2002;
        at: 770 put: 2003;
        at: 1024 put: 2004;
        at: 1280 put: 2005;
        at: 1281 put: 2006;
        at: 1282 put: 2007;
        yourself
```

In this IdentityDictionary, the index is the unique ID for each menu or menu item. The value is the resource ID for the help panel that explains the menu or menu item. Using this help subtable, you can tag the help panels for these menus as follows:

```
:h1 res=2000.The Orders Menu
:h1 res=2001.Opening Orders
:h1 res=2002.Opening New Orders
:h1 res=2003.Opening Back Orders
:h1 res=2004.The Accounts Menu
:h1 res=2005.Opening Accounts
:h1 res=2006.Opening New Accounts
:h1 res=2007.Opening Current Accounts
```

Since this technique does not interfere with IPF external hypertext links, you can also code each of these help panels with the ID and global attributes to enable other help libraries to link to them, as shown in this example:

```
:h1 res=2007 id=2007 global.Opening Current Accounts
```

We have created a subclass of HelpManager, called MyHelpManager, and extended ViewManager and TopPane to support help subtables. The code for these extensions is listed in Listing 1.

Help for Popup Menus

Digitalk assigns all popup menus the ID 513 (see Menu >> buildWindow:). As a result, whether you use a help subtable or Digitalk's approach to providing help, you cannot provide correct help for two or more popup menus within the same window. There are several possible solutions to this problem and each is discussed with its advantages and disadvantages in the following sections:

- Guarantee that the popup menu has the same ID as the corresponding menu on the title bar menu.
- Guarantee that all popup and title bar menus have unique IDs.
- Calculate menu and menu item IDs automatically.

Matching IDs Since a popup menu often corresponds to one of the menus in the title bar menu, one way to solve the problem is to make the popup menu and its items have the same IDs as their counterparts in the title bar menu. However, as mentioned previously, you may not always want a popup menu to correspond exactly to a title bar menu or you may not want to include a title bar menu at all.

Unique IDs Another approach is to guarantee unique IDs for

all popup menus and their submenus within the same main window. To guarantee unique IDs, We have modified the method Menu >> popUpAt:in: and written four new methods, shown in Listing 2. This code represents a noninvasive solution to this problem. Although it may not be technically elegant, it works well with minimum changes to the base image.

Calculating IDs To manually calculate the ID of menu and menu items to build the help subtable is tedious and error prone. TopPane and Menu are extended to accept a string which specifies a menu or menu item and answers its ID. Four methods have been added to automatically calculate menu item IDs and are shown in Listing 3. This approach works for pull-down and pop-up menus. In the sample application illustrated in Figure 3, you can create a dictionary as follows:

