The Smalltalk Report

The International Newsletter for Smalltalk Programmers

May 1994 **Volume 3 Number 7**

EXTENDING THE APPLICATION MODEL

by Tim Howard & Bill Kohl

The enhancements to Smalltalk application development provided by VisualWorks extend well beyond the advantages of GUI window painting. Fundamental among these enhancements is a new application architecture that includes a type of model dedicated exclusively to managing an entire application—or at least an entire window. This type of model is implemented by the class ApplicationModel. Although ApplicationModel offers a rich set of features for application management, we have found additional abstract subclasses of ApplicationModel, which add still more features, to be of tremendous benefit in our VisualWorks development. Currently, we have two very mature abstract subclasses of ApplicationModel. When starting a new project for a client, we typically create yet another abstract subclass, specific to that client's needs. These additional abstract subclasses of ApplicationModel provide several benefits:

- easier control of the interface
- additional functional features
- lean implementations of concrete application model classes
- elegant and readable source code for concrete application model classes
- consistency of features and behavior over all concrete application model classes which comprise an entire application
- improved dialog development

In this article, we will offer some of the more useful enhancements that have been developed so far: component services, aspect services, and containing model reference. These features are provided in a subclass of ApplicationModel we call ExtendedApplicationModel. Although most of these ideas are quite simple, they greatly facilitate application model development. Also, it is important that none of this additional functionality interfere with, or override, what is already provided in ApplicationModel. That is, any existing subclass of ApplicationModel should also be able to run as a subclass of ExtendedApplicationModel without any adjustments in its implementation. Finally, we will show how to add ExtendedApplicationModel to your VisualWorks class creation dialog so that it is available when installing a canvas.

COMPONENT SERVICES

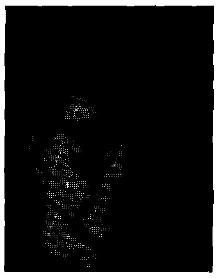
Component services are used to control the components—and related interface objects such as windows and keyboard hooks—during runtime. What they offer the developer is

- elegance of code
- readability of code

continued on page 4...

Contents:

John Pugh


Paul White

EDITORS' CORNER

We're very excited to have the opportunity to be the first to tell you about the upcoming changes planned for The Smalltalk Report as we start our fourth year of publication in September. With Smalltalk's move into the mainstream as an application development tool and as the underlying scripting language for visual programming environments such as IBM's VisualAge and Digitalk's PARTS, the interest and activity within the Smalltalk arena is growing exponentially. Reflecting this and our rapidly growing readership, The Smalltalk Report will be expanded from its current 24 pages to 32 pages and will take on a new look with enhanced use of color throughout the publication. We will be able to expand the editorial content of The Smalltalk Report, which, for our readers, translates into more features, articles, columns, and reviews. Over the next few months we will be introducing you to new columnists and new departments.

A new feature we would like to add to The Smalltalk Report starting in September is something that, for the want of a better name we are calling the "Wow, that's neat..." column. The idea is to have our readers contribute some small snippets of code that provide some interesting or useful functionality. We are hoping that the description and code together will fit into roughly one printed page. If you have some "neat things" that you have built in the past that you'd be willing to share with our readership, why not pass them on?

We are delighted to introduce Tim Howard and Bill Kohl as new columnists this month. They will be writing columns that will be dealing with issues of software construction using Smalltalk. Their first contribution is featured this month and deals with making extensions to VisualWorks' Application Model by introducing useful abstract classes to ApplicationModel. We're sure you'll agree that their contribution will meet the same high standards set by all our columnists.

Also in this issue, two of our regular columnists return. Kent Beck is continuing with his "Where do objects come from?" series, which has been dealing with issues of when and why objects should be introduced during development. Alan Knight presents us with a potpourri of issues that have been raised on Internet's comp.lang.smalltalk, which is a constantly growing forum in which Smalltalkers can participate. If you haven't checked out comp.lang.smalltalk, you may wish to do so. Although not all of what is said there will be of interest to you, you will undoubtedly find gems of information to help in your work.

Wayne Beaton returns this month with a description of extensions that can be made to Smalltalk/V's menu facilities giving them a more object-oriented flavor. Finally, Scot Campbell provides a detailed inside look at Team/V, Digitalk's facility for team development.

John Pugh [illegible]

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. © Copyright 1994 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publisher.

Mailed First Class. Canada Post International Publications Mail Product Sales Agreement No. 290386. Subscription rates 1 year (9 issues): domestic, $79; Foreign and Canada, $114; Single copy price, $8.

POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, P.O. Box 2027, Langhorne, PA 19047. For service on current subscriptions call 215.785.5996.

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.

The Smalltalk Report

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

SIGS Publications
Advisory Board
Tom Atwood, Object Design
François Bancilhon, O$_2$ Technologies
Grady Booch, Rational
George Bosworth, Digitalk
Brad Cox, Information Age Consulting
Adele Goldberg, ParcPlace Systems
Tom Love, IBM
Bertrand Meyer, ISE
Meilir Page-Jones, Wayland Systems
Sesha Pratap, CenterLine Software
Cliff Reeves, IBM
Bjarne Stroustrup, AT&T Bell Labs
Dave Thomas, Object Technology International

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

Columnists
Kent Beck, First Class Software
Juanita Ewing, Digitalk
Greg Hendley, Knowledge Systems Corp.
Ed Klimas, Linea Engineering Inc.
Alan Knight, The Object People
Eric Smith, Knowledge Systems Corp.
Rebecca Wirfs-Brock, Digitalk

SIGS Publications Group, Inc.
Richard P. Friedman
Founder & Group Publisher

Art/Production
Kristina Joukhadar, Managing Editor
Susan Culligan, Pilgrim Road, Ltd., Creative Direction
Seth J. Bookey, Production Editor
Andrea Cammarata, Electronic Publishing Coordinator
Margaret Conti, Production Assistant

Circulation
Bruce Shriver, Circulation Director
K.S. Hawkins, Fulfillment Manager

Marketing/Advertising
Shirley Sax, Director of Sales
Gary Portie, Advertising Mgr—East Coast/Canada/Europe
Helen Newling, Advertising and Exhibit Sales
Sarah Hamilton, Manager of Promotions and Research
Caren Polner, Promotions Graphic Artist

Administration
David Chatterpaul, Accounting Manager
James Amenuvor, Bookkeeper
Amy Melsten, Assistant to the Publisher
Joanna Lowenstein, Administrative Assistant

Margherita R. Monck, General Manager

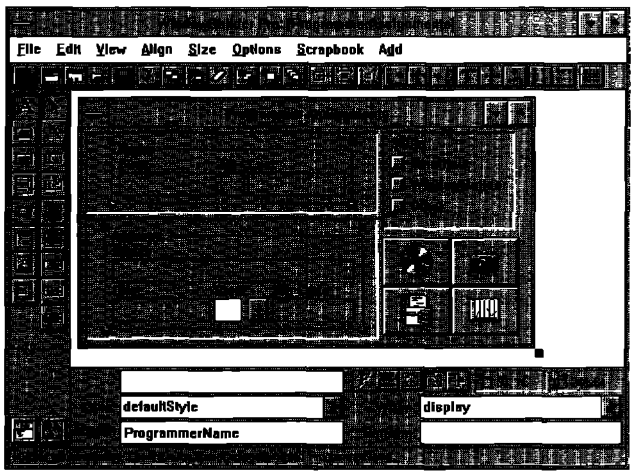
SIGS PUBLICATIONS

OBJECT SHARE SYSTEMS INC.

WINDOWBUILDER PRO!

The New Power in Smalltalk/V Interface Development

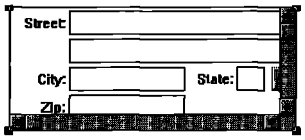
Smalltalk/V developers have come to rely on WindowBuilder as an essential tool for developing sophisticated user interfaces. Tedious hand coding of interfaces is replaced by interactive visual composition. Since its initial release, WindowBuilder has become the industry standard GUI development tool 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.



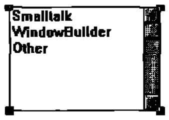
WindowBuilder Pro/V is available on Windows for $295 and OS/2 for $495. Our standard WindowBuilder/V is still available on Windows for $149.95 and OS/2 for $295. We offer full value trade-in for our WindowBuilder customers wanting to move up to Pro. These products are also available in ENVY®/*Developer* 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.

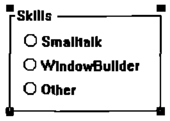
Some of the exciting new features...

• 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.

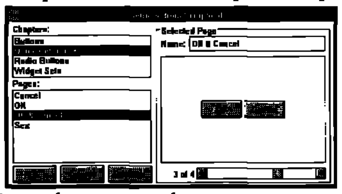


• Morphing: Allows the developer to quickly change from one type of control to another, allowing for powerful "what-if" style visual development. The flexibility allowed by morphing will greatly enhance productivity.

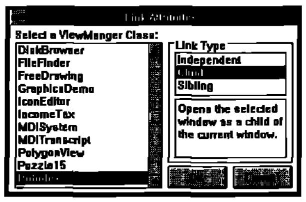


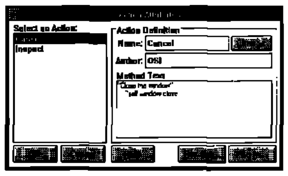


• ScrapBook: Another new feature to leverage visual component reuse, ScrapBooks provide a mechanism for developers to quickly store and retrieve predefined sets of components. The ScrapBook is a catalog of one's favorite interface components, organized into chapters and pages.



• 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

WindowBuilder and WindowBuilder Pro are trademarks of Objectshare Systems, Inc. All other brand and product names are registered trademarks of their respective companies

continued from page 1

- brevity of code
- development efficiency
- flexibility
- safe component access
- additional functionality

After you have written a few meaningful applications in VisualWorks, you may begin to notice a certain repetition of code where component control is concerned. For example, suppose we want a method that is responsible for disabling three components—#name, #address, and #phone. The implementation for such a method might look like this:

```
(self builder componentAt: #name) disable.
(self builder componentAt: #address) disable.
(self builder componentAt: #phone) disable.
```

At first glance, one notices that this code is not very readable. A statement such as self builder componentAt ...disable does not fit very well with our common vernacular. To a second party reviewing this code, it is not readily apparent what is going on. Smalltalk code should be short, elegant, and readable. Also, notice how much redundancy is involved. There is no excuse for this in Smalltalk, which leads us to the next implementation. Most veteran Smalltalkers would implement our example method as follows.

```
#(name address phone) do: [:each |
    (self builder componentAt: each) disable]
```

An improvement, but we are still not quite there. This second implementation is more elegant in that it removes the redundancy, but it is even less readable than the first. In English, what we are trying to do is *disable #name, #address, and #phone.* Is there any reason why we cannot write the method just this way? Certainly not! Application models that are subclasses of ExtendedApplicationModel implement our example method as follows.

```
self disable: #(name address phone)
```

This third implementation is short, concise, and readable. In the time it takes to read this very short line, we know exactly what is taking place. Application models that are subclasses of ExtendedApplicationModel are able to control their components in just such a fashion.

