The Smalltalk Report

The International Newsletter for Smalltalk Programmers

November/December 1992 Volume 2 Number 3

TAKING EXCEPTION TO SMALLTALK, PART I

By Bob Hinkle & Ralph E. Johnson

Contents:

Exception handling is an important part of many languages. Although not provided in the original Smalltalk-80 or in Smalltalk/V, it is supported in the latest version of ParcPlace's Smalltalk-80. This article will show how to build an exception handler for any version of Smalltalk and will use Smalltalk/V 286 as an example. Along the way, we'll show you why it's useful for languages to treat seemingly internal mechanisms such as processes and contexts as first-class objects.

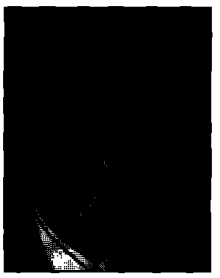
The exception handler was first built for an early version of Tektronix's Smalltalk-80. It was modeled after a version described in an article by Evelyn Van Orden,[1] and we used it in the type inference system of Typed Smalltalk.[2] When we ported Typed Smalltalk to ParcPlace Smalltalk, we wanted to use their faster exception handler, so we modified ours to be compatible. Thus, our exception handler is similar to ParcPlace's, but less powerful. We then developed the V 286 version described here, both to test the generality of the solution and to make the work interesting to a wider audience.

A QUICK LOOK AT EXCEPTIONS

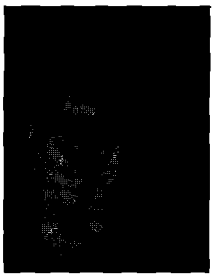
Briefly speaking, exception handling is the provision for non-lexical flow of control in a program when something out of the ordinary (i.e., exceptional) occurs. An exception handler is a part of the program (usually a block in Smalltalk) that can deal with some possible but unlikely event, such as reading past the end of a file, dividing by zero, or referencing out of bounds in an array. In the usual scheme, a program registers an exception handler for a particular kind of event and then continues with its normal processing. If an exceptional event does occur, a signal is raised as a notification to the system. The system finds the last handler that was registered for that signal by searching down the context stack. If one is found, control passes into the exception handler. Depending on the system, the handler will have different options. The handler can usually make whatever changes are necessary; execution can then resume where the signal was raised or where the handler was registered, or return from where the handle was registered.

This description shows that implementing an exception handler requires access to processes and their context stacks. An exception needs to search the context stack to find the correct handler for a given signal and implement non-local control flow. As a result, exception handling could only be added to traditional languages by the language designer. In Smalltalk, however, where processes are objects and contexts can be objects, exception handling can be added by a programmer. Smalltalk's first-class treatment of contexts is one aspect of a concept called *reflection*, which is the idea that languages and systems should objectify their internal mechanisms to make them accessible to the programmer. In that way, programs can monitor and change their behavior, in a sense reflecting on themselves. Our example of exception handling shows how some reflectiveness makes a language more adaptable.

continued on page 3...



John Pugh



Paul White

EDITORS' CORNER

Another OOPSLA has come and gone. This conference represented a significant milestone, both personally (since it's finally done and behind us!) and as Smalltalk users. Based on this conference, it would appear the language wars of the past are now over. Smalltalk is definitely well-entrenched as the language of choice within many organizations and few, if any, of the so-called research-language–type complaints about Smalltalk were to be found. Smalltalk has clearly made it.

Interestingly, the void left by the language wars seems already to have been filled by a full-fledged, drag-em-out war over methodologies. It seemed there were nothing but methodology tools vendors on the exhibit floor. Many were designed specifically for methodologies such as Booch or Rumbaugh, while others were "applicable to all methodologies" (which, of course, more often than not means "useful for none").

Two aspects of this methodology war are worth noting. First, it is not clear that any one will emerge as the winner. That is not such a bad thing. Just as no one language is appropriate for all applications, even within an organization, no one methodology should be applied universally. Like the language wars before it, though, this plea for reason and tolerance will likely be lost among the battle cries.

The second and more subtle aspect of this war is that these methodologies seem better geared for the C++ world. Smalltalk developers seemed, for the most part, removed form the debate. They talked much more about tools that would help you deliver and much less about methodologies. We will have more to say on this subject and the need for better tools that go beyond any particular methodology in future issues.

It is with great pleasure we introduce Ralph Johnson and Bob Hinkle, two well-known members of the Smalltalk community, as our featured writers this month. Over the next few issues, they will address in detail the issue of exception handling using Smalltalk. This is a topic important to all computing languages and one that is often misunderstood. In their opening article, they describe the interface for their exception handler, along with the machine-independent aspects of its implementation.

Also in this issue, Kent Beck continues his survey of the Collection classes, highlighting interesting facts about many of the more popular classes. Rebecca Wirfs-Brock speaks about the need for properly described classes and applications. Juanita Ewing describes a straightforward mechanism for managing source code on small projects. Greg Hendley and Eric Smith survey the events supported by PM's Pane classes. Richard Peskin reviews John Bourne's new textbook, written for engineering programs that introduce the object-oriented paradigm. Finally, Alan Knight returns with more discussion from the USENET world.

Happy holidays to all!

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

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

This article and its sequel next month present a Smalltalk implementation of exception handling. This month, we'll describe the system's interface and the machine-independent aspects of its implementation. Next month, we'll complete the picture by describing the V 286-specific implementation.

THE EXCEPTION HANDLING INTERFACE

At the heart of the exception handling system are the classes Signal and Exception. An instance of Signal represents an exceptional event that might occur and its most important methods, handle:do: and raise. Sending handle:do: to a Signal object registers a block that can be evaluated if that event occurs. For example, suppose OutOfBoundsError is a global variable that holds a Signal object. As the name implies, this signal is intended to signify out-of-bounds references in arrays and might be used in a method of class Array as follows:

```
checkFifthElement
    OutOfBoundsError
        handle: [ :exception | ^self handleException: exception]
            do: [ ^self at: 5]
```

The effect of handle:do: is to evaluate the second parameter (do: block), with the addition that a raised OutOfBoundsError will be handled by evaluating the first parameter (handle: block). So, as you might expect, evaluating #(1 2 3 4 5) checkFifthElement will return 5, but evaluating #(1 2 3 4) checkFifthElement will cause the block [:exception | self handleException: exception] to be evaluated. What happens next depends on Array>>handleException: It might define a default value for that array, prompt the user for information, or form some other appropriate response.

For this scheme to work, the system must use OutOfBoundsError to signify the out-of-bounds condition. This can be done by sending the raise message to OutOfBoundsError in the midst of at: (and methods like it), as follows:

```
at: anIndex
    <primitive: 60>
    (self outOfBounds: anIndex)
        ifTrue: [^OutOfBoundsError raise]
```

One interesting aspect of the handleException: message is its parameter exception, which is an instance of the class Exception. Each time a signal is raised, a new exception is created to objectify that fact. The exception is a convenient place to encapsulate information about both the signal and the context in which it was raised. Particular error information or a special error message can be associated with an exception by using variations of the raise message, in this case raiseWith: and raiseErrorString:, respectively. In this way, an exception handling block can learn a great deal about the error by accessing the exception, which allows it to respond more intelligently.

In addition, class Exception provides support for common exception-handling techniques, including the messages proceed, reject, restart, and return. When an exception proceeds, control resumes in the context where its signal was raised, and a value can be returned if desired. This is how a new default value can be defined for an array. Thus, handleException: could be implemented as:

```
handleException: anException
    anException proceedWith: 'Bob'
```

This will cause the string 'Bob' to be returned as the value for any index outside the array's bounds. In addition to proceed, you can send restart to an exception, which causes the handle:do: context to be restarted, or send return, which causes the handle:do: message itself to return, again with the option of returning a specified value. Finally, sending reject to an exception is a way of saying that the current handler can't solve the problem. The system looks for the next handle:do: context down the stack that can handle the signal and evaluates its handle: block. These possibilities are illustrated in Figure 1.

For the purposes of this example, we assume that Array>>foo is implemented as:

```
foo
    Transcript show: self checkFifthElement printString
```

Now, if #(1 2 3 4) foo is selected and evaluated, when fetchHandlerBlock returns, the context stack will be as shown in Figure 1, with the exception's instance variables signalContext and handlerContext referring to the indicated contexts.

There are several ways to define Array>>handleException:. One possibility is for it to proceed from the exception, as in

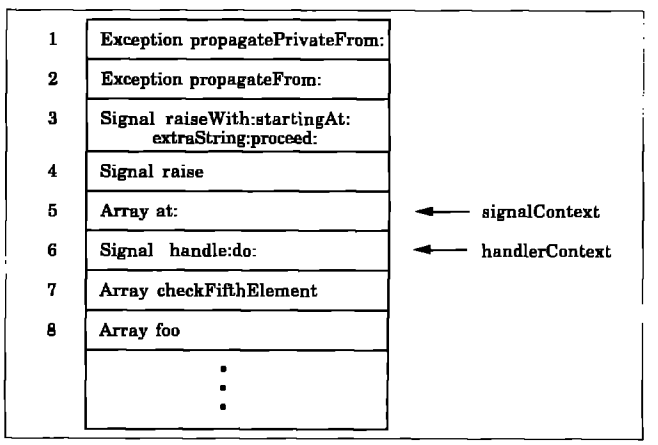


Figure 1. Stack during exception handling.

```
handleException: exception
    exception proceed
```

In this case, when the handle: block of handlerContext is evaluated, nil will be returned as the value of the Array>>at: message send, the fifth context on the stack, and execution will proceed in the sixth context. However, if handleException: is defined as

```
handleException: exception
    exception return
```

then nil will be returned as the value of the Signal>>handle:do: message send corresponding to the sixth context on the stack, and execution will proceed in the seventh context. Using restart, as in

```
handleException: exception
    exception restart
```

will cause the handlerContext, the sixth context on the stack, to be restarted from the beginning, in effect reevaluating the do: block. Finally, the exception handler may reject the Exception, as in

```
handleException: exception
    exception reject
```

In this case Exception>>propagatePrivateFrom: will be called again, but this time the search for a handler will proceed downward from the context just below the handlerContext, in this case the seventh one on the stack.

> “Briefly speaking, exception handling is the provision for non-lexical flow of control in a program when something out of the ordinary (i.e., exceptional) occurs.”