```
^Dictionary new
        at: 'Orders' put: 2000;
        at: 'Orders>Open' put: 2001;
        at: 'Orders>Open*New' put: 2002;
        at: 'Orders>Open*Back Orders' put: 2003;
        at: 'Accounts' put: 2004;
        at: 'Accounts>Open' put: 2005;
        at: 'Accounts>Open*New' put: 2006;
        at: 'Accounts>Open*Current' put: 2007;
        at: '^Orders' put: 2000;
        at: '^Orders>Open' put: 2001;
        at: '^Orders>Open*New' put: 2002;
        at: '^Orders>Open*Back Orders' put: 2003;
        at: '^Accounts' put: 2004;
        at: '^Accounts>Open' put: 2005;
        at: '^Accounts>Open*New' put: 2006;
        at: '^Accounts>Open*Current' put: 2007;
        yourself
```

The strings that include a carat define popup menus and their submenus and menu items.

Using this approach you do not have to calculate the menu or menu item IDs. If a menu item is repositioned within the same menu, there is no need to reconstruct the help subtable. This solution gives you the best of both worlds: it conforms to IPF's requirements to use a help subtable to associate an application resource with a help panel, and it relieves you of the burden of calculating control IDs, which is one of the goals of Digitalk's implementation of HelpManager.

Destroying help instances

As explained in part 1 of this article, an application must destroy help instances upon closing windows Smalltalk/V for OS/2 does not destroy help instances. The method ApplicationWindow >> closeView is supposed to destroy its help instance. ApplicationWindow keeps its help instance in the properties dictionary. In the middle of closeView, ApplicationWindow sends the message *super close* to Window. Among the things that Window >> close does is to nil the properties dictionary. Since the message to destroy the help instance is at the end of closeView, by the time it executes, it can no longer access the help instance. As a result, it does not destroy its help instance. You can address this problem by moving the code that destroys the help instance to execute before the message *super close*.

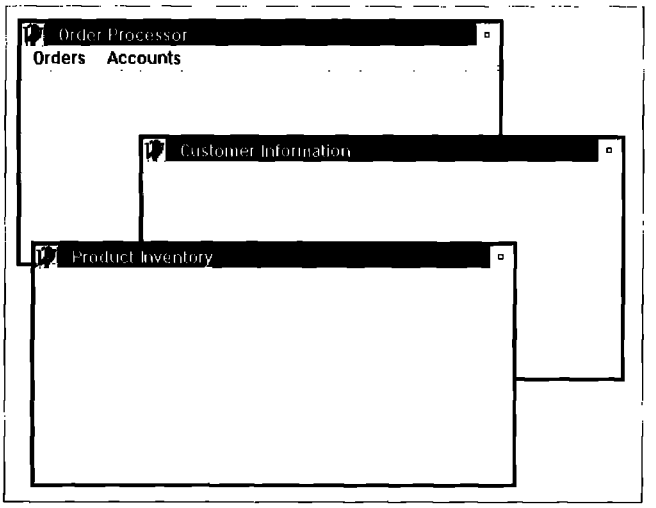


Figure 2. Example of a multiwindow application.

HELPFUL HELP

The enhancements described here can make your on-line help panels easier to retrieve and easier to navigate through. Of course, no amount of programming can guarantee well-written help, but by making it easier to obtain context-sensitive help and create hypertext and hypergraphic links among help panels, you can place more information before your users. The more information you can give users about your application, the more you can reduce the frustration that leads them to seek help in the first place. ■

References

1. SMALLTALK V FOR OS/2 PROGRAMMING REFERENCE, Digitalk Incorporated, 1992, p. 161.
2. COMMON USER ACCESS ADVANCED INTERFACE DESIGN REFERENCE (SC34-4290-00) (1991). International Business Machines Corporation, First Edition, Armonk, NY.

Marcos Lam is a member of the technical staff, and Susan Mazzara is a technical writer at Knowledge Systems Corporation. Both have worked on the development team for CoDesign, a Smalltalk development environment that integrates design and code. They can be contacted at Knowledge Systems Corporation, 114 MacKenan Drive, Cary, NC 27511, 919.481.4000.

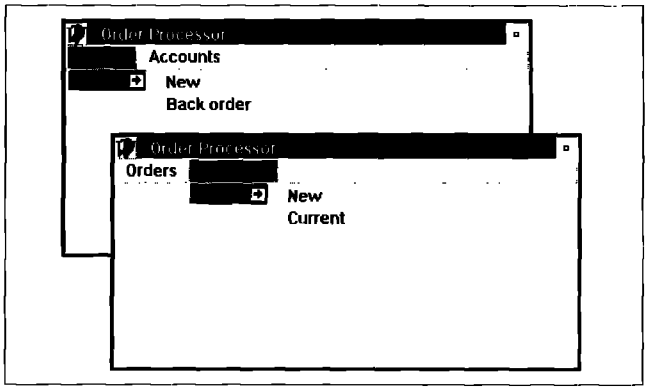
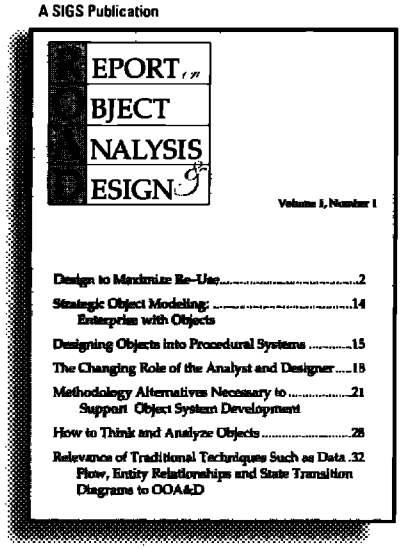


Figure 3. Duplicate submenu names.

New from SIGS in May '94



Report on Object Analysis & Design

Your first step for planning and building object–based software systems.

ROAD's editor, Dr. Richard Wiener, has assembled the leading international authorities as regular columnists. Regular columns, written by the very creators of the most popular methods, include...


Grady Booch
The Booch Method


Ed Yourdon
Objects


Peter Coad
Object-Oriented Analysis


Derek Coleman
Mechanics


James Coplien
Multiparadigm Design Techniques


Don Firesmith
Views on Modeling


Ivar Jacobson
More than just Object Orientation


Meilir Page-Jones
Object-Oriented Design Notation


Tsvi Bar-David
Formal Methods


Sally Schlaer
Methods & Architectures


Rebecca Wirfs-Brock
Pragmatics


Steve Mellor
Methods & Architectures


Ian Graham
Rapid Application Development

ALSO...
Ralph Johnson
Patterns of Thought
Patrick Ryan
Practical Experiences
Nancy Wilkenson
An Informal Approach

The Report on Object Analysis & Design **(ROAD) is a new bi–monthly (6x), advertising–free journal, which focuses on language–independent, architectural concerns about object–oriented analysis, design and modeling. Each issue provides you with in-depth articles addressing the complex questions related to the system architecture prior to when language issues are addressed, including...**

- **the fundamental issues related to object modeling**
- **notational schemes for representing A&D models**
- **the processes for performing OOA or OOD**
- **revisions and updates of various design methods**
- **comprehensive comparisons of OOA&D approaches**
- **specifications on which method to use, and when**
- **expert reports on the tools currently available**

And much more.

REPORT on OBJECT ANALYSIS & DESIGN

For software developers and project leaders either currently working on an OT project, or moving toward that goal.

Platform and system independent, ROAD is written for all levels of project complexity.

clip and mail or fax

Yes—Enter my subscription to ROAD at the pre-publication rate marked below:

❑ **Individual 1 Year** (6 issues) ~~$99~~ **$74**
A SAVINGS OF 25% OFF THE BASIC INDIVIDUAL RATE!

❑ **Institutional 1 Year** (6 issues) ~~$199~~ **$174**
YOU SAVE $25!

METHOD OF PAYMENT

❑ Check enclosed, payable to **ROAD**
(in U.S. dollars, drawn on a U.S. bank)

❑ Charge my
❑ Visa ❑ MasterCard ❑ AmEx

ACCT#_____

EXP _____

SIGNATURE _____

PRE-PUBLICATION OFFER EXPIRES MAY 1, 1994

This is the lowest rate offered!

Name _____

Title _____

Address _____

City _____

State/Province _____

Zip/Postal Code _____

Country _____

P:_____ F: _____

RETURN TO
ROAD, P.O. Box 2027,
Langhorne, PA 19047-9027
Fax: 215-785-6073 Phone: 215-785-5996

Important: For non-U.S. orders, please add $30 for air service. The basic 1-year individual rate is $99, institutional rate is $199. Offer expires May 1, 1994.

Save up to 25%!

To start your subscription to ROAD at these *special pre-publication discounts*, mail or fax this coupon by May 1, 1994!

Kent Beck

Where do objects come from? Part 2

PREVOUSLY, I talked about how objects could be created from the states of objects that acted like finite state machines (the Objects from States pattern). I'll continue on the theme of where objects come from for this and several issues.

I won't be saying much about the conventional source of objects, the user's world. There are lots of books that will tell you how to find those objects. Instead, I'll focus on finding new objects in running programs.

In all programming, many of the most important design insights don't come until after a program has been deployed for awhile. Smalltalk is unique in that it is possible to fold those insights back into the program. Polymorphism, in particular, is invaluable for introducing new objects without disturbing existing ones. Unlike programs written in more conventional languages, Smalltalk programs can get better and better, and easier to extend over time. Such programs tend to spin-off reusable pieces, as well, which multiplies their value.

Ward Cunningham is a pioneer of this technique, which he calls *episodic design*. In an episodic design process, design doesn't happen all at once, as in the barnacle-encrusted waterfall model. Instead, design happens in episodes, whenever you understand an issue well enough to know that your previous design is limited in some way.

To avoid overdesigning, design episodes are typically triggered by the desire to add a new feature. Some features seem to slide right in with little effort. Others must be forced in at the cost of violating good design. When encountering the latter, an episodic designer will first "make a place" for the feature by fixing the design so it's easy to add.

Design episodes typically consist of finding new objects, new responsibilities, or new collaborations. New objects often come about to add degrees of freedom to your program. For example, you may have thought initially that interest calculation was a simple computation, so it was buried in a method in FinancialInstrument. In adding new functionality, you realize there are many different ways to calculate interest, so you need an InterestCalculator, which a FinancialInstrument collaborates with to compute interest. Then you can add new InterestCalculators without disturbing the rest of the design.

When creating new objects, you might think a flash of insight is required to discover them. Not so. While some objects come out of the blue, most can be found in the program itself. The next couple of columns will explore where you can find some of these derived objects. This month's pattern helps you find objects you just didn't quite want to create when you found them in the first place. The programming "convenience" it represents is particularly common in former LISP programmers, but I've seen it come from C and assembly language hacks, too.

PATTERN: OBJECTS FROM COLLECTIONS

Problem

Collections where two or more methods in the same or different objects have to agree on a fixed set of indexes are a maintenance headache (the same observation applies to Associations or Points being used to represent a duple). The programming environment doesn't help you find where all these implicitly meaningful indexes are used. If you have such a collection, how can you make it easy to maintain?

Constraints

- ***Simplicity.*** The reason such collections arise in the first place is because creating classes is a fairly heavyweight activity. You have to find the right name (System of Names) for the class, then you have to find the right name for the messages, then you have use the programming environment to define it. Once it's there, you have to document and maintain it. Where you can't imagine the object being used anywhere else, like returning two values from a method, you aren't likely to bother.
- ***Readability.*** The problem with simply using collections instead of an object is that even in the small it fails to convey the intent of the code. A good example from the VisualWorks 1.0 image is Browser>spawnEdits:from:. It creates a three-element array with the text to edit, the start of the selection, and the end of the selection. This array gets passed through two intermediate methods before it is finally torn apart in Browser class>setTextView:fromTextState: and turned into messages for the newly created text editor. Reading the code, the only clue you have to the contents of the array is the names of the temporaries in the latter method.
- ***Maintainability.*** Closely allied with readability is the issue of how hard the code is to maintain. If I wanted to add a

fourth element to the array in the above example, perhaps for a special font for the selection, I would probably go to Browser>spawnEdits:from: and Browserclass>setTextView:from-TextState: and make the change. However, this would break the debugger, which also spawns edits. This hidden multiple update problem is the best reason for making collections into objects.

Solution

Create a new class. Give it the same number of instance variables as the size of the collection. Name the variables according to what goes in them.

Example

To simplify the above example, let's say you wanted to be able to spawn a text editor. TextEditor has a method textState:, which takes as a parameter a three-element array:

```
TextEditor>textState: anArray
    self text: (anArray at: 1).
    self selectFrom: (anArray at: 2) to: (anArray at: 3)
```

Our browser uses this method:

```
MyBrowser>spawnEdits
    | array |
    array := Array
        with: self text
        with: self selectionStart
        with: self selectionStop.
    TextEditor open textState: array
```

Both methods are now vulnerable to change in the other. By creating an object from the collection, we solve this problem:

```
TextState
    variables: text selectionStart selectionStop

TextState class>text: aString selectFrom: startInteger to: stopInteger
    ^self new
        setText: aString
        selectionStart: startInteger
        selectionStop: stopInteger

TextState>setText: aString selectionStart: startInteger selectionStop:
stopInteger
    text := aString.
    selectionStart := startInteger.
    selectionStop := stopInteger
```

Then we can use a TextState in the TextEditor:

```
TextEditor>textState: aTextState
    self text: aTextState text.
    self selectFrom: aTextState selectionStart to: aTextState
selectionStop
```

And create it in MyBrowser:

```
MyBrowser>textState
    ^TextState
        text: self text
        selectionStart: self selectionStart
        selectionStop: self selectionStop

MyBrowser>spawnEdits
    TextEditor open textState: self textState
```

The result is code that is slightly more complicated, but much easier to read and maintain. The beauty of Objects from Collections is not just in the immediate results. The new objects often become the home of important behavior in their own right. Code that lived uneasily in one of the objects that understood the format of the array can now live comfortably in the new object. Also, the new object becomes a new degree of freedom in the system. If there are a variety of ways the information can be structured or used, you can capture that variety in a family of objects all responding to the new object's protocol.

In the next issue, we will examine two more patterns for creating objects from code: Objects from Variables and Objects from Methods. ■

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

THE BEST OF comp.lang.smalltalk

Alan Knight

Net resources

This month we move away a little from USENET discussions and talk about some of the Smalltalk resources that you can access from your computer. There are many such resources, and they may be available over ftp, email, or direct modem connections.

VENDOR RESOURCES

Software companies usually provide some sort of on-line support. This comes in a variety of forms, and is normally described in the documentation. Most of the Smalltalk vendors have some sort of email access.

Digitalk has a vendor forum on CompuServe, which is where most of their support activity seems to take place. They do have Internet email, though, and can be reached at info@digitalk.com.

ParcPlace has a bulletin board called ParcBench, which can be reached at 415.691.6716. They also have Internet email, and can be reached at info@parcplace.com.

QKS, makers of SmalltalkAgents, can be reached at info@qks.com.

FTP SITES

I discussed anonymous ftp archive sites for Smalltalk in my very first column, but that was almost two years ago, so I think it's worth repeating the information for new readers. Anonymous ftp means that you connect through ftp and use "anonymous" as a username and your email address as a password.

Two main archives have large amounts of Smalltalk code.

- st.cs.uiuc.edu (University of Illinois, Internet ID 128.174.241.10)
- mushroom.cs.man.ac.uk (University of Manchester, Internet ID 130.88.13.70)

These two sites have a "mirroring" arrangement, so they both have exactly the same files available, although they are organized a bit differently. Transfers will probably be faster if you use the site closest to you.

As with any freely distributed software, be careful of copyright restrictions, particularly "copyleft" if you plan on distributing this as part of commercial code. Files collected by the Manchester library (which is a subset of those available from the Manchester site) are often copylefted unless otherwise specified.

Other sites

A number of other sites also have files of interest to Smalltalkers.

- GNU Smalltalk (see THE SMALLTALK REPORT 1[8]) is available from most of the many archives that carry GNU software. The reference site is prep.ai.mit.edu, but that site is heavily loaded, so it's best to look elsewhere.
- A Little Smalltalk is Tim Budd's UNIX implementation of a subset of Smalltalk. It's available from cs.orst.edu (or by an email server that can be reached by sending a message with the text "send guide" to almanac@cs.orst.edu). A Little Smalltalk is described in the Addison-Wesley book of the same name. An extended version, which includes graphical support for the X Window System, is available from beach.rockwell.com.
- The comp.lang.smalltalk frequently asked questions (FAQ) list and an archive of discussions from the smallmusic mailing list is available from xcf.berkeley.edu.
- QKS has recently started an ftp site for Smalltalk Agents material at pineapple.qks.com (192.55.204.66).

email

ftp sites are very convenient if you have ftp access, but most of the world doesn't. Those with Internet email access (note that many commercial on-line services can send and receive Internet mail), can use the Manchester and Illinois Smalltalk archives through email servers.

To reach the Manchester archives, send a message of the form:

```
To: goodies-lib@cs.man.ac.uk
Subject: help;index
```

You should receive an explanation of how to use the archives and a fairly informative listing of what's available along with a summary. If you need to communicate with a human, try lib-manager@cs.man.ac.uk.

To access the University of Illinois archives, send:

```
To: archive-server@st.cs.uiuc.edu
Subject: <Doesn't matter>
path yourname@your.Internet.address
help
```

This will send a help file. To also receive a complete listing, send

```
To: archive-server@st.cs.uiuc.edu
Subject: <Doesn't matter>
archiver shar
encoder uuencode
help
encodedsend ls-lR.Z
```

The listing doesn't have the nice explanations you get from the Manchester archive (although the Manchester catalogue is available as one of the files). The listing will also come as a uuencoded, compressed file.

SMALLTALK MAILING LIST

One of the most valuable network resources is comp.lang.-smalltalk itself. There's a lot of junk to sort through, but there's also a large community of knowledgeable Smalltalk users, many of whom normally charge a lot of money for the kind of advice they give over USENET.

> "I'd like to be able to point you to some good file dialog code (for Parc Place), but I don't know of any."

Unfortunately, not everyone can receive USENET news. For those who would like to, but who have only email access, there is a mailing list that echoes postings in comp.lang.smalltalk. The list is maintained by Joerg Rade (jrade1@gwdg.de), and described as follows:

> **Info-CLS (formerly Smalk) is a mailing list which is bidirectionally gatewayed with comp.lang.smalltalk (via NET-NEWS@AUVM.BitNet). Every posting to c.l.s (with distribution options usa or world) gets distributed to all subscribers of Info-CLS, and vice versa every mailing to Info-CLS gets posted to c.l.s.**
>
> **In order to get subscribed, send a mail message to the listserv (LISTSERV@vm.gmd.de), containing SUBSCRIBE Info-CLS name@node.net.world Your F. Name or drop me a note.**

A knowledgeable group of Smalltalk users is a wonderful thing, but the delay involved in getting an answer can be very painful. This is particularly true when the question has been asked before and you could get an answer very quickly if you only had an archive of old messages to search through. Luckily, there is such an archive:

There is a mailarchive associated with Info-CLS; that is, every posting/mailing gets archived and can be keyword searched by email. To search the archive, which started somewhere around September '92, send a mail message to

```
LISTSERV@vm.gmd.de
```

containing something like:

```
//
Database Search DD=Rules
//Rules DD *
Search type & checking in INFO-CLS
index
print
/*
```

as the body part of the message. For details on more fine-grained retrievals, read the document that is obtainable by sending

```
GET LISTDB MEMO
```

as the body part of a mail message to the listserv.

A FEW GOODIES

That's all the general network resources that I'm aware of at the moment. If you know of others that you think should be mentioned, please tell me about it. In the remaining space, I'll briefly mention some of the Smalltalk source code available from the Manchester and Illinois Smalltalk archives. These bits of code are often referred to as "goodies." As with all freely available code, you need to exercise some care. The code may be of very good quality, or it may not. It is almost certainly not maintained, so it may be out of date. In most cases, it will not have been tested on a very wide variety of different machines. Basically, you will probably want somebody who knows what they're doing to look the code over. You also need to beware of copyright restrictions, as I mentioned in the ftp section.

Object debugging

This is described in great detail in the July-August 1993 issue of THE SMALLTALK REPORT (Debugging Objects, by Bob Hinkle, Vicki Jones and Ralph E. Johnson 2[9]), so I won't say much except to recommend it. It works with ParcPlace Smalltalk and uses instance-specific behavior to provide enhanced debugging facilities. In particular, it lets you set breakpoints (real breakpoints) on methods in individual objects (e.g., halt when **add:** is invoked for *this particular* OrderedCollection). This is really, really nice. It does make some very deep system changes, which is scary, but I think it's well worth it. To my mind, the biggest drawback is that it doesn't coexist well with ENVY/Developer (as of version 1.42). I have an idea of how to make it work, but it's pretty sneaky, and I'm not using ENVY right now, so I haven't tested it.

FileNavigator

It's very annoying that ParcPlace Smalltalk (including VisualWorks) still doesn't have any kind of a file selection dialog. We have a system that is supposedly good for writing graphical user interfaces, and the standard way to get a file name is still