As another illustration of the utility of component services, consider changing one of the colors of a component. This is not at all a straightforward task. There are some subtleties involved with this process and several lines of code are required to do the job. For example, if we wanted to change the foreground color of a component whose ID is #notes, then we would write something like the following.

```
| lp comp |
comp := self builder componentAt: #notes.
lp := comp lookPreferences.
lp := lp foregroundColor: ColorValue red.
comp lookPreferences: lp
```

This is quite a bit of code and not at all readable. Of course, the acid test of readability is to state in English exactly what we are trying to accomplish. In this case, the English version reads *change the foreground color of #notes to the color red.* The ExtendedApplicationModel offers a component service that allows us to write the code in just such a way.

```
self change: #foregroundColor
  of: #notes to: ColorValue red
```

Notice that we also gain in brevity and elegance. For someone reviewing the code, it is immediately obvious what this statement does. To provide some flexibility, the color argument can be either

Table 1. Component Services.

Message	**Behavior**
abortFocusShift	Prevents a shift of focus from taking place.
change: aSymbol of: aSymbolOrArray to: aSymbolOrColor	Change the color role aSymbol of the component(s) identified by aSymbolOrArray to the color indicated by aSymbolOrColor
component: aSymbol	Return the component (SpecWrapper) whose id is aSymbol
controllerFor: aSymbol	Return the controller for the widget identified by aSymbol
disable: aSymbolOrArray	Disable the component(s) identified by aSymbolOrArray
enable: aSymbolOrArray	Enable the component(s) identified by aSymbolOrArray
invalidate: aSymbolOrArray	Redraw the component(s) identified by aSymbolOrArray
keyboardHook	Return the keyboard hook.
keyboardHook: aSymbol	Retrun the keyboard hook for the widget of the component whose ID is aSymbol.
keyboardProcessor	Return the keyboard processor.
makeInvisible: aSymbolOrArray	Make invisible the component(s) identified by aSymbolOrArray
makeVisible: aSymbolOrArray	Make visible again the component(s) identified by aSymbolOrArray
replaceWidgetControllerIn: aSymbol with: a Controller	Replace the controller in the widget identified by aSymbol with aController
takeFocus: aSymbol	Give focus to the component identified by aSymbol
turnOff: aSymbolOrArray	Turn off the component(s) identified by aSymbolOrArray
turnOn: aSymbolOrArray	Turn on the component(s) identified by aSymbolOrArray
widget: aSymbol	Return the widget identified by aSymbol
window	Return the builder's window.

Now! Automatic Documentation

For Smalltalk/V Development Teams — With ***Synopsis***

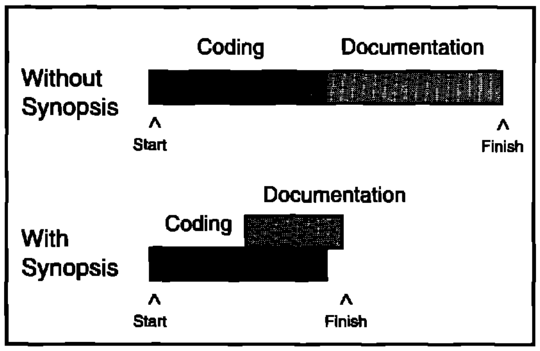
Synopsis produces high quality class documentation automatically. With the combination of Synopsis and Smalltalk/V, you can *eliminate the lag between the production of code and the availability of documentation.*

Synopsis for Smalltalk/V

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

Dan Shafer, Graphic User Interfaces, Inc.:
"Every serious Smalltalk developer should take a close look at using Synopsis to make documentation more accessible and usable."

Development Time Savings



Products Supported:

Digitalk Smalltalk/V Windows	$295
Digitalk Smalltalk/V OS2	$395

(OS/2 version works with Team/V and Parts)

Synopsis Software
8609 Wellsley Way, Raleigh NC 27613
Phone 919-847-2221 Fax 919-847-0650

some kind of Paint object or a Symbol identifying one of the named ColorValues such as #navy or #lightGray.

As another form of flexibility, many component services that take a component ID as an argument can also take an Array of component IDs. For instance, we can make a single component invisible with

```
self makeInvisible: #name
```

Or with the same message, we can make an arbitrary number of components invisible, such as

```
self makeInvisible: #(name rank serialNumber)
```

The component services are also robust enough to ignore any errant component IDs—those that do not identify a component. The component lookup is conducted such that component IDs not found in the builder's named components collection are ignored.

There are two component services that add some additional functionality not currently available in VisualWorks. These component services turn components off and on. Turning off a component is similar to disabling it in that the turned off component will not respond to user input. Unlike disabling, however, the component is not redrawn in a gray hue but maintains its original color. The implementation is quite a bit different as well. A component is turned off by giving its widget a NoController. The purpose of turning off a component is to provide a read only effect. Turning on a component merely reinstates the widget's default type of controller so that it can accept user input again.

There are 18 component services; these are listed in Table 1.

ASPECT SERVICES

When designing an application model that manages several components, the instance variable list tends to become somewhat overloaded. Traditionally, good Smalltalk style frowns on class definitions with excessive instance variables. Such a symptom can be indicative of

- a lack of factoring in the hierarchy
- a lack of support object types and collaborator object types
- unnecessary and unused instance variables

Table 2. Aspect service argument for ValueHolder.

aSelector	A Symbol which is the name of the method using the service.
anObject	The initial value of the ValueHolder.
aSelectorOrArray	A Symbol or an Array. If it is a Symbol, it is the name of a unary message to be sent to the application model on a change of value of the ValueHolder. If it is an Array, the first element is the change message, and the second element is the receiver of the message.

In general, an excess of instance variables usually indicates that the class is assuming too much responsibility and that some of this responsibility should be defined in one or more super classes or delegated to collaborator and support objects. In the case of concrete application model development, however, these two options usually do not apply. This leaves only one avenue for reducing our number of instance variables—removing those that are unnecessary. Well, the aspect instance variables are unnecessary! The reason for this is that a UIBuilder caches these objects in its bindings variable. This means the application model maintains redundant references to its aspect models. The application model can always reference its aspect models via its builder, so why keep them as instance variables? ExtendedApplicationModel uses this information to provide aspects for the components without having to load up on aspect instance variables.

Typically, an aspect method returns some type of value model, or some other type of aspect object such as a SelectionInList. In an extended application model, aspect methods can be written in one of two ways. For example, a method for the #documentName aspect might look like the following:

```
documentName
    ^documentName isNil
        ifTrue: [documentName := String new asValue]
        ifFalse: [document]
```

This first implementation, which is the traditional approach, requires the allocation of an instance variable, documentName. In an extended application model, the method can also be written as

Table 3. Aspects service arguments for SelectionInList.

aSelector	A Symbol that is the name of the aspect and of the method.
anObject	This can be nil, a sequenceable collection, or a Symbol. If it is nil, then an empty SelectionInList is created. If it is a collection, the SelectionInList is initialized with the collection. If it is a Symbol, then it is interpreted as a message sent to the application model to retrieve the list which is then used to initialize the SelectionInList.
aSelectorOrArray1	A Symbol or Array. If a Symbol, then it is the change message sent to the application model on a change of the list. If an Array, then the first element is the change message and the second element is the receiver of the message.
aSelectorOrArray2	A Symbol or Array. If a Symbol, then it is the message sent to the application model on a change of the selection index. If an Array, then the first element is the change message and the second element is the receiver of the message.

Table 4. Aspect service arguments for SubCanvas.

aSelector	A Symbol which is the name of the aspect and of the method.
aModel	A subclass of ApplicationModel or an instance of such a class.

```
documentName
    ^self valueHolderFor: #documentName initialValue: String new
```

This second implementation does not require an instance variable! As long as the aspect is accessed using the accessing message (a practice we strongly encourage), the correct value will be returned and the application model will behave just as if the implementation was that of the first type. Also notice how much more readable it is than the traditional implementation.

There are two aspect services for ValueHolders:

```
valueHolderFor: aSelector initialValue: anObject
```

and

```
valueHolderFor: aSelector initialValue: anObject
changeMessage: aSelectorOrArray
```

The arguments are described in Table 2.

There are two forms of the aspect service for SelectionInList and MultiSelectionInList

```
    selectionInListFor: aSelector list: anObject
```

and

```
selectionInListFor: aSelector list: anObject listChange:
    aSelectorOrArray1 selectionChange: aSelectorOrArray2
```

The arguments are defined in Table 3.

When the component is a subcanvas, then the aspect is another application model. The aspect service in this case is

```
    applicationFor: aSelector model: aModel
```

The arguments are presented in Table 4.

Also, if the sub environment is itself an extended application model, then it will automatically receive a reference to its containing model. The containing model properties of ExtendedApplicationModel are covered shortly.

All the aspect services use the same approach. For illustration, we will use the valueHolderFor:initialValue:changeMessage: aspect service. Whenever someone sends a message to access an aspect, that method's implementation is an aspect service message. For instance, an application model with a #productID aspect might have a method which looks like the following.

```
productID
    ^self
        valueHolderFor: #productID
        initialValue: String new
        changeMessage: #changedProductID
```

The implementation of the valueHolderFor:initialValue:changeMessage: aspect service method defined in ExtendedApplicationModel is shown below.

```
valueHolderFor: aSelector initialValue: anObject changeMessage:
    aSelectorOrArray
    ^(self builder bindings includesKey: aSelector)
        ifFalse:    [self
                    registerInterestIn: (ValueHolder with: anObject)
            using: aSelectorOrArray]
        ifTrue: [self builder aspectAt: aSelector]
```

This method checks first to see if the builder already has the aspect model in its bindings. If so, then access it from the builder and return it (the ifTrue: clause). If not, then create the ValueHolder with the initial value of anObject and use the information in aSelectorOrArray to register interest in the ValueHolder. Then return this new ValueHolder (the ifFalse: clause). The interest in the ValueHolder is registered by sending the message registerInterestIn: aValueHolder using: aSelectorOrArray. The implementation for this method is shown below.

```
registerInterestIn: aValueModel using: aSelectorOrArray
    aSelectorOrArray isNil ifTrue: [^aValueModel].
    (aSelectorOrArray isKindOf: Array)
        ifTrue: [aValueModel
                    onChangeSend: (aSelectorOrArray at: 1)
                    to: (aSelectorOrArray at: 2)]
        ifFalse: [aValueModel
            onChangeSend:
            aSelectorOrArray to: self].
    ^aValueModel
```

If aSelectorOrArray is a Symbol, then it is understood that the interested object is the application model itself. If aSelectorOrArray is an Array, then its first element is expected to be a Symbol naming the change message and its second element is expected to be the interested object, that is, the receiver of the change message.