There is one final part of the system that interacts with exception handling, though it's not implemented in either of the above two classes. This feature is something called an unwind mechanism, which is a way for a programmer to ensure that certain actions are performed, even if a context is skipped during exception handling. For example, when an exception does a proceed, restart, or return, the flow of control jumps into lower contexts on the procedure's stack, and any higher contexts are removed from the stack without ever returning to them. This can be a problem: The contexts that were skipped might have performed some clean-up actions, such as closing files or releasing semaphores, if they'd been allowed to finish execution and return normally. Skipping these contexts during exception handling means skipping important clean-up jobs. The solution to this problem is to define a special method, whose purpose is to ensure clean-up blocks will be executed, even in the presence of exception handling. The name of this method in Smalltalk-80 is valueOnUnwindDo:. Assuming aCollection is defined, evaluating

```
[aCollection checkFifthElement]
    valueOnUnwindDo: [Transcript show: 'Time to clean up!']
```

will cause the first block, [aCollection checkFifthElement], to be evaluated. If aCollection has five or more elements, the value of the fifth element will be returned, and nothing more needs to be done. However, if aCollection has four or fewer elements, and if the exception handler for OutOfBoundsError causes control to return past the context of the valueOnUnwindDo: method (in effect skipping it), the second block will be evaluated, allowing any clean-up or finalization to be done. In Smalltalk-80, unwind blocks are even executed if they're skipped by a normal method return, because up-arrow is treated just like a return from an exception. In V 286, though, the meaning of up-arrow is hardwired into the virtual machine, so we can't duplicate this behavior.

THE MACHINE-INDEPENDENT IMPLEMENTATION

Although an implementation of exception handling inevitably delves into system-specific code, much of our solution is system independent. In fact, the same implementation of class Signal is used for Tektronix and Digitalk platforms (and potentially for ParcPlace), and most of class Exception is common as well. This section considers the system-independent aspects of the exception-handling package.

To begin with, there are a number of predefined signals, all of which are defined in the Signal class>>initialize method and accessible using messages to Signal. These basic signals include ones for unhandled exceptions and keyboard interrupts. In addition to these, a class variable called ErrorSignal is added to Object (just be careful how you add it!) and is accessible by using Object>>errorSignal.

To create a new signal, you send the message newSignal to an existing signal. So, for example, we could create the signal OutOfBoundsError by evaluating

```
OutOfBoundsError := ErrorSignal newSignal
```

either in a workspace or (more likely) in a class initialization method. The newSignal method creates the new object and sets its parent instance variable to the receiver. The parent variable in class Signal is used to provide more structure in signal handling. When a signal is raised, it can be handled by an exception handler for the signal, by one for the signal's parent or by one for any of the signal's ancestors. In this way, a programmer can define some general response for a tree of signals by registering a handler for the signal at the root. This response can then be specialized by registering more specific handlers for the signals further down in the tree.

Once a signal has been defined, sending it handle:do: registers an exception handler for it. The code for handle:do: is

```
handle: handlerBlock do: doBlock
    "Evaluate doBlock. If all goes well, return its value. If an exception
    occurs then the returned value could be generated by evaluating
    returnBlock."
    | returnBlock |
    returnBlock := [ :value | ^value].
    ^doBlock value
```

This method's most significant role is as a placeholder. Its basic function is simply to evaluate its second parameter, the do: block. But it also marks a place on the context stack so the system can find an appropriate handler when an exception occurs. How this happens will be explained next month when we consider Exception>>fetchHandlerBlock:. The block stored in the returnBlock temporary variable is used to make implementing Exception>>return easier.

The only other method we mentioned for class Signal was raise. As we said before, there are actually many variations of the raise message, depending on whether the exception handler can proceed through the exception, whether there's a parameter or error string needed, and so on. All these raise combinations call the same private method, which is Signal>>raiseWith:startingAt:extraString:proceed:. This is implemented as follows:

```
raiseWith: parameter startingAt: context
extraString: str proceed: aBoolean
    "Create a new exception and have it look for handlers
    starting at context."
    | exception |
    exception := self newException
        signal: self
        parameter: parameter
        extraString: str
        proceedBlock:
            (aBoolean
                ifTrue: [[:value | ^value]]
                ifFalse: [nil]).
    ^exception propagateFrom: context
```

This method creates a new instance of Exception, passing the signal as one of the parameters in the creation message. In addition, if aBoolean is true, the signal is "proceedable", which means that the handler is allowed to send the exception the proceed message, in effect declaring the error completely resolved and causing a return from the raise message send. If it is proceedable, the new exception will be passed the block [:value | ^value]. Like returnBlock in the handle:do: method, the block here simplifies our implementation, in this case making Exception>>proceedDoing: much simpler. Finally, this new exception is sent the message propagateFrom: with the context passed in as a parameter. This begins the process of finding a handler for the exception.

Exceptions have five instance variables: signal, parameter, extraString, proceedBlock, and handlerContext. The first four are set by the signal:parameter:extraString:proceedBlock: message, which is sent by a signal when the exception is created. The value of proceedBlock, if it isn't nil, is the [:value | ^value] block we saw above. After creating a new exception, a signal sends the propagateFrom: message, which in turn calls the propagatePrivateFrom: method. In addition to error handling, propagatePrivateFrom: sends the message fetchHandlerBlock: to find the right handler for the exception (in the process, it sets the instance variable handlerContext to the appropriate handle:do: message's context) and evaluates that handler. The implementation of fetchHandlerBlock: is described in next

continued on page 14...

Greg Hendley and Eric Smith

Significant supported events in Smalltalk/V PM as illuminated by Window Builder

If you have used Window Builder by Cooper & Peters, then you have taken advantage of its fill-in-the-blank way of writing when:perform: statements for the open method. You have probably noticed there are more events than you thought you needed. You may have even asked yourself, "Should I be using these events and, if so, how?"

In this installment of GUI Smalltalk, we will discuss some of the significant supported events for the subpanes and controls directly supported by Window Builder. This is not intended to be an exhaustive discussion of every event; it will, however, get the adventurous off to a good start.

We decided classes that implement supportedEvents would be the most interesting to look at. The remaining classes should inherit their superclasses' behavior. We will discuss each class in turn, including some significant supported classes.

TopPane

Nearly all windows involve some kind of TopPane, which is usually the window containing all the other controls. TopPanes support a number of events that no other kind of window is interested in.

- **#validated.** This event is generated as the final act in opening a TopPane. When this occurs, the pane represents a valid Presentation Manager (PM) window. This event seldom requires a handler. However, in some rare instances, it provides an opportunity to do any necessary twiddling of the PM frame window after it has been opened but before any of the children have been opened.
- **#activate.** When a frame window becomes the 'active' window (i.e., it is selected, given the active window border color, and the input focus), the window message WM_ACTIVATE is sent along to the PM frame window. In Smalltalk, this results in the #activate event. A newly opened window usually becomes the active window, so this happens when the window is opened as well as each time the frame window is activated.
- **#menuBuilt.** The #menuBuilt message is generated after the menu bar has been created but before children are opened or the TopPane validated. If you are using WindowBuilder, this event is unlikely to occur. Cooper & Peters have circumvented the normal menu bar creation methods in their open methods. Ordinarily, this event might be used to initialize the enable/disable state of the various menu choices, add custom menus, etc. When using WindowBuilder, these sorts of activities can be performed in the #initWindow method.
- **#close.** Whenever TopPane, not ViewManager, receives the message #close, it will generate the event #close before taking any action. If there is no handling method, or the handling method returns nil, then the close operation will proceed normally. Otherwise, no panes will be closed. Handlers for this event are quite common, particularly if dependents are used. This provides the ideal place to clean up dependents, PM resources, and other potential garbage as the window disappears.
- **#help.** The #help event occurs when help is called for via the F1 key. Using the help menu (should one be available) will not generate this event. The handling method may do whatever it pleases by way of providing help (e.g., toss up a dialog, open another application, put up a message box). If there is no handling method, or the handler returns nil, then the problem will be passed along to the PM help manager. Note that if you have a HelpManager defined for a window as well as a handling method for the help event, then unless the handling method returns nil, the PM help manager will not come up when F1 is pressed.
- **#timer.** This event will be generated whenever the frame window receives the window message WM_TIMER. This only occurs in special circumstances beyond the scope of this column.
- **#opened.** This event is a red herring. It won't hurt a TopPane to have a handler for this event, but that handler will never be activated because TopPanes don't generate this event.

DialogTopPane

DialogTopPanes behave just like TopPanes in most respects (including those having to do with generating events). All the events described for TopPane above are inherited, except that those having to do with the menu bar will not be given a chance to occur. One additional event is generated by dialogs:

- **#opened.** After a dialog is built, but before processing begins, the event #opened occurs. This provides the owning ViewManager with the opportunity to fill in entry fields, initialize button choices, etc.

SubPane

SubPane is included even though it is an abstract class. Many normal behaviors are described in this class. We will take advantage of inheritance in our descriptions and only deviations and additions will be described for subclasses.

- **#display.** While SubPane supports this event, it is only received by GraphPane. So for all other subclasses, unless you write a method that sends event: #display, you can disregard this event.
- **#resize.** This is sent after PM has resized a top pane (or other subclasses of ApplicationWindow). Most applications have no need for this event. Possible exceptions are special uses of GraphPane and GroupPane. Most resizing is handled with the normal get contents and display methods. This is supposedly one of the advantages of using an existing windowing system such as PM.
- **#getPopupMenu.** This event normally occurs as a result of the mouse button2 click. No surprise here.
- **#getMenu.** This event is usually not sent if the window was built using Window Builder. The exception (there's always an exception) is when the pane looks for its pop-up menu. If it can't find one, it looks for its regular menu to use for a popup. Therefore, it is your choice to use this or the previous event for your pop-up menus. Proper discussion of menus would require its own column.
- **#getContents.** Now we are back on familiar ground. This event is sent whenever a subpane is opened. It is used to set the text of a text pane, list of a list pane or combo box, and label or text for other controls. This setting is normally done using the method contents:. It is also sent as part of the restore and update methods for many classes.
- **#help.** This is normally sent when the F1 key is pressed. Not all subclasses receive this event.

TextEdit

- **#textChanged.** This event is sent each time a character key, backspace, or delete is pressed. Think about whether you want to respond. This event will be sent frequently if entire paragraphs are being typed.
- **#horizScroll.** You normally will not care about this event, which is sent when you scroll using the horizontal scroll bar. It also happens with automatic scrolling, which occurs when you type past the pane and word wrap is off.
- **#vertScroll.** This is similar to horizScroll.
- **#help, #getPopupMenu**, and **#getMenu**. None of these are received.

TextPane

TextPane inherits events from TextEdit. It also adds one event:

- **#save.** This is sent through selecting the "save" item in the pop-up menu for TextPane.