```
DialogView request: 'Enter file name and path'.
```

Forget using shortcuts like "~".

continued on page 21

PRODUCT REVIEW

Jeff Cantwell & Douglas Camp

Reportoire

Smalltalk is a highly productive development environment that excels at object modeling—building the underlying 'object-model' (the M in MVC) part of applications. However, Smalltalk also has some definite low productivity shortcomings. For example, until Smalltalk GUI builders arrived, it was often true that you could build the model portion of an application at a blindingly fast rate then spent countless tedious hours hand-coding your interface. Other problems remain. What is needed is for third-party tool developers to step in and produce extensions to the ST environment that address these problems.

Fortunately, although the number of third-party tools now available for VisualWorks is fairly small, some exciting new products are beginning to appear. If you're on the ParcPlace mailing list, you probably recently received a large envelope auspiciously labeled "Essential Extensions For VisualWorks". Inside was a brochure describing *Reportoire*, a new query and reporting tool from Synergistic Solutions, Inc. You may also have learned about Reportoire at OOPSLA '93. We've been part of the Reportoire beta test group and recently began using release 1.0. In this article we'll outline what Reportoire is, what you can do with it, and briefly describe some of our experiences using it.

WHAT IS REPORTOIRE?

Reportoire is a set of ObjectWorks applications for doing database queries and creating reports. Using Reportoire, developers can quickly and easily create queries and reports on any data which can be accessed from within VisualWorks. The Reportoire toolset is extensive, including several database browsers (Schema, QBE (Query-By-Example) and Free Form browsers), tools for defining data sources and filters, a WYSIWYG report designer, a callable runtime reporting engine, and a 'librarian' application for managing access to all this functionality in a multi-user environment. Reportoire tools are accessed via an iconic launcher, shown in Figure 1.

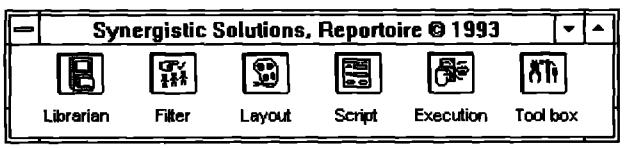


Figure 1. Reportoire's launcher.

FEATURES

One of the first things you notice about Reportoire is that it contains *a lot* of functionality. The User's Manual is just over 400 pages, and the full-blown Development version of Reportoire contains more than 400 new classes, more than 9000 new methods, and increases the size of an Envy image by about 2.5M, and about 1.8M for a standalone VisualWorks image. (Note that these numbers are for the full-blown Development version. Reportoire is available in three flavors: Development, Standard, and Embedded Execution.). Some of Reportoire's major features:

- Full integration with VisualWorks. Standalone image and ENVY versions available.
- Concurrent access to any accessible data sources, including SQL relational databases, object-oriented databases, PC databases via ODBC, and any Smalltalk object in the image. (Some drivers must be purchased separately.)
- An excellent WYSIWYG report designer which supports different text sizes and fonts, event driven formatting and a sophisticated macro language for formatting and control.
- Access control provided by the Librarian tool through username and password restrictions.
- Reuse is encouraged—all reporting elements can be stored in a central repository and reused.
- Device independent output—print drivers are included for Postscript, Windows Metafile, HP PCL and ASCII output. (This feature alone could be an excellent third party add on. You mean I don't have to have a postscript printer? WOW!)

WHAT DOES REPORTOIRE DO?

To get a feel for how developers actually use Reportoire, let's work through the design of a simple report to show which of our customers have balances which exceed their credit limits.

Before a report can be created, a data source must be defined. The source is the connection to the raw data provided by the database driver. Note that this is a one time process—once a source is defined, the Reportoire user can reuse it in other reports. Defining a source is a simple process—popup a short dialog and supply a name for the component, a password

and username for the database (if required), and select the driver type (e.g. Sybase) from a list.

After the Reportoire environment is configured (creating users, defining sources, etc.), creating a report in Reportoire is a 3-step process that creates discrete, reusable components: develop any scripts necessary to prompt users for variable data at runtime, develop filters for selecting only the data of interest, and define the report layout. (Many times only the report layout will need to be created because the other components may already exist and can be reused.) When these steps are complete, the report can be executed at any time—from the Reportoire environment, or from your VisualWorks application code.