One caveat to using the aspect services is that the aspects are necessarily public. If a completely private aspect is desired, you must abandon the aspect services and declare an instance variable that does not have an accessing method.

Although there is no functional benefit in using aspect services, we do get a threefold increase in elegance, namely,

- a reduction in the amount of instance variables in the class definition
- much more readable and descriptive aspect methods
- the discipline of referencing aspects by their accessing messages

CONTAINING MODEL

Quite often, when an application model is launched from another application model, the new application model will want to reference its parent. Also, it is quite convenient for an application model running a subcanvas to reference the containing application model. For these reasons, ExtendedApplicationModel defines an instance variable, containingModel, which allows an application model to reference the containing application model which launching it or contains it as a sub environment.

ADDING TO VISUALWORKS

The complete source code for ExtendedApplicationModel, along with an example application, can be acquired from the University of Illinois Smalltalk Archives in ST_80VW directory as "extendedApplicationModel.st." To make ExtendedApplicationModel available from the class creation dialog, open the interface in UIFinder class>>classCreationDialog for editing. Now add a radio button below the ApplicationModel radio button. Give this new radio button the following properties and install the canvas.

For example,

```
Label   ExtendedApplicationModel
Aspect  #superPick
Select  #ExtendedApplicationModel
```

To make ExtendedApplicationModel the default selection for this dialog, you must edit the UIFinder class method

```
openNewClassDialogForName: aClassName subClassing: aSuperName
    inCategory: aCategory
```

This is a very long method. Find the part that reads

```
builder
    aspectAt: #superPick
    put:
        (superPick :=
            (superName value isEmpty
                ifTrue: [#ApplicationModel]
                ifFalse: [#Other]) asValue).
```

and change the ifTrue: value form #ApplicationModel to #ExtendedApplicationModel. This will make ExtendedApplicationModel the default superclass of any new classes created as a result of installing a canvas.

CONCLUSION

VisualWorks projects of any merit should include one or more abstract subclasses of ApplicationModel to facilitate application development. In this article, we developed such a class, ExtendedApplicationModel, and populated it with some very useful features: component services, aspect services, and a containing model reference. Component services facilitate the control of the interface objects during runtime and provide more readable and elegant implementations. The aspect services eliminate the need to load up on instance variables when defining an application model class and also provide brief, readable aspect method implementations. The containing model reference is an instance variable that allows an application model to reference its parent or containing model for which it serves as a subenvironment. These three enhancements are good examples of why it is advantageous to create abstract subclasses of ApplicationModel. ■

Tim Howard holds an MBA and a MS in Industrial Engineering and has been developing application software for eight years. Presently he is working on a VisualWorks book for SIGS Publications and consults for RothWell International. He can be reached at the RothWell offices at 800.256.0541, at home at 713.784.9730, or via email at 74213.1517@compuserve.com.

Bill Kohl is a Training Administrator at RothWell International and can be reached at the RothWell International offices at 800.256.0541.

FREEDOM

THROUGH

ADVANCED

TECHNOLOGY

Smalltalk X offers

- system independence: Each application within the runtime systems ported is independent of the underlying system and - as a rule - can be ported and run without any changes or modifications

- Inline C code means it is easy to link existing applications and libraries. Additional, systemapplications and classes can be supplied in machine code (protects your technology)

- compact, standalone applications are possible thanks to static or dynamic linking features

- With the integrated development environment you can develop software and see the results immediately in an "organic" system

- Extensive class libraries both simplify and speed up software development and improve quality and serviceability

Get more information from:

Tomcat computer GmbH

Fruehlingstr. 4

D - 82110 Germering (Germany)

Tel.: (++49) 89 - 84 99 92

Fax: (++49) 89 - 84 95 44

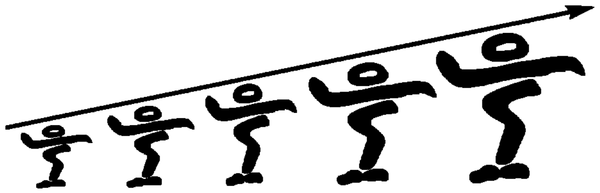
E-Mail: office@tomcat.de

NEW

SMALLTALK X

Your way to

High - Productivity



● developed by Claus Gittinger ●

The object-oriented development tool for high-portable applications

Beta CD-ROM only US$ 140,-

ALL AVAILABLE PLATTFORMS ON OUR Beta-CD-ROM 1.0:

R 3000/4000: Silicon Graphics/Irix 4.0.x, ; *NeXT*: Mach with X11; *SPARC*: SunOs 4.1.x; *80386/486*: LINUX V.0.99 pl 10/11

Yet also running:PARisc: HP-UX 9.0; *SPARC*: Solaris 2.x; *R 3000/4000*: PCS/MUNIX; *80386/486*: Sys 5.41/5.42, SINIX 5.4x; UnixWare; Solaris x86

All trademarks are the property of the respectiv holders

"Smart Menus" in Smalltalk/V for Win32

Wayne Beaton

Menus play an important role in any Windows-based application. Typically, an application will "gray-out", or disable menu entries that don't make sense in the current context. Menu items may or may not have a check mark to their left, indicating that an option is active or inactive. Managing these menu items can be cumbersome, if not downright difficult using the mechanisms built into Smalltalk/V (for large applications, it certainly takes far too much effort to get menus to operate correctly).

The current mechanism feels a lot like functional programming: if, for example, an application has a menu titled 'File' and a menu titled 'Edit', each containing a number of entries, a ViewManager subclass might have code to update these menus which looks something like:

```
self isDocumentDirty
  ifTrue: [
    (self menuTitled: 'File')
        enableItem: #fileSave]
  ifFalse: [
    (self menuTitled: 'File')
      disableItem: #fileSave].
self clipboardContainsObject
  ifTrue: [
    (self menuTitled: 'Edit')
      enableItem: #editPaste;
      enableItem: #editPasteSpecial]
  ifFalse: [
    (self menuTitled: 'Edit')
      disableItem: #editPaste;
      disableItem: #editPasteSpecial]
 (...etc...)
```

For a small number of entries, this technique may be easy to understand and use. As the size of the application and the number of menus increase, this type of code can grow exceedingly complex. To start, the label of the menu is required to access it. If the application is required to function in multiple languages, this issue becomes even more complex (the typical solution might be to have a specific method that will determine each menu title). If the logic that determines the enabled or disabled state gets any more complicated, the method becomes so nested that even expert Smalltalkers choke, gag and eventually collapse and die.

AN OBJECT-ORIENTED APPROACH?

As strange as it may seem, Smalltalk is object-oriented, which means (in part) that objects know things about themselves. Why not have menus know how to update themselves?

When a menu item is created, a label, selector and accelerator is specified for it. The label is the string that will be displayed for the user when the menu is selected and pulled down. The selector is a unary message that will be sent to the menu's owner when the item is selected by the user and the accelerator is a description of the keyboard equivalent of selecting the item.

Smart menus will consider additional information. When an item is added to a smart menu, the programmer can also specify a block of code as the "enabled condition" for the item. The block must result in a Boolean when evaluated—if the result is true, the item will be enabled, if false the item will be disabled. Similarly, the programmer can specify a "checked condition" which is also a block resulting in a boolean which determines if the item is checked or not. If no conditions are specified, then by default an item will be enabled and unchecked.

A subcass of Menu has been introduced named SmartMenu (see listings 1 and 1a) that handles menu updating. A single method has been added to MenuWindow (see listing 2) to simplify the interaction between an application and its menus.

A menu is created using the existing methods in class menu. Enabling and checking behavior can be added for an item after that item has been added. For example, to create a "File" menu, a method using the following code might be employed:

```
buildFileMenu
  "Build and answer the menu titled 'File'."
  ^SmartMenu new
    owner: self;
    title: 'File';
    appendItem: 'Open...' selector: #fileOpen;
    appendItem: 'Save' selector: #fileSave;
    enableItem: #fileSave
      when: [self isDocumentDirty];
    yourself
```

This method builds a menu titled "File" with two entries. The entry labeled "Open..." is always enabled. The entry labeled "Save" is only enabled when the document is dirty.

There are two ways that the menu updating process can be initiated. Before a menu is pulled down, the menu window is triggered with the event aboutToDisplayMenu; the handler for this event can send the message updateMenus to the menuWindow. Alternately, the menuWindow can be asked at any time to update its menus (again using the message updateMenus).

The class ClassList has been created to demonstrate the use of SmartMenus (see listing 3). This class builds a window containing a list box populated with the classes known to the sys-

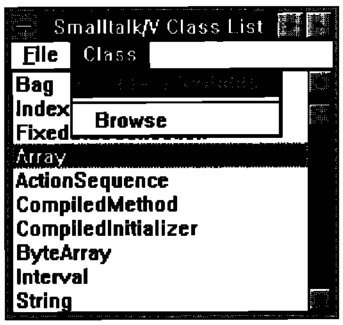


Figure 1. An instance of ClassList.

tem (see Fig. 1). A single menu, titled "Class" is added containing two entries. The first entry, labeled "Class Is Variable", is always disabled and checked only if the selected class is a variable byte class. The second entry, labeled "Browse", is enabled only if a class is selected and is never checked.

The menu is created in response to the event needsMenu. When this event is encountered, the method updateClassListBoxMenu: assigns the menu created by the method buildClassMenu to the list box using the SubPane method setMenu:. The method buildClassMenu generates a new instance of SmartMenu by adding each item, and then indicating under what conditions each item is enabled or checked.

When a class in the list is selected, the event clicked: is triggered, sending the method clickedClassListBox: to the ClassList. This method remembers the selected class and updates the menus by sending the message updateMenus to the MenuWindow.

SOME CONCLUSIONS

Smalltalk/V provides a rich set of user interface tools, but it seems they have not yet evolved aspects of it into a more usable form. As is often the case, the environment can be manipulated into a more usable form with minimal impact.

The code presented here is only a first step. With some imagination, there are other facilities that can be integrated easily. Enabling and disabling items is something that most, if not all, applications do. Some applications dynamically change the contents of some menus, change the text of some entries, or even include a graphic that may change. Such less-generic behaviour can be easily added. ■

Wayne Beaton is a senior member of the Development Team at the Object People. His interests include User Interfaces, Neural Networks, and snickering of people who wear socks with sandals. He can be reached at The Object People in Ottawa at 613.225.8812 (Wayne@ObjectPeople.on.ca).