ListBox

- **#charInput.** Most Smalltalkers do not use this event; they use the event #select, which happens when a character is typed. If a character is the first character of one of the items, that item is selected.
- **#drawItem** and **#highlightItem.** Seldom used by most Smalltalkers, these are sent only when a user-drawn item is included in the list of items. This deserves its own column and will not be discussed here.
- **#select.** This event occurs when an unselected item is selected, not when a selected item is re-selected. It also occurs when an item is selected by typing its first character.
- **#doubleClickSelect.** This event happens whenever an item is double clicked. Behavior is the same whether or not the item was already selected.

ListPane

Although neither super- nor subclass of ListBox, ListPane behaves similarly. The exception is as follows:

- **#select.** This event occurs when selecting an item that is already selected.

ENTRYFIELD

Entryfield is the Smalltalk class representing one-line entry areas commonly seen littered about dialogs, although they may be used in any window. Most of Entryfield's interesting behavior can be used by paying attention to only two events:

- **#getContents.** As with most other panes, this event is generated by an Entryfield when it first comes up. It provides a nice opportunity to initialize the text contained in the entry field before the user gets to it. This is done in the handling method by sending #contents: to the pane with an appropriate String as an argument.
- **#textChanged.** Any time the contents of an Entryfield are changed, the #textChanged event is generated. It doesn't matter how the change originated; whether the user typed in more characters or somebody sent #contents: to the Entryfield, a #textChanged event is generated. This means that setting the contents of an Entryfield in the handler for a #textChanged generated by that Entryfield will lead to infinite recursion.

ComboBox

- **#textChanged.** Be careful about using this event as a trigger for other activities. We recommend you save the new text somewhere or note that the text is changed. One thing you do *not* want to do is update. This will create a circularity. The event #textChanged is sent in response to several activities: once when contents is set and twice when you type the first letter of one of its list elements. It is *not* sent when you type any other character. It *is* sent when

you press the pull-down button and when you select an item from the list.

- **#charInput.** This happens whenever *any* character is typed. Notice the difference between this and the previous event. A character can be typed without being entered into the text part of the combo box.
- **#select.** This event occurs at peculiar times the way #textChanged does. It is sent twice when text is in the entry field part and the list is pulled down. It is sent once when no text is in the entry field part and the list is pulled down. It is *not* sent when an item is selected that matches the text in the entry field part. It *is* sent once when an item is selected that does not match the text in the entry field part.
- **#doubleClickSelect.** This event does not happen for the ComboBox.
- **#drawItem.** This event occurs when a user-drawn item needs to be drawn. Most Smalltalkers will not use this event.
- **#highlightItem.** This event occurs when a user-drawn item needs to be highlighted. Most Smalltalkers will not use this event.
- **#listVisible.** This happens when you press the pull-down button. Most Smalltalkers will not use this event.

BUTTON

Button is the superclass of several kinds of controls that get clicked. Nearly all of them generate events, which are expected to be handled in similar ways.

- **#getContents.** This occurs when the pane first comes up. It can be used as an opportunity to set the contents of the button. For most kinds of Button, the #contents: message expects a String as an argument. This String will become the label for the button.
- **#clicked.** Any time a Button is pressed, the #clicked event occurs. For instances of Button, all you need to know is that the Button was pressed. For toggle-type buttons, the action of your handler may depend on whether the button was clicked on or off. This can be determined by sending the message #selection to the button. The Boolean returned will reflect the state of the button.

DrawnButton

The class DrawnButton represents a fairly special subclass of Button. It isn't like the others in that it has no predefined look. Instead, the owning window (or, in our case, the ViewManager) is expected to draw whatever it wants on the button's graphics context.

- **#getContents.** This event occurs when the pane first comes up. It may be used as an opportunity to provide the pane with a Bitmap, which it will draw on itself. DrawnButtons expect a Bitmap as an argument for the #contents: message.
- **#drawItem.** Any time a DrawnButton pane that does not have a Bitmap is asked to display, it will generate this event. When the handling method gets control, the DrawnButton pane will have a valid graphics tool. The handler method may then ask for its pen and draw whatever it wants on it. Note that this event also occurs as a result of the button being clicked.
- **#highlightItem.** This message is generated as a result of pressing a DrawnButton. The underlying PM window messages inform as to whether highlighting is to be added or removed. Alas, by the time we reach the event level, this information has been lost. As with #drawItem, the graphics tool of the DrawnButton in question is valid while this event is processed.

SpinButton

Admittedly, this class is not directly supported by Window Builder. It is included in the standard image and can be added to Window Builder as a custom pane.

- **#getMenu, #getPopupMenu,** and **#help.** None of these are received.
- **#textChanged.** This is an unusual event in the number of times it occurs for a given action. It is sent once for each character typed. It is normally sent once when the up or down button is pressed. When there is text in the entry field that does not match any of its enumerated values, and the up or down button is pressed, the event happens twice. It happens once when the backspace key is pressed and twice when the delete key is pressed.
- **#up.** This event is sent when the up button is pressed. Normally, you would only look at the #textChanged event.
- **#down.** This event is sent when the down button is pressed. Normally, you would only look at the #textChanged event.
- **#getContents.** This event is ignored if the spin button is numeric. When the spin button is non-numeric, it expects to be told its list of enumerated values.

ScrollBar

Scrolling, with or without the scroll bar control, deserves more space than we can give here. We can, however, point out a few features.

The following events occur as a result of pressing the arrows, clicking in the blank areas, or moving the tab: #nextPage, #prevPage, #nextLine, #prevLine, #sliderPosition, #sliderTrack, and #endScroll.

The following events do not occur: #getMenu, #getPopupMenu, and #help.

#getContents occurs in the same manner as for most sub panes, but scroll bars do not know the method contents:. Instead, they use position:. ■

Greg Hendley is a member of the technical staff at Knowledge Systems Corporation. His OOP experience is in Smalltalk/V(DOS), Smalltalk-80 2.5, Objectworks Smalltalk Release 4, and Smalltalk/VPM. Eric Smith is also a member of the technical staff at Knowledge Systems Corporation. His specialty is custom graphical user interfaces using Smalltalk (various dialects) and C. The authors may be contacted at Knowledge Systems Corporation, 114 MacKenan Drive, Suite 100, Cary, NC 27511, or by phone, 919.481.4000.

Juanita Ewing

How to manage source without tools

Many Smalltalk programmers develop significant applications without any source-management tools. Although it takes a certain amount of discipline, small- to medium-sized applications can be developed without additional tools. This column will describe several sound practices for the successful management of application source.

The code in this column is for versions of Smalltalk/V under Windows and OS/2. The ideas are applicable to other versions of Smalltalk/V and to Objectworks\Smalltalk.

CONCEPTS

One concept is critical for successful management of application source:

- Never view your image as a permanent entity.

And there are two corollaries:

- Don't depend on your image as the only form of your application.
- Store your application in source form and rebuild your image frequently.

Viewing the image as a non-permanent entity doesn't necessarily imply that vendors are selling unreliable software. There are several ways an image can become non-functional, other than a serious Smalltalk bug or disk crash.

An image can become unusable because of some simple mistake on the part of a developer, such as accidentally removing a class that is relevant to the application under development. If the image is the only form of an application, recovering sources for an application class can be difficult and tedious. Another common mistake is the accidental deletion of the change log or changes file. The source for all the changes you've made to an image is stored in this file.

Not all motivation for storing an application outside an image derives from mistakes. When your vendor releases a new version, migration to the new version may be necessary to take advantage of new features or continue to the highest level of technical support.

PRACTICE

What is your application? In Smalltalk, this is not always a straightforward answer. Images contain large class libraries, and applications are developed by adding to and modifying class libraries. There is no clear distinction between system and application code. Because of this, it is very difficult to extract all parts of an application from an image, especially after the development is completed. It is better to extract or list the parts of your application as you develop it. Then short-term memory can help you decide if the modification you made was necessary for your application or if a temporary modification was needed for debugging. One of the most common errors is to omit a critical piece of one's application.

I will discuss two techniques for extracting your application code as you develop it. The first technique uses the browser to file out code right after it is developed. Most application code will be located in new classes, which can be filed out as a unit. Other application components are extensions to system classes, which can be filed out at the method level. The result of this technique is many small files.

There are dependencies among the classes defined in these files. For example, a subclass depends on its superclass. I use a script to reassemble all these files in correct order, rather than try to remember what the dependencies are. It is possible to create the script for reassembly at the same time the parts of an application are filed out.

Figure 1 contains a script for installing multiple files. The script consists of a list of file names, which is enumerated to install each file into the image.

```
"Read and file-in application files."
#(
'ExtendedListPane.cls'
'AviationGraphPane.cls'
'JetEngine.cls'
```

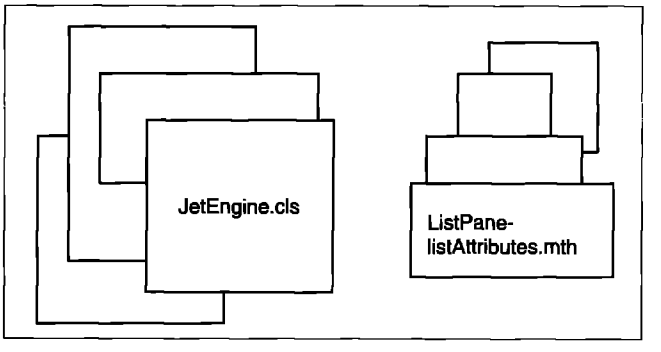


Figure 1. Example of reconstructing an application using multiple files.

```
'PropEngine.cls'
'RudderMechanics.cls'
'ListPane-class-supportedEvents.mth'
'ListPane-listAttributes.mth'
'ListPane-listAttributes:.mth'
'GraphicsMedium-bezierCurve:.mth'
)
    do:
        [:fileName |
        (Disk file: fileName) fileIn]
```

Another technique is to make a list of all relevant application pieces as they are developed. The list can be maintained in order of reassembly and used to extract all components of an application on demand. The result of extraction is a single file. Reconstruction of the application is a simple matter of installing one file. The source can be partitioned into several files, if necessary.

In Listing 1, the script has three lists: one for classes, one for instance methods, and one for class methods. The classes listed in the first script are written to the stream, then the methods in the second list are written to the stream. The file-out code makes use of ClassReader, which knows about Smalltalk source-file format.