1. *Script(s).* Scripts are primarily used as short dialogs to prompt users for variable values at runtime. Clicking the Scripts icon on the launcher brings up the Script creation tool. Defining a script is a simple, fill-in-the-blanks process: name the component, provide variable values and prompts, and select the variable type (e.g. String, Boolean, Date). For this simple example we don't need any scripts.

2. *Filter(s).* Filters determine which data is selected from the data sources. Filters can use SQL for databases which support it (e.g. Sybase, Oracle, ODBC); filters can also use Smalltalk code, the Reportoire macro language, or custom data source specific language. For example, a filter to retrieve all the customers whose balance exceeds their credit limit (for an SQL data source) might look like Figure 2.

3. *Layout.* The Layout Designer is where Reportoire really shines. This tool uses a "spreadsheet metaphor"—the design space is broken into columns and rows, and individual elements of the report are added much as you would type data into a spreadsheet. Once added to the layout, you can drag elements around with the mouse to fine tune the look of the report. A layout for our customer report might look like Figure 3.

Different fonts, typestyles and sizes can be applied to each element of the report. All the functionality you'd expect to find in a good report designer is here, including header/footer sections, multilevel grouping, conditional formatting, and a macro language. An especially nice feature is the ability to define a "row variable" as a Smalltalk object (done in the filter definition). This means that instead of working with report data as simple relational rows, the actual ST object (and all of its behaviors) are available to the report at runtime.

Once the layout is started, reports can be executed and the results viewed immediately from the layout designer—this facilitates easy, incremental report testing and design. Reports can also be executed from Smalltalk code, meaning you can embed calls to the report engine anywhere in your application. It's a simple process in VisualWorks to create a new canvas, paint a button on it, and call the report engine in the button action method.

The Layout Designer is the key feature of Reportoire. This tool alone enormously simplifies the often tedious and time-consuming task of report design.

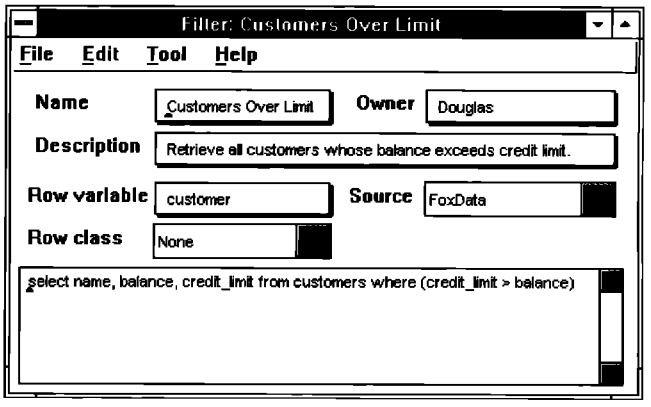


Figure 2. An example of a Reportoire Filter.

EXPERIENCE REPORT

We've used Reportoire to create several custom Smalltalk class reports for use as development aids, and to prototype reports for our major application (a financial/banking system). We've used the Smalltalk image, ODBC, BOSS and ASCII files as data sources.

Overall, our experiences with Reportoire have been very positive, and we plan to use Reportoire for the reporting component of our VisualWorks applications. However, there are some problems. Unexplained walkbacks do occur occasionally—in a development environment, or for strictly in-house systems, this isn't a major problem (what developer isn't used to seeing walkbacks?). However, if you planned to deploy applications with Reportoire embedded, it might be a concern. Many of the problems we encounter are a function of the complexity of the environment: When you layer Reportoire on VisualWorks on Envy, small things (like a class variable not getting properly initialized) can cause annoying, time consuming problems. Other problems and concerns with this release:

- Installation can be difficult. Again, this is a function of the environment (VisualWorks/Envy). We've spent many hours reinstalling the application, trying to resolve obscure load problems.

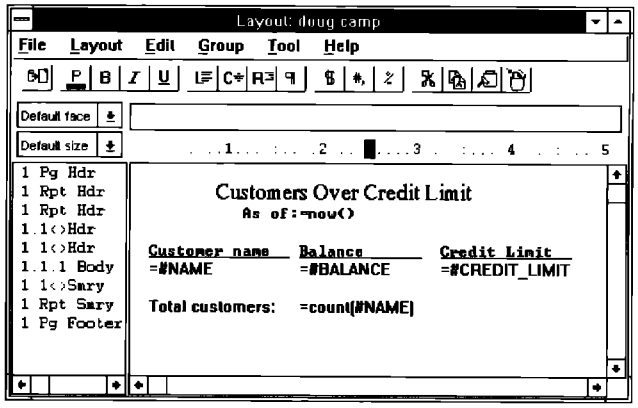


Figure 3. An example of a Reportoire Report.

- Some features simply don't work well. For example, Reportoire comes with some handy ready-to-run class reports, however these occasionally fail without any immediately obvious reason.
- Context sensitive help is available, but there is a very small number of help topics, and these are in general not as useful as one would hope.
- Reportoire inherits some problems from VisualWorks—for example, Windows 3.x/VisualWorks users occasionally see "ghost" popup menus left behind when a parent window closes, which usually cause a walkback.
- We've no experience attempting to strip/whittle an image containing Reportoire, and we have some concerns about image size. However, all our experience has been with the full-blown Development version—adding the Embedded Execution version of Reportoire may have a much smaller impact on the image.

Although we have had some problems with Reportoire, SSI technical support (in the form of email, faxes and phone conversations with Bill Reynolds, the president of SSI and chief developer of Reportoire) has been excellent. The company has been extremely responsive, often solving our problems immediately or within just a few hours.

NOTES ON THE EVALUATION ENVIRONMENT

For the record, we've tested Reportoire only in our environment: VisualWorks (standalone and Envy) running under Windows 3.1 on 486/66 systems with 16 meg of memory—our development platforms.

SUMMARY