Listing 1. The class SmartMenu.

```
Menu subclass: #SmartMenu
 instanceVariableNames:
  ' itemEnableConditions itemCheckConditions ' classVariableNames: ''
 poolDictionaries: '' !

!SmartMenu methods !

checkItem: aSymbol when: block
  "Set a check mark beside the item only when block evaluates true."
  self itemCheckConditions
    at: aSymbol put: block.
  self updateItemWithSelector: aSymbol!

enableItem: aSymbol when: block
  "Enable the item named aSymbol only when block evaluates true."
  self itemEnableConditions
    at: aSymbol put: block.
  self updateItemWithSelector: aSymbol!

initialize
  "Private - Initialize myself." super initialize.
  self
    initializeItemEnableConditions;
    initializeItemCheckConditions!

initializeItemCheckConditions
  "Private - Initialize my collection of check conditions. "
  self itemCheckConditions: Dictionary new!

initializeItemEnableConditions
  "Private - Initialize my collection of enable conditions. "
  self itemEnableConditions: Dictionary new!

isSmartMenu
  "Answer whether I am an instance of SmartMenu."
  ^true!

itemCheckConditions
  "Private - Answer my collection of check conditions."
  ^itemCheckConditions!

itemCheckConditions: aDictionary
  "Private - Set my collection of check conditions."
  itemCheckConditions := aDictionary!

itemEnableConditions
  "Private - Answer my collection of enable conditions."
  ^itemEnableConditions!

itemEnableConditions: aDictionary
  "Private - Set my collection of enable conditions."
  itemEnableConditions := aDictionary!

itemSelectorsDo: block
  "Private - Evaluate block with the selector for each of my selectors
    as parameter."
  items do: [:item |
    block value: item selector]!

shouldItemBeChecked: aSymbol
  "Private - Answer whether the item named aSymbol should be
    checked or not."
  ^(self itemCheckConditions
    at: aSymbol ifAbsent: [^false]) value!

shouldItemBeEnabled: aSymbol
  "Private - Answer whether the item named aSymbol should be
    enabled or not."
  ^(self itemEnableConditions
    at: aSymbol ifAbsent: [^true]) value!

update
  "Update all of my items."
  self itemSelectorsDo: [:each |
    self updateItemWithSelector: each]!
```

```
updateItemWithSelector: aSymbol
   "Private - Update the item with selector aSymbol.  First enable or
      disable the item based on the value of the appropriate enable
      block. Second, check or uncheck the item based on the value of
      the appropriate check block."
   (self shouldItemBeEnabled: aSymbol)
      ifTrue: [self enableItem: aSymbol]
      ifFalse: [self disableItem: aSymbol].

   (self shouldItemBeChecked: aSymbol)
      ifTrue: [self checkItem: aSymbol]
      ifFalse: [self uncheckItem: aSymbol]! !
```

Listing 1a. Extenstion to class Object.

```
!Object methods !

isSmartMenu
   "Answer whether I am an instance of
   SmartMenu."
   ^false! !
```

Listing 2. Extension to class MenuWindow.

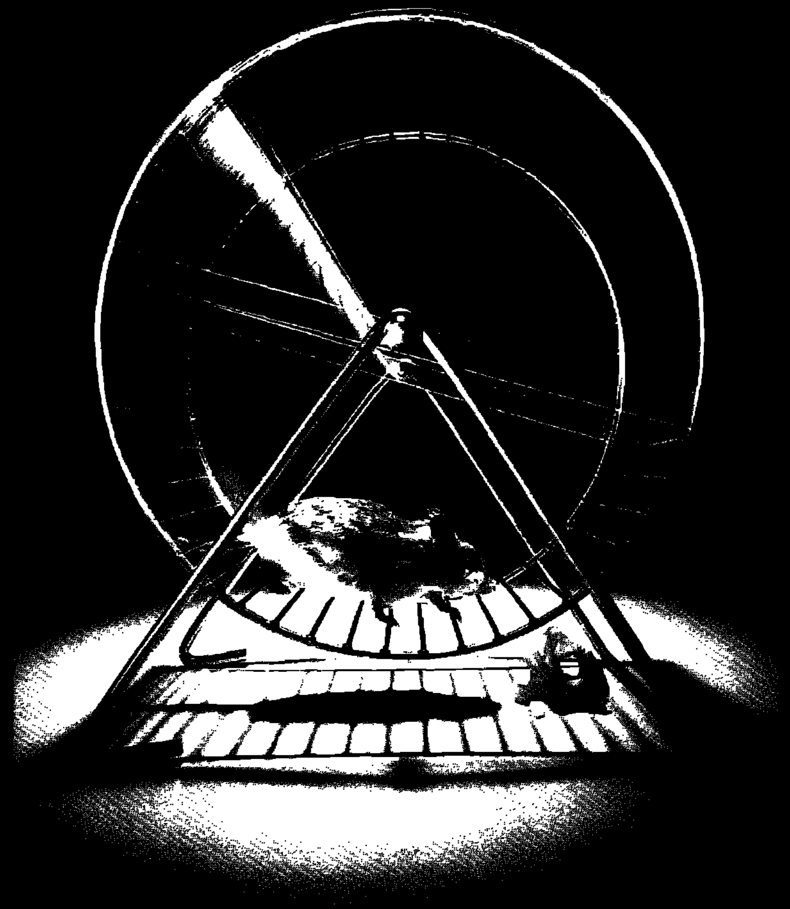
```
!MenuWindow methods !

updateMenus
   "Force my menus to update themselves."
   menus do: [:each |
      each isSmartMenu
         ifTrue: [each update]]! !
```

Listing 3. The class ClassLister.

```
ViewManager subclass: #ClassLister
  instanceVariableNames:
   ' classList selectedClass '
  classVariableNames: ''
  poolDictionaries: '' !

!ClassLister methods !

browseClass
   "Browse the selected class."
   self selectedClass edit!

buildClassMenu
   "Private - Build and answer the class menu."
   ^SmartMenu new
      owner: self;
      title: 'Class';
      appendItem: 'Class Is Variable'
         selector: #classIsVariable;
      appendSeparator;
      appendItem: 'Browse'
         selector: #browseClass;

      enableItem: #classIsVariable when: [false];
      checkItem: #classIsVariable when:
         [self selectedClass notNil
            and: [self selectedClass isVariable]];
      enableItem: #browseClass
         when: [self selectedClass notNil];

      yourself!

classList
   ^classList!

classList: aCollection
   classList := aCollection!

clickedClassListBox: selectedItem
   self selectedClass: selectedItem.
   self menuWindow updateMenus!

open
   "Open myself on all the classes in Smalltalk."
   self openOn:
      (Smalltalk rootClasses
         inject: OrderedCollection new
         into: [:sum :each |
            sum
               addAll: each withAllSubclasses;
               yourself])!

openOn: aCollection
   "Open myself on the list of classes in aCollection."
   | pane |
   self
      classList: aCollection;
      label: 'Class List';
      addSubpane:
         ((pane := ListBox new)
            owner: self;
            setName: #classListBox;
            when: #needsMenu
               send: #updateClassListBoxMenu:
               to: self with: pane;
            when: #needsContents
               send: #updateClassListBox:
               to: self with: pane;
            when: #clicked:
               send: #clickedClassListBox: to: self;
            yourself).

   self openWindow!

selectedClass
   ^selectedClass!

selectedClass: aClass
   selectedClass := aClass!

updateClassListBox: aListBox
   "Update the contents of the class list box."
   aListBox
      contents: self classList;
      selection: self selectedClass!

updateClassListBoxMenu: aListBox
   "Update the menu for my class list box."
   aListBox setMenu: self buildClassMenu! !
```

ANNOUNCING
VisualWorks ReportWriter™
For Portable Reports, Just Point And Click!

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

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

ParcPlace®

Alan Knight

Miscellaneous

This month's column covers several unrelated things that don't require an entire column's worth of space. This includes some recommended reading, the announcement of a new version of the Self language, a template of examples for class comment rules used in the APOK tool kit, and a handy bit of code for finding references to objects in ParcPlace Smalltalk.

READING MATERIAL

Books on Smalltalk programming are coming out regularly, but the most interesting Smalltalk-related reading I've come across lately has little to do with programming. It's Alan Kay's "The Early History of Smalltalk," part of the Second History of Programming Languages conference (HOPL-II) sponsored by the Association for Computing Machinery (ACM) Special Interest Group on Programming Languages (SIGPLAN).

The conference covers the early history of more than 14 different programming languages, including Smalltalk and C++ (SIMULA was covered in the first HOOPLA). I naturally thought the Smalltalk article was the highlight, but I found the whole thing very interesting. I include a few choice quotes from the article:

> One way to think about progress in software is that a lot of it has been about finding ways to late-bind, then waging campaigns to convince manufacturers to build the ideas into hardware.
>
> A language I now called "Smalltalk"—as in "programming should be a matter of..." and "children should program in...." The name was also a reaction against the "Indo European god theory," where systems were named Zeus, Odin and Thor, and hardly did anything. I figured that "Smalltalk" was so innocuous a label that if it ever did anything nice people would be pleasantly surprised.
>
> ...I think the enormous commercialization of personal computing has smothered much of the kind of work that used to go on in universities and research labs, by sucking the talented kids towards practical applications.
>
> Should we even try to teach programming? I have met hundreds of programmers in the last 30 years and can see no discernible influence of programming on their general ability to think well or take an enlightened stance on human knowledge. If anything, the opposite is true.

I received the preprints of the conference papers as a special issue of SIGPLAN notices. I understand, however, that a book including these papers and much other conference material is being planned. Unfortunately, it won't be published until sometime in 1995. In the meantime, the preprints are available from the ACM: US $27 for ACM members and US $54 for nonmembers. The order number is 548931, ISBN 0-89791-5704. The ACM publications office can be reached at 800.342.6626 or 212.626.0500 or at P.O. Box 12114, Church Street Station, New York, NY, 10257 USA.