This script makes use of a new method, fileOutClassOn:, defined in Listing 2. The new method, which writes a class definition and its methods on a stream, takes an instance of fileStream as an argument. It is similar to an existing method, fileOut:, which takes a file name as an argument, creates the file, then writes a class and its methods to the file:

The script in Listing 1 works in the simplest cases, in which there are no forward references to classes. For example, if code in the class JetEngine refers to the class PropEngine, the filein will not proceed properly. This problem can be avoided by defining all classes before any methods, as in the script in Listing 3. This script also has two lists, but the first list is enumerated over twice. A supporting method is defined in Listing 4.

INITIALIZATION

Applications consist of more than classes and methods. Instances of windows, panes, and domain-specific classes are also part of an application. Application reconstruction, therefore, must consist of more than filing in class and methods. The expressions executed in a workspace or inspector to set up the state of your application, such as initializing classes and creating new objects, need to be re-executed when your application is reconstructed. Save these expressions by collecting them in a file and executing them after reconstructing your application. In a future column I will discuss these types of expressions, and ways to execute them as part of a script.

ERRORS

The most error-prone portion of these techniques is recording pieces of the application as it is developed. Source-management tools are quite valuable because they record this information automatically. Because pieces of the application are recorded by hand, it is also common practice to search back through the change log to make sure no pieces have been for-

Listing 1. Example of creating a single file for application reconstruction.

```
| sourceStream reader |
"Create filestream for storing sources."
sourceStream := Disk file: 'AviationSource.st'.
"Write application classes."
#(
ExtendedListPane
AviationGraphPane
JetEngine
PropEngine
RudderMechanics)
    do:
        [:className |
        reader :=ClassReader forClass: (Smalltalk at: className).
        reader fileOutClassOn: sourceStream].

"Write standalone instance methods"
#(
(ListPane listAttributes)
(ListPane listAttributes:)
(GraphicsMedium bezierCurve:)
)
    do:
        [:classNameAndSelector |
        reader :=ClassReader forClass: (Smalltalk at:
            (classNameAndSelector at: 1)).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on:sourceStream].

"Write standalone class methods"
#(
(ListPane supportedEvents)
)
    do:
        [:classNameAndSelector |
        reader :=ClassReader forClass: (Smalltalk at:
            (classNameAndSelector at: 1)class).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on: sourceStream].
sourceStream close.
```

Listing 2. Supporting code in ClassReader for filing out a class onto a stream.

```
ClassReader
instance method

fileOutClassOn: aFileStream
    "Write the source for the class (including the class definition,
    instance methods, and class methods) in chunk file format
    to afileStream."
    class isNil ifTrue: [^self].
    CursorManager execute change.
    afileStream lineDelimiter: Cr.
    class fileOutOn: afileStream.
    afileStream nextChunkPut: String new.
    (ClassReader forClass: class class) fileOutOn: afileStream.
    self fileOutOn: afileStream.
    CursorManager normal change
```

Listing 3. Example of creating a single filefor application reconstruction.

```
| sourceStream classListreader |
"Create file stream for storing sources."
sourceStream := Disk file:'AviationSource.st'.
```

continued on next page

Listing 3 continued

```
"Classes in the application"
classList := #(
ExtendedListPane
AviationGraphPane
JetEngine
PropEngine
RudderMechanics).

"Write application class definitions."
classList
    do:
        [:className |
        reader :=ClassReader forClass: (Smalltalk at: className).
        readerfileOutClassDefinitionOn: sourceStream].

"Write the methods for the application class"
classList
    do:
        [:className |
        reader :=ClassReader forClass: (Smalltalk at: className).
        reader fileOutOn: sourceStream].

"Write standalone instance methods"
#(
(ListPane listAttributes)
(ListPane listAttributes:)
(GraphicsMedium bezierCurve:)
)
    do:
        [:classNameAndSelector |
        reader :=ClassReader forClass: (Smalltalk at:
            (classNameAndSelector at: 1)).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on:sourceStream].

"Write standalone class methods"
#(
(ListPane supportedEvents)
)
    do:
        [:classNameAndSelector |
        reader :=ClassReader forClass: (Smalltalk at:
            (classNameAndSelector at: 1)class).
        reader
            fileOutMethod: (classNameAndSelector at: 2)
            on: sourceStream].

sourceStream close.
```

Listing 4. Supporting code in ClassReader for filing out a class definition without methods.

```
fileOutClassDefinitionOn:aFileStream
    "Write the source for the class (but not for the instance
    methods and class methods) in chunk file format
    to aFileStream."
    class isNil ifTrue: [^self].
    CursorManager execute change.
    aFileStream lineDelimiter: Cr.
    class FileOutOn: aFileStream.
    aFileStream nextChunkPut: String new.
    CursorManager normal change
```


gotten. This activity is usually performed in a regular fashion, such as before each snapshot.

Another common error is to rebuild an application on top of an image that has been used for development. This is not a good idea because the state of the image is unknown. There may be unwanted side effects from objects in the image. It is imperative, therefore, that the application is reconstructed from a clean, pristine image.

FREQUENCY

How often should the application be rebuilt? Early in development, when many classes are being created, the scripts are modified rapidly. It valuable to rebuild often to test the scripts; if they're too far out of sync with the application source, it can be difficult to debug the reconstruction process. In the middle stages of development the scripts are not in so much flux and the application doesn't need to be rebuilt so often to test them out. Other considerations may force application reconstruction, such as redesign of parts of an application. As the product is nearing completion, the development team may want to reconstruct the application often to confirm that the build process is bug-free. ■

Juanita Ewing is a senior staff member of Digitalk Professional Services. She has been a project leader for several commercial O-O software projects, and is an expert in the design and implementation of O-O applications, frameworks, and systems. In a previous position at Tektronix Inc., she was responsible for the development of class libraries for the first commercial-quality Smalltalk-80 system.

THE BEST OF comp.lang.smalltalk

Alan Knight

Smalltalk performance

Many people think of Smalltalk as slow. Unfortunately, they're right, especially as compared with the reference point of optimized C. This column will explore why Smalltalk code runs so slowly, just how slow it is, and the possibility for improvement.

WHY IS SMALLTALK SLOW?

Although surprisingly fast for what it does, Smalltalk is slow for various reasons. Conventional wisdom blames garbage collection. After all, Smalltalk collects garbage while those other, fast languages don't. Garbage collection does have a price, but not nearly as high as people think. More time-consuming is safety checking. Smalltalk checks all array references to make sure they are in bounds, every object reference for null values, every integer operation for overflow, and so on. C does none of these things.

If you have a compiler like Turbo Pascal, which allows you to turn array-bounds checking on and off, try doing it with a program that uses arrays. The effect on performance is very noticeable. I still leave checking on by default, and always turn it on when I'm trying to debug. When I learned C I wasted a lot of time trying to figure out how to turn on bounds checking, but I finally did. It involves paying a lot for an interpreter so my code can run more slowly than equivalent Smalltalk, but it's worth it.

Of course, these approaches have the advantage that you only pay the price during development. Safety features can be turned off when shipping the "bug-free" production code. It would be an interesting experiment for a vendor to provide a fast, unsafe version of the Smalltalk virtual machine for standalone applications.

Another important factor is message passing, for two reasons. First, message sends are a little pricier than function calls. You have to additionally figure out which function to call at runtime. However, the high cost of message sends is due to their number. Everything in Smalltalk except instance-variable access requires a message send. Even if messages cost less than function calls, the fact that there are so many more in the average Smalltalk program than the average C program makes Smalltalk slower.

HOW SLOW IS IT?

Quantitative performance measurements are always difficult. Results vary greatly between applications and minor changes can make a big performance difference.

Given this difficulty, we are fortunate to have someone with a good knowledge of the subject, at least with respect to ParcPlace Smalltalk. This impressive disclaimer is from Peter Deutsch (deutsch@smli.eng.sun.com)

> **I was the principal designer and implementor of ParcPlace's Smalltalk code generators, including the portability architecture, the code generation framework, the stack management architecture, and the individual generators for 680x0, 80386, SPARC, MIPS, and RS/6000. The opinions expressed below are my own and should not be attributed to ParcPlace or to Sun.**

He then writes:

> **In my experience, based on a variety of both micro- and macro-experiments, the ParcPlace Smalltalk system does benchmark around a factor of 8 slower than optimized C for integer, structure, and array computation that does not contain large numbers of procedure-call-free loops. For straight-line integer computation, the ratio can get down as low as 4 or 5 to 1. (Of course, ParcPlace Smalltalk does overflow checking on all arithmetic operations, so any such comparison is not entirely appropriate.) For highly optimizable loops, especially ones involving access to arrays or strings (which ParcPlace Smalltalk always bounds-checks, and C never does), the ratio can get up as high as 40 or 50 to 1 under the most unfavourable circumstances, such as the 1-statement loops of strlen or strcpy.**
>
> **It is because of these things that ParcPlace recommends that, when necessary, users write their high-usage loops in C. Smalltalk's advantages are in areas other than highest performance for unchecked inner loops.**

IS THIS FAST ENOUGH?

For many applications, this kind of speed is high enough. The numerous advantages of Smalltalk are worth the performance hit in these areas. For other application areas, the speed is definitely unacceptable, but this is partly psychological. If Smalltalk is running as fast as it reasonably can, we must either accept the performance or use another language. If, on the other hand, it runs slowly because the implementors haven't bothered to make it go faster, then we may get annoyed about it.

A strong voice for the possibility of improving performance comes from the implementors of Self. Self is a prototype-based language that is even more difficult to optimize than Smalltalk,

but its implementation achieves much better performance. This is done using an extremely aggressive optimizing compiler. For example, Self exploits range information in integer computations. Using this information, it can omit overflow checks in cases where they're shown to be unnecessary.