Reportoire 1.0 is impressive in scope, functionality and design. When developers first see the WYSIWYG report designer, and begin to understand just how much time it can save, the response is very positive. However, this is version 1.0—some bugs haven't been worked out and the product needs refinement. In their favor, SSI has been supporting Smalltalk since 1989 (with already successful products such as Smalltalk/LAN and Smalltalk/SQL) and we expect future versions will resolve these problems. In the meantime, if your application requires reporting, you should strongly consider using Reportoire. Perhaps even more exciting, and important for Smalltalk, the language that has been touted as the MIS software development tool of the 90's finally has a third party tool that speaks directly to MIS software needs in a powerful way.

For further information on Reportoire contact SSI at 908.422.0450 or via email at 70233.2017@compuserve.com. ■

Jeff Cantwell is the Vice President of Research and Development, and Douglas Camp is a Software Developer for Private Business, Inc. They can be reached at Box 1603, Brentwood, TN, 37024, (615) 790-0484. Jeff's email address is cantwell@vanderbilt.edu, Doug's is 74017.2614@compuserve.com.

continued from page 18

People doing serious development have the privilege of writing their own file dialog, which isn't that easy to do well, given the bizarre behavior of some Filename operations.

I'd like to be able to point you to some good file dialog code, but I don't know of any. The best I can do is the FileNavigator goodie, which is a start, even though it's trying to solve a different problem. FileNavigator is an improved version of the standard FileBrowser, written by Carl McConnell (mcconnel@cs.uiuc.edu). He describes it as follows:

> **FileNavigator provides the same functionality as FileBrowser, but with an easier-to-use select-and-click interface reminiscent of a Macintosh file dialog box.**

He also warns that

> **I've only tested FileNavigator on the Macintosh, so although it's supposed to be portable, minor problems may crop up on other platforms.**

> "For those who have only email access, there is a mailing list that echoes postings in comp.lang.smalltalk."

I did experience some problems when I tried this out under MS-Windows VisualWorks, but I think they wouldn't be hard to fix. I didn't think it was very reminiscent of a Mac file dialog, but it was an improvement on the FileBrowser, and could be a good source of ideas for other file-manipulation tools.

Importing Bitmapped Images

Applications often need pictures, and they often need to import those pictures from some external source, in one of many incompatible image formats. You can display it on the screen, then manually import it using ImagefromUser, but that takes a lot of time. There are a few goodies available for dealing with this problem. One is PNMImport, by Frerk Meyer (frerk@telematik.informatik.uni-karlsruhe.de). The file-in describes it as follows:

> **PNMImport adds methods for importing Portable aNyMaps (PNM) as of Jef Poskanzer's pbmplus package. With the help of his package or the xv program you can import ANY bitmap you like into your Smalltalk image.**

The pbmplus package, which is available by ftp from many sources, converts a wide variety of image formats to and from the formats that PNMImport reads. As far as I know, these tools are only available on UNIX platforms, but it's possible they've been ported or that other programs can produce these formats. ■

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

RECRUITMENT

To place an ad, contact Wendy Plumb at 212.274.0640.

Technology Consulting, Inc. is a dynamic and rapidly growing Software Development Firm with challenging assignments. We are a leader in client-server systems integration and application outsourcing. We have immediate openings to support exclusive client projects and our state-of-the-art regional development center.

CLIENT SERVER - SMALLTALK, C, C++, VISUAL BASIC/C++, ORACLE, SYBASE, POWERBUILDER

TCI offers competive salaries, attractive benefits, and relocation assistance. For consideration, send resume or call:
1800 Meidinger Tower, Louisville, KY 40202
(502) 589-3110

TCI TECHNOLOGY CONSULTING INC

Member NACCB **FAX 502-589-3107**

SMALLTALK ARCHITECTS/DEVELOPERS

Minneapolis • Atlanta • Raleigh
NY • Boulder

As a leader in the delivery of Object-Oriented System Integration Services, SHL Systemhouse invites you to explore challenging and unique opportunities within our organization. SMALLTALK opportunities exist in Minneapolis, Atlanta, Boulder, Raleigh and NY, for Technical Architects, Project Managers, Senior Software Developers and Software Engineers.

We seek client/server, object-oriented professionals with impressive industry credentials who share our worldwide commitment to excellence. These results-oriented information professionals must thrive on challenges and possess exceptional technical skills as well as business advisory experience.

For Consideration send your resume in confidence to:
Michelle Hayden Dept. SMR394
SHL Systemhouse
950 South Winter Park Drive, Suite 200
Casselberry, Florida 32707
1.800.769.8704 or Fax 407.767.5309