SELF

Self is a prototype-based experimental language similar in many ways to Smalltalk, with a very aggressive optimizing compiler. It's often mentioned in discussions about optimizing Smalltalk. If you're interested in checking it out for yourself and have a Sun workstation handy, Version 3.0 was recently announced. Excerpts from the announcement follow:

> The Self Group at Sun Microsystems Laboratories, Inc., and Stanford University is pleased to announce Release 3.0 of the experimental object-oriented programming language Self....
>
> Designed for expressive power and malleability, Self combines a pure, prototype-based object model with uniform access to state and behavior. Unlike other languages, Self allows objects to inherit state and to change their patterns of inheritance dynamically. Self's customizing compiler can generate very efficient code compared to other dynamically-typed object-oriented languages.
>
> The latest release is more mature than the earlier releases: more Self code has been written, debugging is easier, multiprocessing is more robust, and more has been added to the experimental graphical user interface which can now be used to develop code. There is now a mechanism (still under development) for saving objects in modules, and a source-level profiler.
>
> The Self system is the result of an ongoing research project and therefore is an experimental system. We believe, however, that the system is stable enough to be used by a larger community, giving people outside of the project a chance to explore Self.
>
> This release is available free of charge and can be ob-

tained via anonymous ftp from Self.stanford.edu. Also available for ftp area number of published papers about Self.

There is a mail group for those interested in random ramblings about Self: Self-interest@Self.stanford.edu. Send mail to Self—request @self.stanford.edu to be added to it (please do not send such requests to the mailing list itself!).

Self currently runs on SPARC-based Sun workstations running SunOS 4.1.x or Solaris 2.3. The Sun-3 implementation is no longer provided.

❝ I have met hundreds of programmers in the last 30 years and can see no discernible influence of programming on their general ability to think well. ❞

CLASS COMMENT RULES

Part of ParcPlace's Advanced Programming ObjectKit (APOK) is a class reporter that includes checking parts of class comments against some simple rules. This is good. In my experience, there are far too many classes out there in real products that have no comments at all. Of those that do, far too many are inadequate or out of date. Let's face it. When you're on a roll, writing detailed comments to explain everything just slows you down. Going back afterward and figuring out which comments need to be fixed is tedious and annoying. It's too easy to just skip it and let things get a little bit out of sync. You can fix it later. Naturally, later never arrives. To help prevent this, comment checking should be an important part of any code review. A tool to make some of this checking automatic is a godsend for reviewer and programmer alike.

The rules that it checks, while relatively simple, aren't that well documented. It always seems easier to just refer to other class comments for examples (I have to congratulate ParcPlace for actually having class comments and for following their own rules). In the interests of making commenting easier, Niklas Bjoernerstedt (nbt@funsys.se) has written some templates for these rules. They are reproduced below:

```
Instance Variables:
    bufferType <nil | String class
> Class of the byte-type object used to store data from the io
Connection
ClassVariables:
    IM90Roots <Set of: IM90Object>
Class Instance Variables:
    activated  <true | false | nil > Holds the activated status of the
application
Pool Dictionaries:
    IOConstants    <Dictionary> of characters keyed by symbols
Subclasses must implement the following messages:
    printing
        defaultRelationType
    class protocol
        instance creation
            newApp
```

FINDING REFERENCES IN PARCPLACE SMALLTALK

Tracking down references to objects using allOwners can be painful. One problem is that the process of searching for references generates more references. This is particularly noticeable in ParcPlace Smalltalk, where it is common to see half a dozen "false" references among one or two real ones. Many of these are arrays, and it can be troublesome to pick out which are legitimate references and which are artifacts of the search process.

To help filter out these false references, Jan Steinman (jan.bytesmiths@acm.org) provides the following bit of code. This should be added as a method in Inspector, and the fieldMenu method modified to call it (don't forget to evaluate Inspector flushMenus)

```
inspectHolders
    "Inspect all who have a reference to the fieldobject. Do not include
        references generated by the reference gathering process!"

    (self fieldValue allOwners reject: [:each |
    "don't include the object under inspection (it always contains the
        field object)"
        each == object
    "don't include this inspector (in case the field is 'self')"
        or: [each == self
    "don't include the methods that got us here"
        or: [(each class == MethodContext
            and: [each selector == #allOwners
            or: [each selector == #allOwnersWeakly:]])
    "don't include the stack array with a temporary variable containing
        the object"
        or: [each class == Array
            and: [each size = 12
            and: [each first == false
            and: [(each at: 2) == self fieldValue]]]]]]) inspect
```

This isn't foolproof (in a pathological case, it could filter out a legitimate reference), and it doesn't do much filtering that couldn't be done manually, but I find it very useful. It clears away the clutter and allows you to concentrate on the important references. I added one extra line, each==self, to this method because often I seemed to be using it on inspectors with the field for self selected giving me an extra reference from the inspector itself.

Note that in ParcPlace, Smalltalk references sometimes hang around awhile after they should be garbage, presumably due to WeakArray references. This can be confusing, so when in doubt, force a garbage collect from the Launcher and see if it helps. ■

Alan Knight is an object person with The Object People. He can be reached at 613.225.8812 or by email at knight@acm.org.

Kent Beck

Where do objects come from? From variables and methods

Let's see if I can get through this third column on how objects are born without blushing. So far we've seen two patterns: objects from states and objects from collections. This time we'll look at two more sources of objects: objects from variables and objects from methods. All four patterns have one thing in common—they create objects that would be difficult or impossible to invent before you have a running program.

These patterns are part of the reason I am suspicious of any methodology that smacks of the sequence, "design, *then* program." The objects that shape the way I think about my programs almost always come out of the program, not out of my preconceptions. Thinking "the design phase is over, now I just have to push on and finish the implementation" is a sure way to miss these valuable objects and end up with a poorly structured, inflexible application to boot.

PATTERN: OBJECTS FROM VARIABLES

Problem: How can you simplify objects that have grown too many variables?

Constraints: It is common to add a variable to an object during development, then add related variables later. After a while, this process of accretion can lead to objects that have many variables. Such objects are difficult to debug, difficult to explain, and difficult to reuse.

Still, the object more than likely works as desired. You'd like to avoid changing code and risking breaking the system for no reason. You will pay a space penalty for breaking the object up, as each object requires an 8 or 12 byte overhead.

Solution: Take variables that only make sense together and put them in their own object. Move code that only deals with those variables into methods in the new object.

Example: The classic example of this pattern is dimensioned numbers. Because Smalltalk doesn't have a built-in framework for dimensioned numbers, programmers often simulate computing with dimensions by storing a value and a dimension together:

```
Class: Page
    variables: lines widthNumber widthUnits heightNumber heightUnits
```

Code has to take the different possibilities for units into account:

```
area
    | widthInches heightInches |
    widthInches := widthNumber *
      (widthUnits = #mm ifTrue: [25.4] ifFalse: [1]).
    heightInches := heightNumber *
      (heightNumber == #mm ifTrue: [25.4] ifFalse: [1]).
    ^widthInches * heightInches
```

The number and units for width don't make sense without one another. Take away one variable and the other no longer is useful. The same is true for height. Both are candidates for objects from variables. First we have to create a Length object to hold both the measure and units:

```
Class: Length
    variables: magnitude units
```

Now the Page can be simplified:

```
Class: Page
    variables: lines width height
```

and the area method can be simplified, too:

```
area
    ^(width * height) inches
```

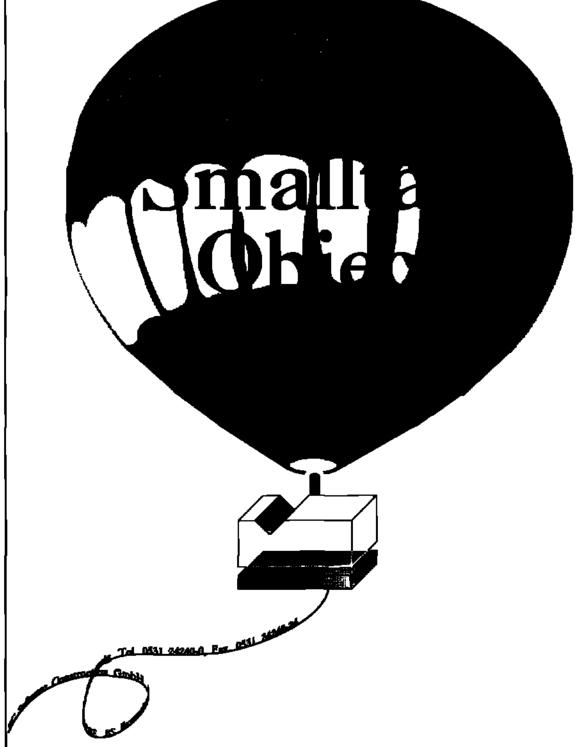
I'll leave the implementation of Length arithmetic as an exercise for the reader and maybe as the subject of a future column.

Once you have Length, you will find many places to use it. The resulting code will be much cleaner, easier to read, and more flexible. If you have to add cubits as a measure, you won't have to visit a hundred methods, you'll just have to fix Length. Following up on object from states, I suppose this is another way to avoid the need for case statements. Rather than build the cases into many different methods, you build it into one object and hide the *caseness* of it.

How can you know when and how to simplify an object that seems to have too many variables? You should obviously avoid the extremes: no object with fewer than two variables will work because you'd never have enough information in one place to write a readable method. All the variables in the world in one object would result in an entirely unreadable, unreusable mess. How can you walk the delicate line between breaking objects up too much and too little?

One telling sign that this pattern is appropriate is when you have two variables in an object with the same prefix and different suffixes. Thus, if you see headCircumference and headWeight as variables, they likely could be factored into their own

ODBMS



ODBMS 2.0
Smalltalk Object Management

Client-Server Architecture
Object Management supporting
Versions, Transactions, Distribution
Multimedia-Objects
Objects to RDBMS

Available as
Single User, Network and Server Version

Supports Smalltalk under
Windows, Windows NT, OS/2, Unix

Successful applications:
Smalltalk Team Development
Personal Data Manager
Configuration of Complex Systems

Objectoriented Technology by
VC Software

USA: VC Software, Houston TX
v: (713) 333-8936, f: (713) 333-3743

Germany: VC Software Construction
Petritorwall 28, 38118 Braunschweig,
v: +49 531 24240-0, f: +49 531 24240-24

USA: Object Power, Harverd MA
v: (508) 456-8854, f: (508) 263-0696
UK: Cocking & Drury, London
v: +44 71 436 9481, f: +44 71 436 0524

object, reducing the original object's variable count by one.

Now for the second pattern du jour, objects from methods. This isn't a pattern I have. (This is a usage that has spread quickly in the pattern community. You'll present a pattern and someone will say, "I have that pattern," meaning they use it, even if they haven't ever articulated it before.) Several people I respect have reported excellent results with it, so I'll do my best to make the case for it. Perhaps there is something else in my programming style that causes me to find these objects another way, or maybe I just never find them. I haven't really thought much about it. Anyway, here is the pattern:

PATTERN: OBJECTS FROM METHODS

Problem: Sometimes you write a method that is too long to read well. Reduction with the usual techniques (e.g., *compose* methods), doesn't seem to make it read any better. How can you simplify methods that resist easy reduction?