Bruce Samuelson (bruce@ling.utafl.edu) doesn't think current Smalltalk performance is fast enough. He writes:

> **ParcPlace, your dynamic compilation technology, is indeed impressive. . . . But you can do better, and you have chosen not to because you don't think it is high priority:**
> **1) The Self authors claim in the literature that Smalltalk could be sped up by about a factor of 5. They claim in person that PPS is not interested in doing so (at least as of OOPSLA '91).**
> **2) Mike Khaw's recent posting showed that Smalltalk did integer arithmetic in a tight loop about 1/8 the speed of C. . . . This is in the ballpark of what one would expect for such low level comparisons.**
> **3) A Smalltalk "VM implementor" told me at OOPSLA '91 that the machine code generated by the dynamic translator is of "plain vanilla," unoptimized quality. For example, he thought the code for SPARC machines (he was not the SPARC VM implementor) did not make use of register windows, SPARC's idiomatic technique for passing function arguments efficiently. Perhaps he was wrong, or perhaps I misunderstood him, but times past when I've posted this and asked for comments from PPS, you have remained silent. It seems like this is one area in which you could apply some fairly standard optimization techniques in your VM that wouldn't require modifications to the compiler in the VI.**
> **4) A PPS employee was engaged in a serious optimization project before he left PPS. I have not heard from PPS on the status of this project, except a comment I would paraphrase as follows: "We are impressed with the speed of forthcoming new machines [based, I suppose, on DEC Alpha, HP-PA, Intel 586, TI Viking, etc.] and feel that hardware vendors will solve possible Smalltalk performance problems."**
> **5) Critique of (4): Yes, Smalltalk grows faster in proportion to the hardware. But so does every other language, and Smalltalk remains 5–10 times slower than C. The hardware vendors are not improving the competitive position of Smalltalk, except to make it feasible to use at all, and they already did that a few years ago. As machines get faster, applications get more ambitious and demand more cpu cycles. . . . A software vendor offering a development environment should regard decent optimization as a priority. Reviews of software products, whether of languages or applications, usually give performance a prominent place. You will make us, your customers, look better if you give us the tools to write blazing applications.**
> **6) I have had to spend more time on optimizing my Smalltalk code than I would have liked, which has taken time away from more productive activities. I imagine this has happened to other programmers.**
> **7) A turbocharged Smalltalk that could even modestly compete with C and C++ in speed would be an absolute dynamite product. How many of the postings to comp.lang.c++ give efficiency as a reason for using this "engineering compromise"? Take away efficiency as a criticism of Smalltalk and a lot of programmers and managers will take note.**
> **8) Digitalk must have had some money to spend to able to buy out Instantiations. What if they put some of their money into doing a bang-up job at optimizing ST/V? Where would that leave ParcPlace?**
> **9) Despite all these comments, which are directed to PPS in response to Tim Rowledge's posting, I realize that PPS is a small company with finite resources. Your founders have profoundly influenced the entire computer industry (GUIs, object orientedness) for the better. And you sell a very nice Smalltalk environment indeed. So I will counsel myself to remain patient and trust your marketing instincts. But please don't keep performance on the back burner forever...**

REGISTER WINDOWS

There are quite a few complaints here, and I entirely agree with the main thrust that ParcPlace needs to place more emphasis on performance. I'd like to specifically deal with one of the claims that attracted particular attention on the net: the assertion that ParcPlace Smalltalk does not use register windows on the SPARC. For those of you even more blissfully ignorant of hardware than myself, I will attempt to explain register windows.

Machine registers are very fast to access and CPU designers like to have lots of them. The downside of this (apart from having to use valuable chip space) is that when there are many registers, more bits in the instruction word are needed to specify which one you want.

There are various ways of getting around this. One is to have more than one set of registers, used for different purposes (e.g., integer and floating point). The SPARC designers provided lots of registers, but made only a few of them visible at a time. By changing the register "window," you change which registers are visible.

Changing the window normally is done when making a procedure call. Rather than put arguments onto the stack, which is in main memory and therefore slow, one can put them into registers, then change the register window. Since the windows have some overlap, values put into the bottom of the register window of the calling routine will appear in the top of the window of the called routine. The arguments are immediately available and the called routine has its own set of registers to play with.

This technique can speed up procedure calls quite a bit. SUN claimed in some document I once read that register windows were aimed specifically at incrementally compiled languages like LISP and Smalltalk. In these languages, the compiler doesn't have as much time to think about how to optimize code and there are many procedure calls. Register windows are supposed to allow these calls to be easily optimized.

If SPARC can't or doesn't exploit SPARC register windows, it sounds like there's a serious communication problem between chip and language designers.

This claim provoked discussion about how easily register windows could be used—whether they would interfere with garbage collection (since values in registers outside the current window would not be easily visible) and other such topics.

Urs Hoelzle (urs@xenon.stanford.edu) mentioned that Self has been using SPARC register windows with garbage collection for some time. Peter Deutsch provided a comprehensive analysis of reasons for Smalltalk not to use them:

> **The problem of pointers buried in register windows is indeed a significant one, but it is not the reason why I would recommend against modifying the Objectworks/Smalltalk (Ow/ST) implementation to use register windows. First, the performance gains would not be dramatic. Ow/ST spends a substantial fraction of its time in support code written in C, which would not be affected. A substantial fraction of the time in compiled Smalltalk code is spent doing message sends, type checks, etc., which would also not be affected. Also, since Smalltalk stacks get very deep and fluctuate more deeply than C stacks, the 7- or 8-register window on current SPARCs would over- and underflow significantly often. My best guess was that we would not see more than 20–25% performance improvement. (On future SPARC processors, where both the cost of memory references relative to register accesses and the number of register windows might be larger, this improvement might be somewhat greater.) Second, one of the keys to Ow/ST's remarkable portability is that it uses a very similar internal storage format for stack frames on all platforms. However, because saving and restoring register frames is done on the SPARC by code that is not accessible to ParcPlace, we cannot affect the storage format for these frames. So in order to use the SPARC register frames, we would have to either provide a complete second set of, or add radical new flexibility to, the large body of code in the runtime support system that manipulates stacks. The bottom line is that, in my opinion, the work required to fit Ow/ST to the SPARC's frame model would not justify the relatively small performance improvement. As for the comparison against Self, the Self authors acknowledge that the factor of 5 is only achievable under some circumstances. I do think it would be exciting to apply the Self compilation ideas to Smalltalk, and doing this could well produce dramatic performance improvements (on all platforms), but this would require wholesale redesign of most of the platform-independent code (other than the memory manager) in the Ow/ST runtime support system. The optimizing compilation experiments I did at ParcPlace were based on an alternative approach that would not have required such substantial changes to the Ow/ST virtual machine, but might have required type declarations (or at least type hints) provided by the user (or a type inference system).** ■

Alan Knight is a researcher in the Department of Mechanical and Aerospace Engineering at Carleton University, Ottawa, Canada, K2C 3P3. He can be reached at +1 613.788.2600 x5783, or by e-mail at knight@mrco.carleton.ca.

...continued from page 5

month's system-dependent section because it depends on the layout of contexts.

Once the handler block is found, it's evaluated with the exception as a parameter. This allows the handler block to send the proceed, reject, restart, and return messages to the exception, and to query the exception for information about the error. Below are the implementations for proceed and reject—those for return and restart are in next month's article because they depend on some specifics of the V 286 system.

Proceeding is simple: Since we have the instance variable proceedBlock, all we need to do is evaluate it, perhaps with some meaningful parameter, as in

```
proceedDoing: aBlock
    "Return the value of aBlock as the value of the raise signal. Unwind
    the stack up to that point and resume execution in the context that
    raised the signal."
    | answer |
    answer := aBlock value.
    signalContext unwindLaterContexts.
    proceedBlock value: answer
```

Evaluating proceedBlock causes control to return into the context where the signal was first raised. The only subtle thing to remember concerns the unwind mechanism. Before evaluating proceedBlock, we call unwindLaterContexts, which evaluates the unwind blocks of every context we'll skip by proceeding.

Implementing reject is also quite simple. The current handler context (as found by fetchHandlerBlock:) is stored in the exception's handlerContext instance variable, so to find the next handler below the current one, we just need to look for some handler for the exception's signal below handlerContext. We can do that by sending propagatePrivateFrom: to the receiver exception with handlerContext as the parameter.

At this point we have a system-independent implementation for much of our package. The class Signal is complete and we need only three more methods for class Exception: return, restart, and the private method fetchHandlerBlock:. We also need to implement unwindLaterContexts to implement our unwind mechanism. Finally, we need some extra functionality for class Process. Next month, we will describe these final aspects of our system, such as the need to create a new set of context-related classes to make dealing with contexts in V 286 consistent and relatively trouble-free. ■

References

1. Van Orden, E. Application talk, HOOPSLA! 1(2), 1988.
2. Graver, J. Type-checking and type-inference for object-oriented programming languages. Doctoral thesis, University of Illinois at Urbana-Champaign, 1989.

Bob Hinkle and Ralph E. Johnson are affiliated with the University of Illinois at Urbana-Champaign. *Mr. Hinkle's work is supported by a fellowship from the Fannie and John Hertz Foundation.*

Kent Beck

Collections idioms: standard classes

Our previous column focused on enumeration methods and how to use all of them to advantage. This column covers the common collection classes, how they are implemented, when you should use them, and when you should be careful.

COLLECTION CLASSES

Array

Use an Array if you know the size of the collection when you create it, and if the indices into the elements (the first argument to at: and at:put:) are consecutive integers between one and the size of the array.

Arrays are implemented using the "indexable" part of objects. Recall that you can declare a class indexable. You can send new: anInteger to an indexable class and you will receive an instance with anInteger-indexable instance variables. The indexable variables are accessible through at: and at:put:. Array needs no more than the implementation of at: and at:put: in Object, and the implementation of new: in Class to operate.

Many people use OrderedCollections everywhere they need a collection. If you

- want a dynamically sized collection without the OrderedCollection overhead (see below)
- are willing to make the referencing object a little less flexible
- don't often add or remove items, compared with how often you access the collection

you can use arrays instead. Where you had:

```
initialize
    collection := OrderedCollection new
```

you have:

```
initialize
    collection := Array new "or even #()"
```

then you replace add: and remove: sent to collection with copyWith: and copyWithout: and reassign collection

```
foo
    collection add: #bar
```

becomes

```
foo
    collection := collection copyWith: #bar
```

The disadvantage of this approach is that the referencing object now has built into it the knowledge that its collection isn't resizable. Your object has, in effect, accepted some of the collection's responsibility.

ByteArray

ByteArrays store integers between 0 and 255 inclusive. If all the objects you need to store in an Array are in this range, you can save space by using a ByteArray. Whereas Arrays use 32-bit slots (i.e., soon-to-be-obsolete 32-bit processors) to store object references, ByteArrays only use 8 bits.

Besides the space savings, using ByteArrays can also make garbage collection faster. Byte-indexable objects (of which ByteArrays are one) are marked as not having any object references. The collector does not need to traverse them to determine which objects are still reachable.

As I mentioned in the last column, any class can be declared indexable. Instances are then allowed to have instance variables that are accessed by number (through at: and at:put:) rather than by name. Similarly, you can declare classes to be byte indexable. at: and at:put: for byte-indexable objects retrieve and store one-byte integers instead of arbitrary objects. A significant limitation of byte-indexable objects is that they can't have any named instance variables. This is to preserve the garbage-collector simplification mentioned above.