(Extra Fine Mode)

SHL SYSTEMHOUSE

Tech Specialists

Precision Staffing. Superior Service.

We currently have numerous opportunities requiring 1+ years experience with:

- ☐ SMALLTALK
- ☐ Object Oriented Design
- ☐ OS/2
- ☐ C++
- ☐ C

For consultants and businesses alike looking for success, we are THE company of choice. Tech Specialists is a professional services firm that provides IS, Software Development and Engineering professionals on a contract basis. For opportunities with a real future, call us to explore the possibilities.

For immediate consideration, please FAX or mail resume to:
Tech Specialists
5711 Six Forks Rd.
Raleigh, NC 27609
(919) 870-5100
(919) 870-7274 Fax

The Smalltalk Report

Provides objective & authoritative coverage on language advances, usage tips, project management advice, A&D techniques, and insightful applications.

☐ **Yes, I would like to subscribe to THE SMALLTALK REPORT.**
☐ **1 year (9 issues):**
☐ Domestic: ☐ Individual $79.00 ☐ Institutional $119.00
☐ Overseas: ☐ Individual $94.00 ☐ Institutional $134.00
☐ **2 years (18 issues):**
☐ Domestic: ☐ Individual $148.00 ☐ Institutional $228.00
☐ Overseas: ☐ Individual $178.00 ☐ Institutional $258.00

Method of Payment
☐ Check enclosed (payable to THE SMALLTALK REPORT)
☐ Bill me
☐ Charge my: ☐ Visa ☐ MasterCard ☐ AmEx

Card No. _____
Exp. Date _____
Signature _____
Name _____
Address _____
Title _____ Company _____
City _____ State _____
Country _____ Zip _____
Phone _____

To order, return this form with payment to
The Smalltalk Report, P.O. Box 2027, Langhorne, PA 19047
Fax: 215.788.6073 Phone: 215.785.5996

SMALLTALK DESIGNERS AND DEVELOPERS

We Currently Have Numerous Contract and Permanent Opportunities Available for Smalltalk Professionals in Various Regions of the Country.

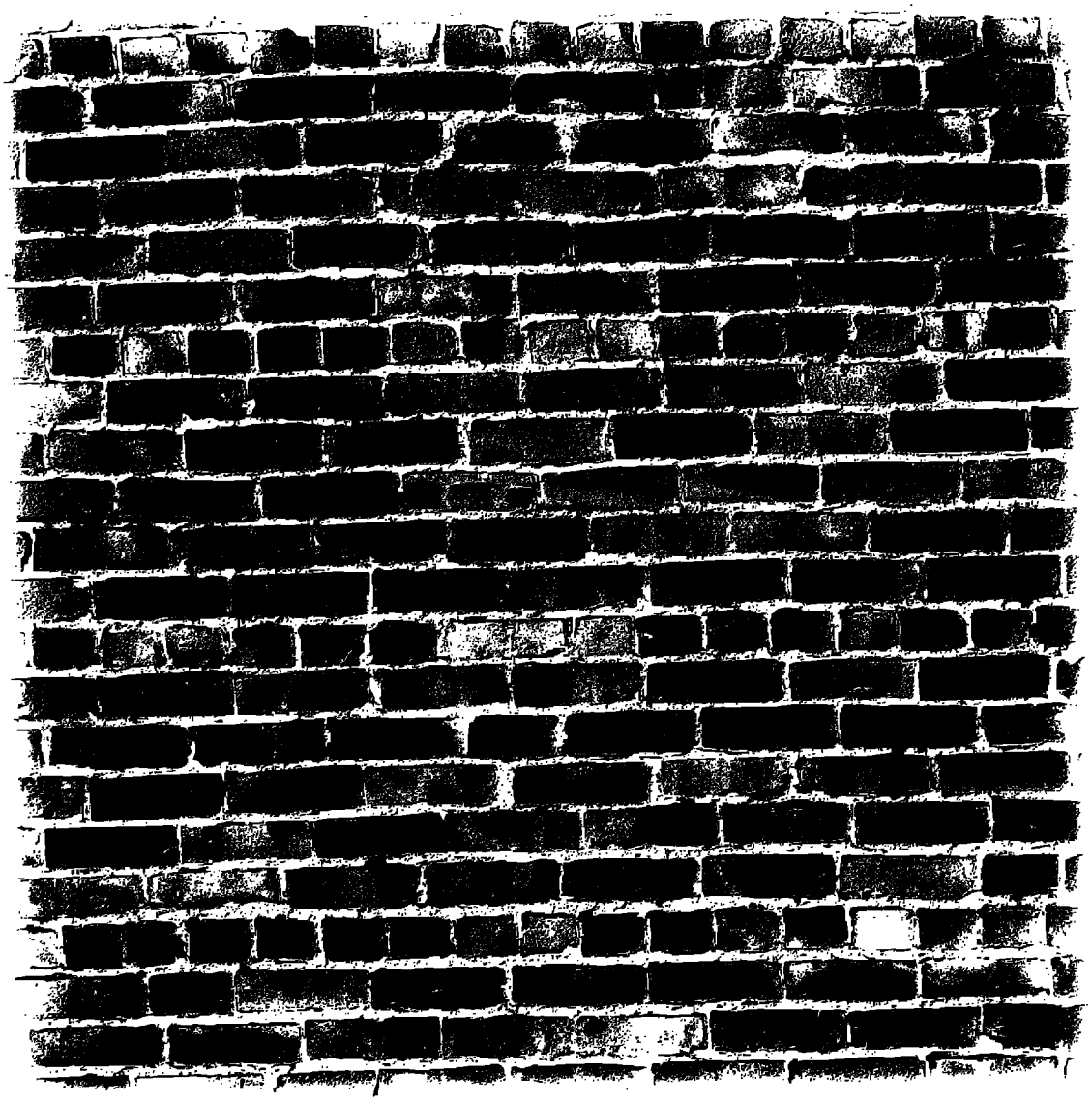
SALIENT CORPORATION

Salient Corporation...
Smalltalk Professionals Specializing in the Placement of Smalltalk Professionals

For more information, please send or FAX your resumes to:

Salient Corporation
316 S. Omar Ave., Suite B.
Los Angeles, California 90013.

Voice: (213) 680-4001 FAX: (213) 680-4030



FORCE-FIT RELATIONAL TECHNOLOGY AND YOU COULD REALLY HIT IT BIG.

Maybe you're beating your head against the relational database wall – trying to integrate your Smalltalk applications with an RDBMS. Maybe you're spending all your time debugging SQL calls instead of building great applications. Or maybe you've hit the relational performance wall because you're wasting too much processing time on object decomposition and recomposition.

Servio™ has a better way. With our high-performance GemStone® object database management system, you can store Smalltalk objects directly in the database. We make your development time more productive and your object applications more efficient.

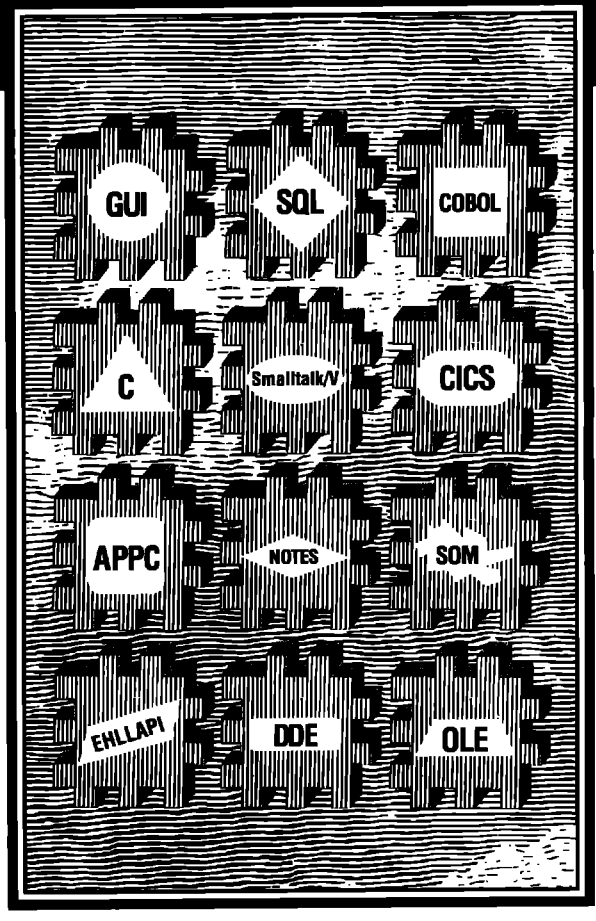
Learn for yourself by calling us today for a copy of "Object or Relational? A Guide for Selecting Database Technology." After all, the best way to deal with an obstacle is to avoid it in the first place.

SERVIO

OBJECT TECHNOLOGY FOR THE REAL WORLD

Call 1 800-243-9369 for a free copy of "Object or Relational? A Guide for Selecting Database Technology."

FINALLY, CLIENT/SERVER INTEGRATION.



Not long ago, client/server development required massive amounts of time, money and expertise to combine different and complex technologies.

Now Digitalk PARTS,® a rapid application development tool set, lets you easily integrate your software assets into client/server applications.

PARTS is the only object-oriented technology that lets you leverage your legacy code and the knowledge of your current staff.

Only PARTS products let you take existing code – written in Smalltalk/V, COBOL, C, SQL and other languages – and wrap it into components or "parts". Which can then be virtually snapped together visually. The result is smooth-running client/server applications in a fraction of the usual time. For a fraction of the usual cost.

PARTS supports all popular SQL databases like Sybase, Oracle and DB2. Plus legacy or late model systems like CICS, COBOL, APPC and SOM. And PARTS lets you develop on both OS/2 and Windows.

RATED #1 – TWICE.

Only months ago, PC WEEK awarded PARTS Workbench the highest rating ever in the OS/2 category, calling it "the definitive visual development tool".

And InfoWorld ranked PARTS the #1 component-based tool for visual development. InfoWorld's Stewart Alsop adds: "There's nothing like it on the PC."

To make large teams productive, PARTS also supports group development and version control. Plus PARTS has a host of graphical power tools to give you all the power of objects – without the learning curve.

10 YEARS EXPERIENCE.

And PARTS is from Digitalk. The company that's been providing object-oriented tools to the Fortune 500 longer than anyone else in the world – with over 125,000 users.

Call 800-531-2344 X 610 and ask about our PARTS Workbench Evaluation Kit.

With minimum effort, you'll learn why PARTS is the maximum solution for client/server integration.

PARTS. THE CLIENT/SERVER INTEGRATION TOOL. DIGITALK