Constraints: Creating a new object is one of the weightiest conceptual decisions you can make when programming with objects. You should never make the decision to create one lightly. If the object in question has no obvious counterpart in the problem domain, you should be even more careful. The increased load on downstream programmers is one reason to create as few kinds of objects as possible. The tendency of objects to leak into the user's consciousness is another.

Objects are great for structuring information, particularly information that has a behavioral or computational component. They are good for representing not just the user's view of a program, but the programmer's view as well. When you have tried simpler methods of writing a computation and failed to produce a result that effectively communicates your intent as a programmer, you are justified in creating new objects to simplify your computation.

Methods that are candidates for this treatment have several features in common. First, they are long. Two, three, and four line methods composed out of other provocatively named methods generally communicate well.

Second, they are not easily shortened by splitting them into smaller methods. This may be because the parts of the method don't make sense when separated, or it may be because you have to pass so many parameters to the submethods that you have trouble naming them all meaningfully. The submethods may also need to return two or more values. Finally, such methods often have many temporary variables (resulting in the many parameters to the submethods).

Solution: Create an object encompassing some of the temporary variables from the complex methods that manipulate those variables into the new object. In the original method, create one of the new objects and invoke it.

Example: As I said in the preamble, I don't have a good example of this pattern. I have used object languages that didn't have points, however, and I can imagine discovering them using this pattern. If you have a method that displays a sequence of pictures:

continued on page 21

Scot Campbell

Digitalk's Team/V

Smalltalk is a highly productive programming environment. Starting with the mature base class library, developers add new classes and new behavior to existing classes using the interactive browsers, inspectors, and debuggers of the environment. However, after reaching a given milestone in the development process, the developer is left with the question, "What did I add to the base image?"

Smalltalk teams, as all software development teams, generally have the challenge of delivering high-quality products under tight timetables. Smalltalk teams have the added pressure of "proving" the new technology to management. Shipping a module with a missing method or an uninitialized variable can be detrimental to these ends, and Murphy's Law predicts the certainty of a runtime error occurring from the omission. Team/V, from Digitalk, is team-oriented configuration management software that aids in structuring the project and delivering complete applications.

The other most notable product in this category is ENVY/Manager, from Object Technology International, Inc., which has been previously reviewed in this publication.[1] This article explains the functionality and features of Team/V and highlights areas where it differs from ENVY/Manager.

Team/V addresses the major issues of team development in Smalltalk:

1. Identification of the additions to the image that make up the application.
2. Coordinating the efforts of the development team.
3. Version Control.

TEAM/V CONCEPTS

Team/V adds mechanisms and tools to help structure and manage the work of a team. *Packages* organize the code that help to identify the makeup of the application and coordinate the efforts of individual members. *New browsers* work with packages, and *version control* mechanisms manage changes to packages. Versions of packages are saved in a repository where they can be loaded by other team members as development progresses. Comparison browsers highlight the differences between any two versions of a package.

PACKAGES

The fundamental organizing structure in Team/V is the *Package*. A *Package* is intended to encapsulate one unit of functionality. For example, the set of classes and methods that support asynchronous communications could form a package. In addition to being a unit of functionality, the package is also a unit of sharing, being used by any applications that require the function. By assigning the packages among the developers of a team, a project can distribute the work to be done. A package consists of the following:

- *Name*—for identification purposes (this name appears in the browsers),
- *Annotations*—are simply key/value pairs for commenting the package,
- *Comment*—a predefined annotation,
- *Definitions*—are ordered collections of units of Smalltalk code.

Packages roughly correspond to ENVY Applications, however, there are differences. ENVY applications maintain two relation-

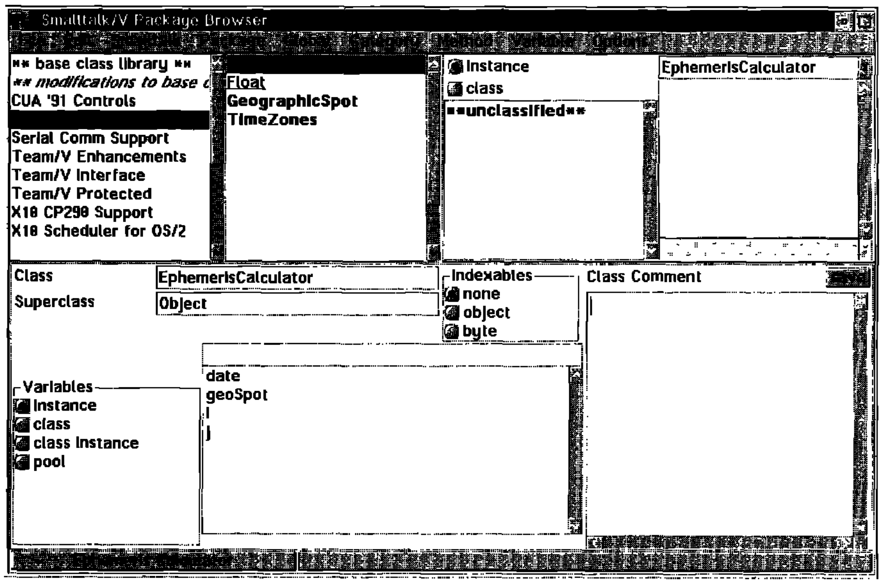
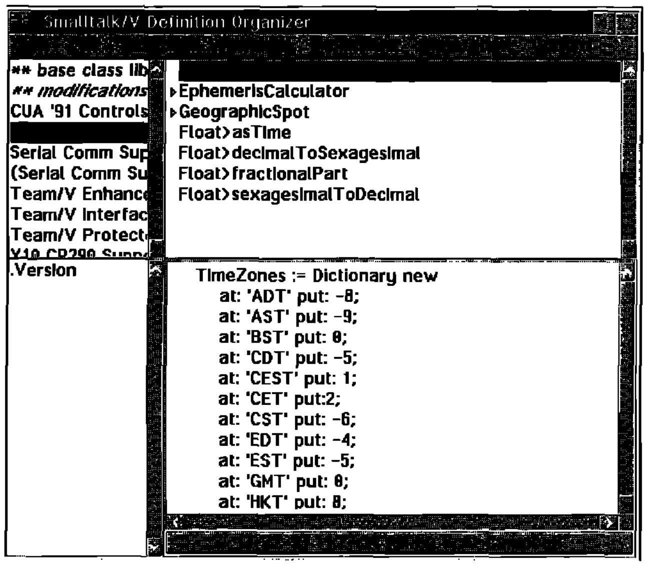


Figure 1. The Package Browser.



```
Smalltalk/V Definition Organizer

** base class lib      ▸EphemerisCalculator
** modifications       ▸GeographicSpot
CUA '91 Controls        Float>asTime
                        Float>decimalToSexagesimal
Serial Comm Sup         Float>fractionalPart
(Serial Comm Su         Float>sexagesimalToDecimal
Team/V Enhance
Team/V Interfac
Team/V Protect
V10 CP290 Supp
.Version                  TimeZones := Dictionary new
                              at: 'ADT' put: -8;
                              at: 'AST' put: -9;
                              at: 'BST' put: 0;
                              at: 'CDT' put: -5;
                              at: 'CEST' put: 1;
                              at: 'CET' put:2;
                              at: 'CST' put: -6;
                              at: 'EDT' put: -4;
                              at: 'EST' put: -5;
                              at: 'GMT' put: 0;
                              at: 'HKT' put: 8;
```

Figure 2. The Definition Organizer.

ships. The first, *prerequisites*, indicate what other applications are required before a given application can be loaded. OTI claims that prerequisites are essential for supporting the concept of pluggable software components while Digitalk claims they limit reusability. Both positions are valid. Anything we add in Smalltalk is based upon what already exists, and in fact an application named "Kernel" (which includes most of the base class library) becomes the default prerequisite for any application we define. And documenting what services are required and enforcing the requirement is helpful, in fact, essential to prevent errors. However, the biggest challenge in effectively using either of these products is coming up with the best configuration of *packages* or *applications*. A poor configuration of applications will lead to overlapping and unnecessary prerequisite relationships that can limit reusability.

The second relationship supported by ENVY applications are *subapplications*, which is a partial relationship. A subapplication is a part of an enclosing application. The most common usage of this is for multiplatform development, where the application contains function common to all platforms, and subapplications contain platform-specific code when it is required. Subapplications can be nested, which allows the project to better organize the classes in a large application.

DEFINITIONS

Definitions are the main component of a package.

An element in the definitions list can be one of the following:

Class—both definition and methods,

Class Extension—methods for a class, no definition (called "*loose methods*" in the Team/V documentation),

Global variable,

Pool Dictionary,

Initializer—an arbitrary Smalltalk expression whose primary purpose is initializing objects.

A new browser, the *Definition Organizer*, provides the capability to order the definitions. By sequencing the definitions, you can ensure that needed initialization is run before any component that requires the initialization. The *Definitions* concept appears more flexible than the collection of *defined* and *extended* classes in an ENVY application or subapplication. The Team/V documentation talks of the elements of a class definition (*instance vars*, *class vars*, *class instance vars*, and the pool usage), as discrete items in the definitions. While this is currently not possible, they have plans to support class extensions that contain more than just a set of methods. You could then have an extension that added some methods and a needed instance variable, required only by the extension.

The other area where definitions appear more flexible is in the ordering of the definitions. ENVY adds protocol to perform initialization before or after an application is loaded. The *loaded* and *removing* methods are commonly implemented for applications in ENVY. The *loaded* method can be used for initializing classes in the application. Team/V defines a *class initializer*, which puts the initialization with the class. The ENVY *removing* method is commonly used for cleaning up the image when an application is unloaded, such as removing a global variable. In contrast, with Team/V, because the global is part of the definition, when a package is deleted, any unreferenced globals defined in the package are removed automatically.

An example *ad-hoc* initializer given in the tutorial tests the package after it is loaded. This rigor would go a long way in testing prerequisite conditions, ensuring that the image the package is loaded into will support the requirements of the package.

VERSION CONTROL

The Glossary that comes with the Team/V documentation defines a *revision* as "One of potentially many incarnations of a given package" while it defines a *version* as "One of potentially many incarnations of a given definition." The distinction is not critical; however, I will attempt to use them in their correct context.

When a package is "committed" the package is stored in a repository under a specified revision number. Team/V supports repositories managed with Intersolv's PVCS or the file system of the OS being used. The PVCS repositories are more space efficient, as only the delta between revisions is stored, and all revisions can be stored in one archive. PVCS also includes access control mechanisms to limit who can make a new revision of a package and mechanisms to deal with concurrent commitment. You do not have to separately purchase PVCS to use the PVCS archives; Team/V ships with support that allows it to utilize PVCS archives.

ENVY defines component ownership, classes have owners and applications have managers. Although any developer can modify a class, it is the owner who maintains the major path of development for the class. The application manager is the only one who can *release* an application, thereby making it available for others.