If you want to create an object that is bit-pattern oriented, but shouldn't respond to the whole range of collection messages, you should create a byte-indexable class. Such objects are particularly useful when passed to other languages because the bits used to encode the objects in a byte indexable object are the same as those used by, for instance, C, whereas a full-fledged SmallInteger has a different format than a C int.

Dictionary

Dictionaries are like dynamically sized arrays where the indices are not constrained to be consecutive integers. Dictionaries use hashing tables with linear probing to store and look up their elements (Figure 1). The key is sent "hash" and the answer modulo the basic size of the Dictionary is used to begin searching for the key. The elements are stored as Associations.

Dictionaries are rather schizophrenic. They can't decide whether they are arrays with arbitrary indices or unordered collections of associations with the accessing methods at: and at:put:. It doesn't help that Dictionary subclasses Set to inherit the implementation of hashed lookup. I treat them like arrays. If I want to

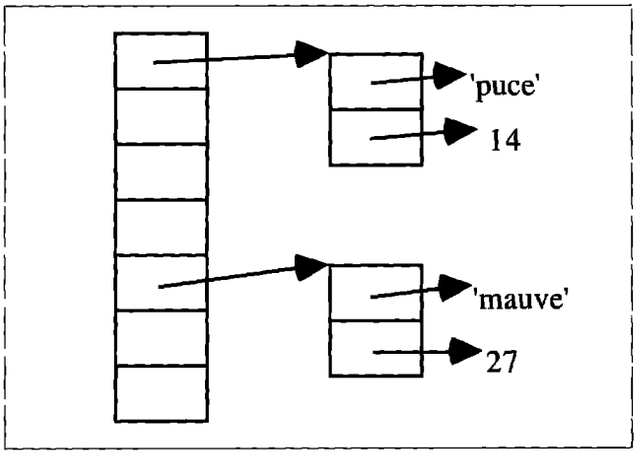


Figure 1. A typical Dictionary.

think of them as associations I use the message "associations" to get a set of associations I can operate on unambiguously.

When a Dictionary looks up a key it uses = to determine if it has found a match. Thus, two strings that are not the same object but contain the same characters are considered to be the same key. This is why when you reimplement =, you must also reimplement hash. If two objects are =, they must have the same hash value.

If you read your Knuth, you will see that hashed lookup takes constant time—it is not sensitive to the number of elements in the collection. This mathematical result is subject to two pragmatic concerns, however: hash quality and loading. When you send hash to the keys you should get a random distribution. If many objects return a number that is the same modulo the basic size of the Dictionary, then linear probing degenerates to linear lookup. If most of the slots in the Dictionary are full, the hash is almost sure to return an index that is already taken and, again, you are into linear lookup. By randomizing the distribution of hash values and making sure the Dictionary never gets more than 60% full, you will avoid most of the potential performance problems.

IdentityDictionary

IdentityDictionaries behave like Dictionaries except that they compare keys using == (are the two objects really the same object?). IdentityDictionaries are useful where you know that the keys are objects for which = is the same as == (e.g., Symbols, Characters, or SmallIntegers).

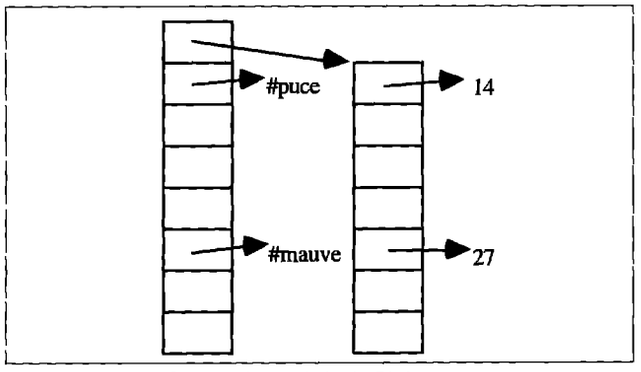


Figure 2. A typical Identity Dictionary.

Instead of being implemented as a hash table of associations, IdentityDictionaries are implemented as two parallel arrays. The first holds the keys, the second the values (Figure 2).

This implementation saves space because each association in a Dictionary takes 12 bytes of header + 8 bytes of object reference = 20 bytes. The total memory usage for a Dictionary is 12 bytes for the header of the Dictionary + 4 bytes times the basic size of the Dictionary + 20 bytes times the number of entries. The memory required for an IdentityDictionary is 24 bytes for the header of the object and the value collection + 8 bytes times the basic size.

For example, a 10,000-element Dictionary that has 5,000 entries free would take 12 + (4 * 15000) + (20 * 10000) = 260,012 bytes. You can see how the overhead of the Associations adds up. The same collection stored as an IdentityDictionary would take 24 + (8 * 15000) = 120,024 bytes.

OrderedCollection

OrderedCollections are like Arrays in that their keys are consecutive integers. Unlike Arrays, they are dynamically sized. They respond to add: and remove:. OrderedCollections preserve the order in which elements are added. You can also send them addFirst:, addLast:, removeFirst, and removeLast.

Using these methods, it is possible to implement stacks and queues trivially. There are no Stack or Queue objects in Smalltalk because it is so easy to get their functionality with an OrderedCollection. To get a stack you use addLast: for push, last for top, and removeLast for pop (you could also operate the stack off the front of the OrderedCollection). To implement a queue you use addFirst: for add and removeLast for remove.

As an example of using an OrderedCollection for a queue, let's look at implementing level-order traversal. Given a tree of objects, we want to process all the nodes at one level before we move on to the next:

```
Tree>>levelOrderDo: aBlock
    | queue |
    queue := OrderedCollection with: self.
    [queue isEmpty] whileFalse:
        [| node |
        node := queue removeFirst.
        aBlock value: node.
        queue addAllLast: node children]
```

OrderedCollections keep around extra storage at the beginning and end of their indexable parts to make it possible to add and remove elements without having to change size (Figure 3).

Because OrderedCollections are dynamically sized they preallocate a number of slots when they are created in preparation for objects being added. If you are using lots of OrderedCollections and most are smaller than the

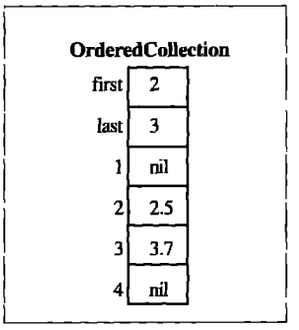


Figure 3. The result of (OrderedCollection new: 4) add: 2.5; add: 3.7.

initial allocation, the space overhead and its effect on the storage manager can be significant. I have heard stories of programs speeding up by a factor of 60 just by replacing OrderedCollection new with OrderedCollection new: 1 at the right spot. Gather statistics on the number and loading of your OrderedCollections to determine if this optimization will help you.

Another performance implication of using OrderedCollections is the level of indirection required to access elements. at: as defined in Object just invokes a primitive to index into the receiver's indexed instance variables. To implement at: and at:put:, OrderedCollections have to take first into account:

```
OrderedCollection>>at: anInteger
    anInteger > self size ifTrue: [self error: 'Out of bounds']. ^
    super at: anInteger + first - 1
```

RunArray

RunArrays have the same external protocol as OrderedCollection, but they are optimized for storing collections in which the same object is added consecutively many times. Rather than just store the objects one after the other, RunArrays store two collections: one of the objects in the collection, the other the number of times the object appears (Figure 4).

Each entry in a RunArray requires two object references. RunArrays require storage related not to the number of elements in the collection, but to the number of times adjacent objects are different. In the worst case, RunArrays require twice as much storage as an OrderedCollection.

Indexing into a RunArray is potentially an expensive operation, requiring time proportional to the number of runs. Here is an implementation of at:

```
RunArray>>at: anInteger
    | index |
    index := 0.
    1 to: runs size do:
        [:each |
        index + (runs at: each) >= anInteger
            ifTrue: [^values at: each].
        index := index + (runs at: each)]
```

This simple implementation makes code like:

```
1 to: runArray size do: [:each | runArray at: each]
```

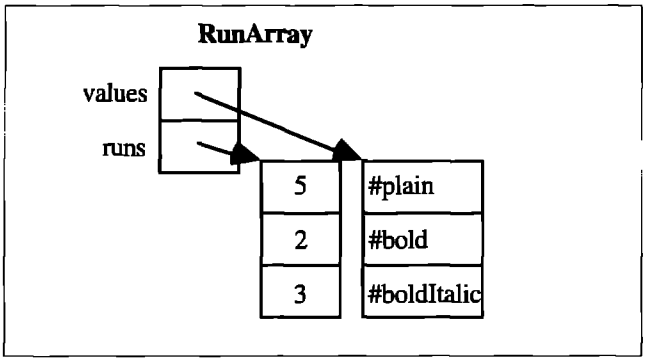


Figure 4. The result of RunArray new addAll: (plain plain plain plain plain bold bold boldItalic boldItalic boldItalic).

take time proportional to the number of runs multiplied by the number of elements in the collection. Because the access pattern for RunArrays usually marches along the collection from first element to last, RunArrays cache the beginning of the run in which the last index was found. Looking up the following index only requires checking to make sure that the new index is in the same run as the old one:

```
RunArray>>at: anInteger
    ^anInteger >= cachedIndex
        ifTrue: [self cachedAt: anInteger]
        ifFalse: [self lookUpAt: anInteger]

cachedAt: anInteger
    anInteger - cachedIndex > (runs at: cachedRun)
        ifTrue:
            [cachedIndex := cachedIndex + (runs at: cachedRun).
            cachedRun := cachedRun + 1].
    ^values at: cachedRun

lookUpAt: anInteger
    | index |
    index := 0.
    1 to: runs size do:
        [:each |
        index + (runs at: each) >= anInteger
            ifTrue: [^values at: each].
        index := index + (runs at: each)
```

With this implementation, an access pattern like the one above will now be slightly slower than the equivalent OrderedCollection because of the overhead of checking for the common case. Accessing the RunArray in reverse is now proportional to the number of runs squared.

Interval

Another kind of run-length encoded collection is Interval. An Interval is created with a beginning number, an ending number, and an optional step number (one is the default). #(1 2 3 4) and Interval from: 1 to: 4 are equivalent objects for most purposes. Number>>to: and to:by: are shorthand for Interval class>>from:to: and from:to:by:.