With the purchase of the full PVCS product, Team/V users can put controls on who can modify or release a package. Without having the full PVCS product, I was unable to determine if PVCS provides the flexibility or dynamics provided by ENVY.

OTHER FEATURES

Class Definition

Class definition is done through a formatted pane on the package browser. In addition to naming the class, its superclass, instance variables, class variables, and pools that are supported in Smalltalk/V, Team/V adds Class Instance Variables, Comments (for both the class and variables), and class initialization. Message categories are also implemented for classifying, or grouping methods. Although the notion of public and private methods is not directly implemented, as in ENVY, private methods can be placed in a category named or containing "Private," which would provide the benefits of documenting the public/private protocol of the class.

Conflict Resolution

When a package is loaded Team/V first checks for any conflicts. A conflict occurs when one of the definitions of a package you are trying to load has the same name as one of the definitions already loaded in your image. A conflict also occurs if you attempt to load a class extension for a class that is not defined. Conflict resolution only occurs when you load a package. Using the Install/File-In menu items does not perform conflict resolution. However, with the Package/Load operation one can select any source file and have the conflict resolution performed when filing in source in nonpackage format. Team/V then generates a package named "From: <filename>," which you can then rename to an appropriate name.

NEW BROWSERS

Package Browser

The *Package Browser* is a new tool for working with Packages (See Fig. 1). The package browser has four panes at the top. The top left pane shows the packages in the image. Packages that are only opened and not loaded are enclosed in parentheses. Open packages are packages that are only open for inspection; they are not part of the executable image. Open packages are useful for comparing with other revisions of the same package or any other package. You can also modify and commit a new version of an open package. In this way you can resolve a conflict in a package you are attempting to load.

The second pane from the left shows global definitions (classes, global variables, and pool dictionaries) for the selected package. The third pane from the left shows the message categories for the selected definition if the definition is a class. And finally the right pane shows the methods for the selected category, if a category has been selected. A nice feature of the method list is that it can show inherited methods in addition to those defined within the selected class.

The lower half of the package browser shows various aspects of a definition, a class definition (as shown in the figure), method source if a method is selected, or any of the comment fields that the system supports. This pane is the primary place for modifications to be made. In fact, the Class Hierarchy Browser has been made read-only in Team/V. Source code formatting for method source is also available from the package browser.

Some of the panes support direct manipulation (drag/drop) for moving components around. However, because the panes are not multiselect, I found this to be tedious at times. For example, a definition that defined a class could be moved from the definitions list to a different package, but a definition that was only a class extension could not. In this case, each method had to be moved individually. This tedium occurs mainly when you are filing in external source while constructing your packages for the first time. ENVY defines a *default application* that receives all filed-in code. I found the default application mechanism to be more convenient.

The browsers show modified components in italics and a different color, defined classes are in bold while class extensions are not. These visual cues are excellent for grasping what changes have been made since the last commit.

Definition Organizer

The *Definition Organizer* lets you examine the definitions within a package in the order in which they will be initialized. You can reorder the definitions as necessary. The top left pane shows the packages currently loaded or opened, the top right pane shows the definition list for the selected package, the lower left pane shows the annotations for the selected package or definition and the lower right pane shows the source for the selected definition or annotation (See Fig. 2).

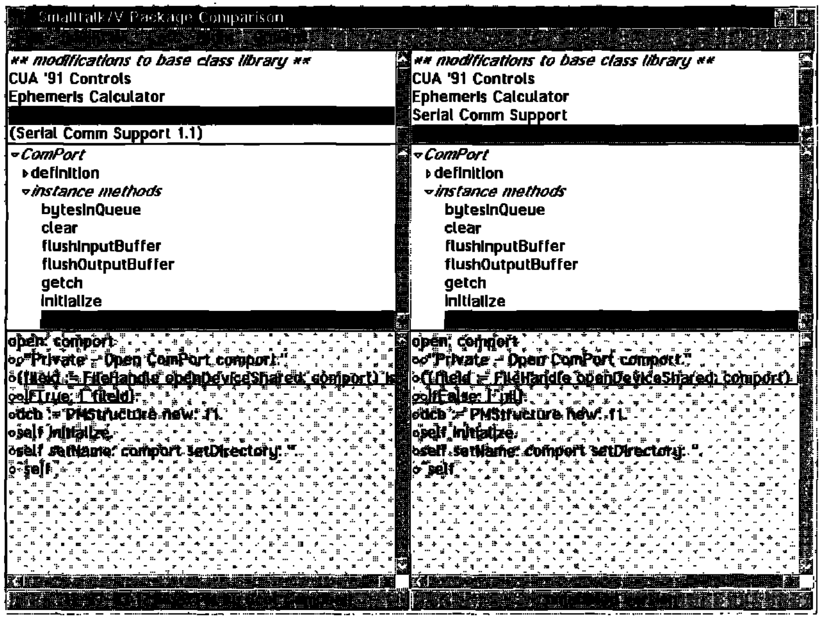
While the Package browser only allows specification of global definitions only (class, pool and global variable definitions), the *Definition Organizer* allows all types of definitions to be specified. Only the definition organizer allows for the *ad hoc* initializers to be added. The Definitions Organizer is also the tool to use for annotating a Package.

Definition Group Browser

The *Definition Group Browser* has the same appearance as the familiar method browser so I have not shown it. The main difference is that the contents can be any definition (e.g., method, class, initializer, etc.) rather than just methods. A nice feature added to the Smalltalk menu is *Browse—>Modified...*which brings up a *Definition Group Browser* on all definitions that have been modified since you last committed the packages.

History Browser

The bottom right of the Team/V browsers contain a button labeled with version information for the given definition. Clicking on this button brings up a *History Browser* that contains all the known versions of the definition. The versions known are only those since the last compress of the changes



Smalltalk/V Package Comparison

```
** modifications to base class library **   ** modifications to base class library **
CUA '91 Controls                             CUA '91 Controls
Ephemeris Calculator                         Ephemeris Calculator
                                             Serial Comm Support
(Serial Comm Support 1.1)

▿ComPort                                     ▿ComPort
  ▹definition                                  ▹definition
  ▿instance methods                            ▿instance methods
    bytesInQueue                                 bytesInQueue
    clear                                        clear
    flushInputBuffer                             flushInputBuffer
    flushOutputBuffer                            flushOutputBuffer
    getch                                        getch
    initialize                                   initialize

open: comport                                open: comport
"Private - Open ComPort comport."             "Private - Open ComPort comport."
(fileId := FileHandle openDeviceShared: comport) is   (fileId := FileHandle openDeviceShared: comport) i
ifTrue: [^fileId].                           ifFalse: [^nil].
dcb := PMStructure new: 11.                  dcb := PMStructure new: 11.
self initialize.                             self initialize.
self setName: comport setDirectory: ''.      self setName: comport setDirectory: ''.
^self                                        ^self
```

Figure 3. The Package Comparison Browser.

file. This differs from ENVY where every save of a method saves an edition in the repository that can later be browsed and compared with other editions. I expected to have versions of methods in *opened* packages available in the History Browser, but they were not. Not being able to have history directly available for versions of methods across versions of packages is a definite disadvantage.

COMPARISON TOOLS

Team/V allows you to compare two versions of a method, or two revisions of a package.

Method Comparison Browser

Selecting *Method—>Compare* from a Package Browser or *Version—>Compare* from a History browser brings up a *Method Comparison Browser.* The Method Comparison browser is basically two method browsers side by side. The differences between the two selected versions of the method are shown both in a different color and underlined.

Package Comparison Browser

Team/V allows you to compare two versions of a given package, or you can compare two completely different packages (See Fig. 3). The top two panes are the revisions list; revisions of all packages appear. The middle two panes are the definitions list. Definitions that are in one revision but not the other are in boldfaced text. Definitions that are in both are in italics. The bottom panes are the contents panes with differences underlined.

The granularity of versions is finer in ENVY than Team/V. ENVY maintains versions of methods, classes, subapplications, applications, and Configuration maps (groupings of applications) in the repository while Team/V maintains only versions of packages.

Documentation

Team/V's documentation is very good and easy to use. In addition to the explanation of how the new browsers work, there is a chapter that explains the concepts behind Team/V and a tutorial that acquaints the new user with the development process under it. One very useful chapter answers questions that arise during the development process. Questions like, "How do we divide up the work of the application?" (i.e., along what lines should we define our Packages?).

The Team/V Programmatic Interface

The Programmatic Interface is a mapping of the semantic components of Team/V into classes that you can use to create your own custom tools and browsers. Digitalk's intention is to keep this interface compatible in future releases of Team/V. Multiplatform development could be built using this interface and an annotation convention on packages. Using an annotation named something like "Platform" and having values that indicate the intended platform for the package (e.g., "All," "PM," "Win," etc.) one could extend the system with a "smart" export to have something like ENVY application line-ups, thereby providing some amount of multiplatform support.

OTHER DIFFERENCES

ENVY defines configuration maps that are groupings of applications that other developers can load to bring their image up to a given point in the development process. Configuration maps are stored in the repository as are all components in ENVY. Team/V does have a *Build Script* operation that will build a script that a developer can run to load current packages into her image. But because they are not stored in the repository, the management would not be as easy.

A particularly hot issue of late on the CIS Digitalk forum has been image size. ENVY includes a packager that will create a separate runtime image containing only the code that is required at runtime. One drawback of this is that changes that Digitalk makes to its implementation of Smalltalk affect the packager. There was a long delay between the release of Smalltalk/V OS/2 2.0 and a packager that worked for it. One reason you may want to stay with a single source for tools.

Team/V is not supported on ParcPlace's Smalltalk. Many large companies have projects using both the Digitalk and ParcPlace versions of Smalltalk. With ENVY these projects could share a common repository.

CONCLUSION

Both Team/V and ENVY share all the mechanisms to support team development and version control. If you are delivering a standalone commercial application on multiple platforms, ENVY with its packager for producing minimal sized delivery sets and application line-ups for supporting multiplatform development seems better positioned. If on the other hand you are in an enterprise that has standardized on a platform and you are delivering many Smalltalk applications that can share base class libraries, you can easily package your applications in object libraries and have small images that bind to needed libraries and open the required user interface. In this case, Team/V provides all the needed tools. ■

Reference

1. Steiman, J., and B.Yates, "Product review: Object technology's ENVY developer," THE SMALLTALK REPORT, 2(2), 5–11, October 1992.

Scot Campbell is a Smalltalk contractor in California. Before contracting, Scot developed programming tools at Chevron Information Technology Co. Scot can be reached at scot@netcom.com or CISD: 70641,2501.

■ **SMALLTALK IDIOMS**

continued from page 16

```
display
| x y |
   x := y := 0.
   10 timesRepeat:
      [picture displayAtX: x y: y.
      x := x + 2.
      y := y + 2]
```

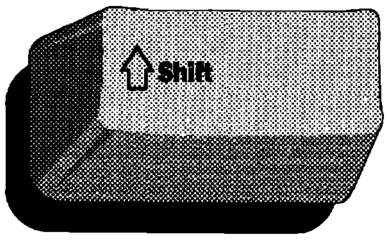
Using objects from methods, we notice that x and y are used together. We create a point object with x and y variables. We can then simplify the above method to the following:

```
display
   | p |
   p := Point x: 0 y: 0.
   10 timesRepeat:
      [picture displayAt: p.
      p := p + 2]
```

I don't find this example compelling, but if you had an algorithm that used a half dozen points, you could easily get lost in the thisX, thisY, thatX, thatY's. The transformation would make much more difference.

Ward Cunningham told me a story of using this pattern on a piece of financial software. There was one method that was long and ugly, but it was important because it computed the value of a bond at a given moment. As soon as they turned the method into its own object, the computation came into focus. These advancer objects became the centerpiece of their caching strategy.

In my next column, I will end my series on the origin of objects by examining two common patterns for finding objects: objects from the user's world and objects from the interface. ■



Shift Over To CPU!

If you're an experienced Smalltalk professional, shift your career over to the exciting world of consulting with Computer People Unlimited. We're the Midwest's premier professional software services firm handling a full spectrum of engagements from staff supplementation to complex turnkey projects. Currently, we have opportunities available for experienced:

SMALLTALK DEVELOPERS

You will be rewarded with a highly competitive salary and complete benefits. To find out more about these Midwest opportunities, please call or fax your resume to:

Appleton/Green Bay --- Marv Miller
Phone: 1-800-960-1278 or Fax: 414-738-4499
Milwaukee --- Julie Endlich
Phone: 1-800-527-8462 or Fax: 414-225-4011
Madison --- John Manyo
Phone: 1-800-280-2782 or Fax: 608-223-3810
Minneapolis --- Scott Fleischmann
Phone: 1-800-278-2850 or Fax: 612-338-3210

You may also send your resume to: Computer People Unlimited, Dept. SR, 732 N. Jackson Street, Milwaukee, WI 53202. Sorry, no entry level positions available. We are an equal opportunity employer.

CPU
COMPUTER PEOPLE UNLIMITED INC.
Building On Our Strengths.

RECRUITMENT TO PLACE AN AD, CALL SHIRLEY SAX AT 212.274.0640

Smalltalk Software Developers

Advanced Technology Systems currently has permanent positions available for **Smalltalk Professionals** to work in Charlotte, NC. Qualified individuals will possess at least 1 year of application experience.

ATS offers excellent salaries and company-paid benefits. For consideration, please forward resume to: **Advanced Technology Systems, Inc., Suite 2460, 301 S. Tryon Street, Charlotte, NC 28282; FAX (704) 376-7369.** An Equal Opportunity Employer.

ATS

Advanced Technology Systems, Inc.

OBJECT EXPO NEW YORK JOB FAIR

JUNE 7, 11–6:30 ■ JUNE 8, 11–6:30
JUNE 9, 9am–2pm

DataLink Corporation and other top recruiting, consulting and hi-tech firms will be at the East Coast's largest OT specific expo looking for YOUR object-oriented expertise and skills. The OBJECT EXPO NEW YORK JOB FAIR, located in a private area on the exhibits floor, allows you to talk one-on-one with recruiters about the best nationwide career opportunities for OT professionals.

BRING YOUR RESUME—RECRUITERS WANT TO MEET WITH YOU!

Fax to 212.274.0899 Dept. OE for a FREE exhibits pass to attend the Job Fair.

micado

micado SoftwareConsult GmbH is one of the leading system houses in Germany for object oriented languages. It has an expert team with wide experience in development and customer support. Due to the astounding growth of the object oriented market in Germany, we are currently seeking the following freelance OO professionals:

Smalltalk Designers and Developers

If you welcome new challenges and if you want to explore your career opportunities please send or fax your resume to

micado SoftwareConsult GmbH
Reutherstr. 1a-c D-53773 Hennef
Tel. (49)2242-871-450 FAX -455
Compu-Serve 100024,2444

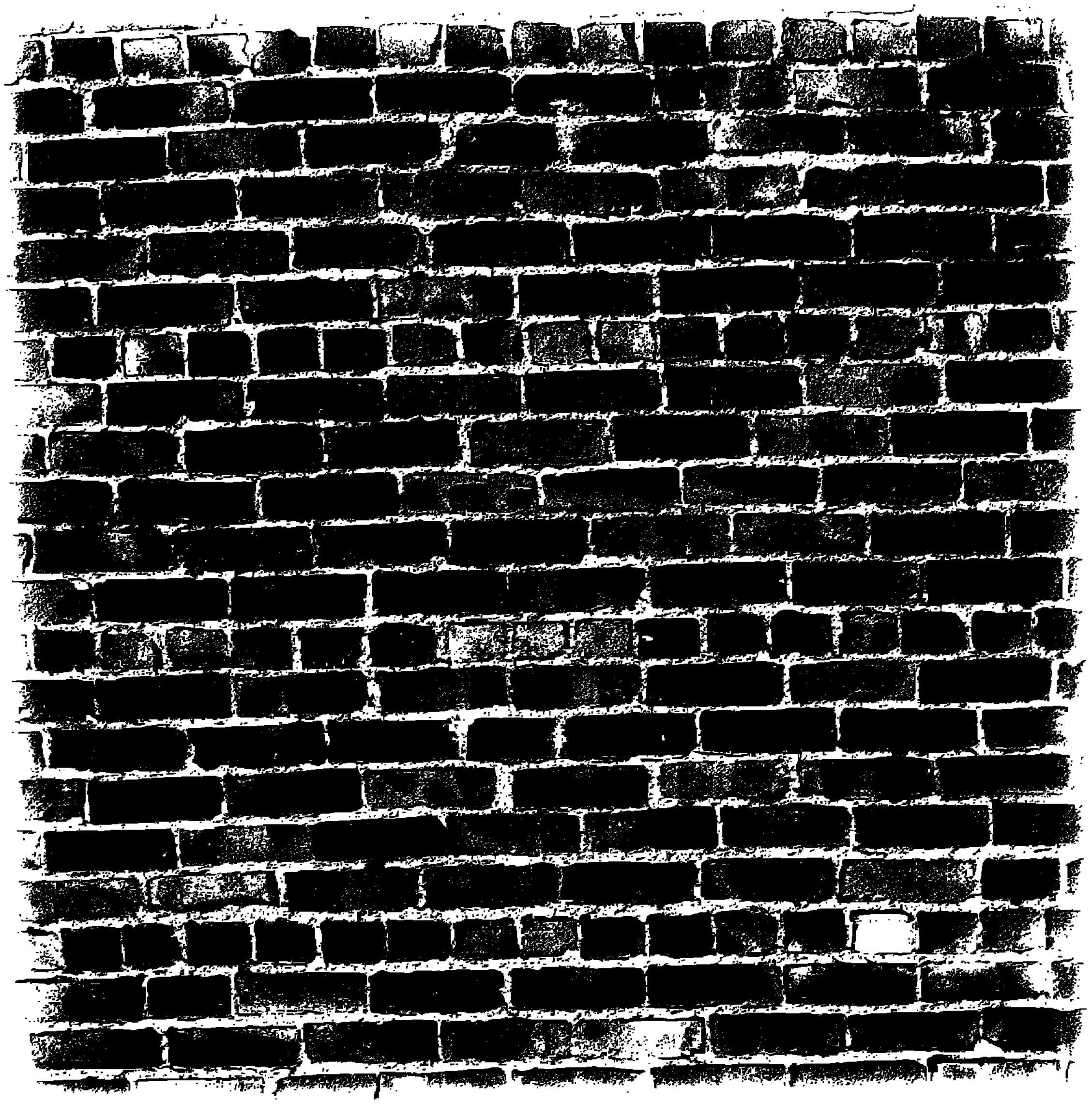
Software Engineering

PRO-STAR

PRO-STAR is a fast growing management and information systems consulting company specializing in applications development. We provide services to our clients that support all phases of the systems development life cycle. We have expertise in virtually all hardware and software technologies.

We have significant client/server development projects underway. We need to add senior smalltalk developers to our staff. Experience with ENFIN is a plus. If you are a talented, hard working information technology professional, come join our team in Sacramento, California. Please respond to:

PRO-STAR, a division of PRODATA, Inc.
6929 Sunrise Blvd., #210
Citrus Heights, CA 95610
(916) 969-0176
FAX: (916) 722-1045
rfellows@netcom.com



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.

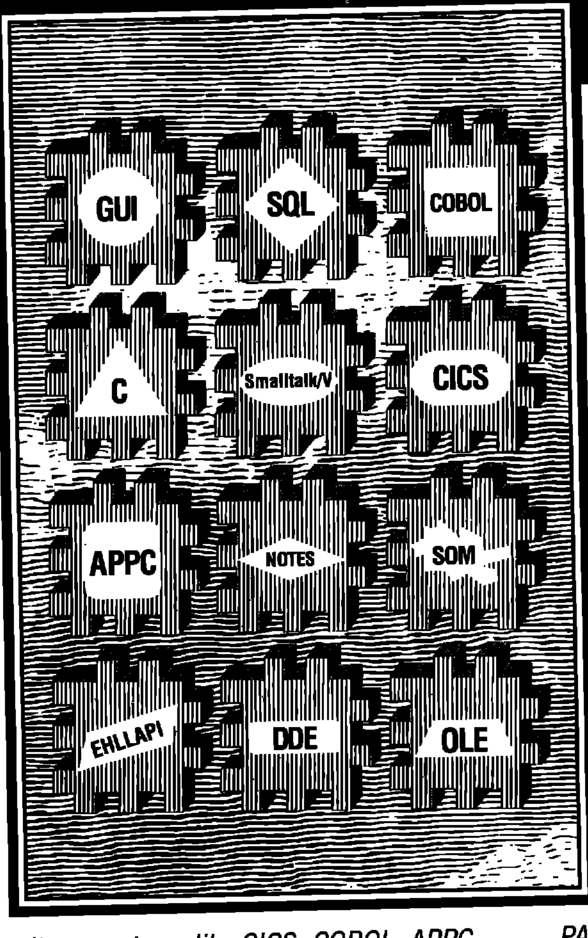
Servio™ has a better way. With our high-performance GemStone® object database management system, you can store Smalltalk objects directly in the database. We make your development time more productive and your object applications more efficient.

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