Intervals are commonly used to represent ranges of numbers, such as a selection in a piece of text. A common idiom is using an Interval with collect:.

```
foo
    ^(1 to: self size) collect: [:each | each -> (self at: each)]
```

Species is sent to an object when a copy is being made for use in one of the enumeration methods collect: and select:. The default implementation in Object just returns the class of the receiver. SequenceableCollection implements collect: and select:, and expects the result of self species to respond to at:put:. Since Intervals don't respond to at:put:, they have to override species to return the class Array.

SortedCollection

Another dynamically sized collection is the SortedCollection. Unlike OrderedCollections, which order their elements ac-

cording to the order in which they were added, SortedCollections rely on a two-argument block to determine, pairwise, the order for elements. This block defaults to [:a :b | a <= b], so simple SortedCollections sort their elements from lowest to highest.

One thing to watch out for when using SortedCollections is sending them add: when you don't have to. add: does a binary search of the collection, moves all of the elements after the added object down one, and inserts the added object. Moving the elements to make room takes time proportional to the size of the collection. If you know you are going to be adding several elements at once, use addAll:, which will stick the new elements at the end and resort the entire collection. Here is a method for comparing time spent using these two methods (notice that I don't hold myself to the same coding standards in workspaces):

```
| sc r t1 t2 |
sc := SortedCollection new.
r := Random new.
t1 := Time millisecondsToRun:
    [1000 timesRepeat: [sc add: r next]].
sc := SortedCollection new.
t2 := Time millisecondsToRun:
    [sc addAll: ((1 to: 1000) collect: [:each | r next])].
'Add: ', t1 printString, ' addAll: ', t2 printString
```

Executing this results in 'Add: 10725 addAll: 1386'.

String

Strings in Smalltalk are like Arrays whose elements are restricted to Characters. Strings are byte-indexable for compactness. They redefine the indexing methods to convert from 8-bit numbers to characters and vice versa:

```
String>>at: anInteger
    ^Character value: (super at: anInteger)

String>>at: anInteger put: aCharacter
    ^super at: anInteger put: aCharacter asciiValue
```

It is common to use , to concatenate Strings. You can use , to concatenate any two sequenceable collections (OrderedCollection, Array, RunArray, and so on). Less common is the use of the other collection methods with Strings. You can capitalize all the characters in a String with collect:

```
asUppercase
^self collect: [:each | each asUppercase]
```

Interestingly, even the ParcPlace release 4.1 image implements this method with five lines containing an explicit loop and indexing.

Digitalk's String class is implemented with the simple model described here. ParcPlace has a much more elaborate implementation that takes care of multibyte characters and different character sets on different platforms, even for odd characters. The design requires six classes for strings and three more for symbols.

Symbol

Symbols behave in most ways like Strings, except that if you have two symbols containing the same characters, they are guaranteed to be the same object. So while String>>= takes time proportional to the length of the strings, Symbol>>= takes constant time:

```
Symbol>>= anObject
    ^self == anObject
```

To preserve uniqueness, Symbols cannot be changed once they are created. at:put: is overridden to raise an error.

Like Interval, because Symbols don't respond to at:put:, they override species. Symbol>>species returns the class String. Thus, executing "#abc , #def" returns 'abcdef', a String, not a Symbol.

If you are programming in Smalltalk/V, be careful of creating too many symbols. There is a limit of 2^16 Symbols. While this may seem like a lot, after you have created many new methods and used Symbols for indices in several places, it is very possible to run out of Symbols. The scrambling you have to do to climb out of the "limited Symbol pit" is not pretty.

A last oddity of Symbols and Strings is the asymmetry of =. "'abc' = #abc" returns true because the String receives the message and successfully checks to see that the characters in the receiver are the same as those in the argument. "#abc = 'abc' " returns false because the two objects are not identical. I can remember long debates at Tektronix over the propriety of this strange fact. The upshot of the debates was that it's regrettable things work this way, but the alternatives are all less attractive for one reason or another.

Sets

Sets are dynamically sized collections. They respond to add: and remove: but, unlike OrderedCollections, they don't guarantee any particular ordering on the elements when they are used later (e.g., by do:). Sets also don't have any indexed access (no at: or at:put:).

Sets implement includes:, add: and remove: efficiently by hashing. The element to be added is sent hash, and that value is used modulo the size of the storage allocated for the Set as the index to start looking for a place to put the element (or remove it). Note that storage for a Set will contain more indexed variables than the Set has elements, so hashing is likely to encounter an empty slot. The Set contains an instance variable, tally, which records how many of the slots are filled. Set>>size just returns tally.

You can eliminate duplicates from any collection (albeit while losing its ordering) by sending it asSet.

IdentitySet

Sets use = to determine if they have found an object. IdentitySets use ==. They are useful where the identity of objects is important. Most applications are in meta-object code, where

you are manipulating the objects but not asking them to do anything. For instance, if you designed a remote object system where transparent copies of objects were transmitted over a network, you might store the objects in an IdentitySet. If you transmitted two objects that were = but not ==, and later changed one of them, storing them in an IdentitySet would ensure that they were different objects on the remote systems.

Bag

Instead of discarding duplicate elements like Sets, Bags count them. Executing this code:

```
| s |
s := Set new.
s addAll: #(a a b b c).s
size
```

returns 3. Changing it to a Bag:

```
| b |
b := Bag new.
b addAll: #(a a b b c).
b size
```

returns 5.

Use Bags anywhere you want a quick implementation of includes—that is, when you don't care about the order of elements and you need a compact representation of duplicate elements.

Bags are not used anywhere in the ParcPlace release 4.1 image or in Smalltalk/V Mac 1.2. The only time I can remember using Bags is in Profile/V. Every time I take a sample, I put the program counter in a Bag. When I display the profile, I map the stored program counters back to source statements, giving the user profiling at the level of individual statements.

CONCLUSION

The Collection classes are one of the most powerful parts of the Smalltalk system. Choosing the right collection for a circumstance has a dramatic influence on the behavior and performance of your system. I have tried to lay out what each major collection class does, what it is good for, what to watch out for, and how it is implemented.

I am amazed at the richness of this seemingly simple set of classes. Originally, I thought I would have to stretch to get enough material for just one column. After two columns that have covered the major issues in using collections, there is still more to be written. I'll give it a rest for now, however, and go on to something else—I'm not sure what just yet. If you have any ideas call me at 408.338.4649 or fax me at 408.338.1115. ■

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPar Computer. He is also the founder of First Class Software, which develops and distributes re-engineering products for Smalltalk. He can be reached at First Class Software, P.O. Box 226, Boulder Creek, CA 95006-0226.

PUTTING IT IN PERSPECTIVE

Rebecca Wirfs-Brock

Describing your design

Objects can be simplistic and passive, holding on to small pieces of information, or they can be busy and active, serving an important role in framing the overall architectural structure of an application. The possibilities for what an object can represent are limited only by human imagination. In this column I want to explore some effective techniques for describing classes so they can be understood, used, and refined by others. You, the author of a class or a group of collaborating classes, know how you intend them to be used. How can you effectively impart this knowledge to others? However you describe a class, your original design intent will be mulled over by different people, each with a slightly different set of expectations, needs, and experiences.

There are basic things that need to be said about any class. These essentials cover roughly 50% of the issues, which I'll cover first. Then I want to explore the remaining 50% that are often left unsaid.

COVERING THE BASICS

Each class you construct in your design has a specific purpose. You know what the class was intended to do and probably what it was never intended to do. (It is easy for someone to torture your code in ways you never dreamed of, but I don't know how to solve that problem.) You also know whether your creation is of major or minor importance, whether you have a polished implementation, or whether you have left room for improvement. The exact details you need to communicate vary depending upon the role of the reader. Different information and levels of detail are needed by:

- a programmer wanting to use this class in a program
- a developer creating a subclass to add new functionality or override existing behavior
- someone adding new functionality to your class
- anyone trying to understand a class inheritance hierarchy
- a tester developing test suites
- someone fixing a programming error

When we describe our classes and our applications, we need first to provide a global context (a road map of the territory). This provides a broad view, allowing readers to understand how individual classes fit into the overall fabric of your design. This should then be augmented by a consistent discussion of classes from both an exterior (usage) and interior (implementation) perspective. Arguably, all potential readers of class documentation need a basic understanding of how a class should be used.

Let's concentrate on what informed class users need to know. At first glance, to use a class, a programmer needs to know:

- what the class was designed to do and not to do
- ways to create an instance of that class and, subsequently, how it typically is used
- what it depends on, including other objects, global states, or host–operating system features
- where to look for further details

Subclass developers need this information to ensure that their new addition follows the expected patterns of behavior defined by its superclass. They should not fix one problem only to break pre-existing contracts with all current users of the class. They need even more details than users, but all proceed from these basics.

Not all basic information is gleaned by poring over a class-browser reading code. Some have claimed that Smalltalk's programming environment eliminates much need to describe this kind of information, but this is just another rather lame argument that XXX code (replace XXX with your favorite programming language) is self-documenting.

Learning an application by reading code and performing experiments can take a long time and often isn't the most effective way to transfer knowledge. We designers and implementors of classes should explain how to create and use our objects. Documentation should supplement a programmer's ability to find and use the right classes for the job.

From an exterior view, I certainly need to know less than someone who is intending to modify, extend, or create a subclass. I want you, the designer, to hide those things I shouldn't care about. I really don't want to concern myself with any of the object's instance variables, unless you explicitly choose to give me access to them. I also don't care about implementation details encapsulated within methods. And I certainly don't care about code that is private, intended to be executed by sending messages to self. So please label those private, internal details as such. Your chosen method partly depends on your Smalltalk environment, and partly on style guidelines used within your organization.

Understanding how to create and use an object can sometimes become confused by all that wonderful detail exposed by

the browser. This is precisely why more recent Smalltalk programming environment extensions come equipped with mechanisms and tools that explicitly enable designers to package the presentation of a class and its interfaces to casual users.

I do not want to digress into a discussion on the merits of recent additions to Smalltalk programming environments. (I am absolutely convinced of their utility.) Nor do I particularly want to defend Smalltalk against languages with explicit support for public and private declarations (which have problems in actual use). However, developers of these newer Smalltalk environments have recognized the danger of information overload. Without removing detail, it may be difficult to discover the essence of a class.

We often create an instance and only use a fraction of its class's features. And we are completely content to do so. I strongly advocate a written textual description of a class, describing the typical and most important patterns of use. Describe the essential 20%, 50%, or 80% (your percentage will vary depending on how full-featured a class is and how much exploration a programmer makes) in a few short paragraphs. Accompany this description with a few pictures describing typical object-interaction sequences. Leave the rest for me to discover by either reading through a more detailed class-design document or by exploring your code and comments. If you are trying to leave a helpful trail for users, embed a typical object-creation message with appropriate arguments inside a comment within an instance creation method. More elaborate examples can be developed with detailed comments, either to be filed into an image or executed.

SPEND TIME ON WHAT MATTERS

Not every class is worthy of the same amount of attention. A class of limited utility, intended to be seen by a very small audience, only deserves light treatment. I am not a proponent of mandating equal discussion for all classes. That leads to either lots of useless boiler-plate documentation or developer mutiny. Instead, spend the time creating a well-considered discussion for classes that provide broadly useful functionality or are central to your design.

Complex classes that require a lot of set up or have highly stylized patterns of usage demand extra attention. From an external viewpoint, I need to know common patterns of usage, as well as how to diagnose an object that's broken and not functioning as expected. We creators of initial designs often don't realize how easy it is for someone else to misinterpret our work. So this kind of discussion is definitely worthwhile, if only to get an idea of potential hot spots.

MAKING THE CONNECTIONS

It is relatively easy to produce documentation for a class intended to be used in isolation. It is much harder to describe classes that are part of a larger framework and intended to be used in conjunction with a number of collaborators. To use a framework requires understanding how objects interact, what role each object plays, and when and how objects should be created and used.

A description for a framework of interacting classes must not only cover the central classes, but also establish a clear model of how these classes are intended to work together. This year's OOPSLA conference had a refreshing paper by Professor Ralph Johnson that explained his process describing a graphical editor framework in Smalltalk, called HotDraw. HotDraw was originally developed by Ward Cunningham and Kent Beck. In five pages of text, Ralph described the central ideas behind HotDraw and documented some common patterns of key objects and their interactions. A nice touch was clear references to the next layer of detail as well as pointers to related concepts for each pattern of use.

Simple, helpful descriptions of object-interaction patterns are straightforward reading. They require that the author has a clear vision of the core ideas of a framework and a simple, if not terse, writing style.

It reminded me of the *Choose Your Own Adventure* books my kids used to read. After one or two pages, you were asked a question. Depending on your answer, you were directed to one of two pages. You could read the entire book and get several different stories, each with different endings. My kids were never satisfied until they had explored all possible paths.

Documentation of interlocking classes of objects needs this touch. First you need a description of core concepts. Then you need to tour key interactions at your own pace, allowing you to discover and explore according to your personal choices. Descriptions should let you navigate, point you to more detail (if you want it), and let you move on (should you want to broaden your understanding).

CONCLUSION

New, useful ways for describing classes of objects and groups of cooperating objects are active research topics. There's plenty of room for formal techniques as well as informal descriptions. What I constantly strive for are pragmatic ways to impart design insight to users.

I don't want you to leave with an impending sense of doom or writer's block. I don't like writing reams of paper that no one reads. And I won't recommend that you take extraordinary measures nor do what I personally am not willing to do myself.

I especially want to appeal to you cynics who might be thinking as you read this, "But she's a writer. Of course she can recommend we do these things. Writing comes naturally to her." Writing is definitely not a natural act for me. I have to struggle to write concise, precise documentation. But as a user of some pretty nicely described systems, I encourage you to perform an enormous service to your users. Take some time to describe how to properly use your classes. ■

Rebecca Wirfs-Brock is Director of Object Technology Services at Digitalk and co-author of DESIGNING OBJECT-ORIENTED SOFTWARE. *Comments, further insights, or wild speculations are greatly appreciated by the author. Rebecca can be reached via e-mail at rebecca@digitalk.com. Her U.S. mail address is Digitalk, 7585 SW Mohawk Street, Tualatin, Oregon 97062.*

BOOK REVIEW

Richard L. Peskin

OBJECT-ORIENTED ENGINEERING

by John R. Bourne

The subtitle of this book is *Building Engineering Systems Using Smalltalk-80.* It is to Bourne's credit that he addresses the important topic of engineering applications of object-oriented software systems. While simulation was a primary target of early object-oriented languages, such as Simula and original versions of Smalltalk, more recent activity in the subject area appears to emphasize business applications, data base applications, etc. If Smalltalk is to take its place alongside more commonly accepted languages, its success in scientific and engineering applications will have to be demonstrated on a much broader scale than is present today. Bourne's effort provides an important step in that direction, namely a book that addresses uses of Smalltalk in the engineering domains.

The author has made some valuable contributions to applications of Smalltalk in the college classroom, one example being his work on engineering tutorial systems implemented in Smalltalk. The book, however, is somewhat disappointing as a classroom tool or general resource for engineers who want to learn more about Smalltalk's potential for technical applications. The material is much too general in its treatment of actual engineering applications, yet at the same time contains too much code-level detail without providing sufficient preparation for beginners.

Part I is an overview of general concepts such as representation of physical processes in terms of objects and behaviors. A serious deficiency is the lack of historical perspective and presentation of important recent contributions in engineering applications of Smalltalk. Notably absent is any mention of the contributions of the (now defunct) Tektronix group. Applications such as INKA, a system that assists in instrument service, represent important real engineering Smalltalk projects. Also omitted are the contributions of Thomas et al. on the uses of Smalltalk in realtime instrumentation and control, work done at Rutgers on scientific data management, and other real-world cases discussed in recent journals and proceedings. Engineers need to be motivated by actual applications.

Turning to more specific issues, this reviewer would have liked to have seen more emphasis on behavioral paradigm, as opposed to software structural aspects (inheritance, etc.). Encapsulated behavior of objects is the crux of what Smalltalk has to offer engineering simulation. Bourne puts much emphasis on the MVC paradigm and attempts to draw real-world analogies. Not only is MVC out of date, but the author's analogies are somewhat questionable. My greatest criticism of this part, and of the book as a whole, is the emphasis it places on use of ACOM cards for the "pre-specification" of a Smalltalk design. Bourne goes so far as say that one must use 4x6 cards as opposed to 3x5 cards for writing down the desired classes, protocols, etc. This approach reflects the traditional "specification" approach to software, not the interactive prototyping style that is Smalltalk's forte. Although he references a 1986 paper by Cunningham and Beck as his rationale for emphasis on ACOM cards, my own reading of that paper was that cards were only a "literary aide" to help explain O-O concepts. The first part of the book ends with an overview of other O-O languages, in which the author does emphasize the importance of having a complete class library for a particular O-O environment to be of real benefit.

Part II concentrates on "tools," namely the Smalltalk language and environment. This section does not flow smoothly from topic to topic and I fear it will be difficult for beginners to follow. Smalltalk code examples are presented in numerous figures without proper preparation for the lay reader. Perhaps Bourne intended this section to be covered by additional classroom material. In addition to Smalltalk specifics, this section covers issues such as "look and feel" (but omitting that PPS release 4 does not have a complete native platform look and feel) and bit editors (without making clear that release 4 does not really support this and Pens as part of the system). As in Part I, great store is place on the ACOM card method and how to transfer information from the cards to the Browser. However, there are some useful pieces in this section. While the discussion on page 147 mixes animation with drawing, at least one is shown how to draw a line using PPS release 4. Chapter 8 concentrates on MVC. There is too much detail, particularly about the viewBuilder, and that level of detail is really not germane to the subject of engineering applications. It is interesting to note that the author's own code example for MVC illustrates the typical MVC problem; that is, where to put drawing methods. The "Counter" examples ParcPlace used to distribute would be better in this context. The author discusses the "Pluggable Gauges" package (from KSC), but doesn't refer to the active value concept that is central to that package and important to engineering applications.

Part III deals with engineering applications, which I found to be the most disappointing. Most of the discussion about exam-

ples is cursory at best. There is a need in this section for emphasis on real examples. The non–electrical engineering coverage is understandably the weakest, but his circuit simulation example is again too detailed with emphasis on code rather than simulation of physical behaviors. As in prior parts of the book, details of extraneous subjects take up too much space—the external interface description is a notable example. While Bourne does not face some critical issues in engineering applications of Smalltalk, such as handling of large numbers of objects generated in technical computations, he does address performance problems with a discussion of user primitives. However, he confuses user primitives (which are limited by the context loss across calls in PPS release 4) and a true C interface (not yet released for PPS at this writing). Table 12.2 illustrates the serious problem with this book. It is a method listing consisting of user prims (<primitive: 11106>, etc.) with no comments, and is presented before the reader is even introduced to the necessary semantics. The book does end with a fairly good discussion of simulation and Smalltalk applications in simulation. Perhaps this discussion should have been presented much earlier.

All in all, I was disappointed. Given the great need for books and monographs on scientific and engineering applications of Smalltalk, perhaps I expected too much. In all fairness, the book is accompanied by an instructor's manual and code disks, which were not available in time for this review. Perhaps their presence would have presented the text in a different viewpoint. Future books on this topic should emphasize Smalltalk as a behavioral paradigm for computational simulation of physical processes. This important "forest" should not be hidden by "trees" of small details. ■

Richard L. Peskin is Professor of Mechanical and Aerospace Engineering at Rutgers University and director of the CAIP Center Computational Engineering Systems Lab. He has been involved with engineering and scientific aspects of Smalltalk since 1984. In addition to doing research in computational fluid dynamics and non-linear dynamics, he is one of the designers of the SCENE (Scientific Computation Environment for Numerical Experimentation) system, a Smalltalk-based distributed computing environment that implements computational steering tools such as interactive scientific graphics and data management, automatic equation solvers, and mathematical expert systems.

Highlights

Excerpts from industry publications

CONCEPTS

. . . In most languages, learning to program means learning the syntax. Learning to program in Smalltalk, however, involves much more. The programmer must have a clear grasp of object-oriented concepts. In addition, Smalltalk's development environment strongly influences the entire approach to software creation. It is absolutely essential that the developer become familiar with the classes provided by the Smalltalk environment. Although this can take some effort, it's a prerequisite for developing more than the most trivial programs. Fortunately, this is an interesting activity and is one of the best ways to learn Smalltalk.

An earful of Smalltalk, John D. Williams, PCAI, 9-10/92

TASKS

. . . The tasks in an object-oriented effort are different. New tasks are required to identify, characterize and document objects. These tasks focus on identifying objects and the interactions required of these objects to provide a system that meets stated requirements. Object-oriented efforts, like other development approaches, need requirements and design specifications. Yet these documents localize around objects, and not functions or data. In addition, these specifications clearly delineate which components are reused from an in-house reusability library and which are developed from scratch to support the application at hand. Tasks associated with the construction of structure charts, data flow diagrams and other function- or data-oriented modules are obsolete and replaced with modeling approaches more in concert with object-oriented development.

Designing the object-oriented way, Ron Schultz, OPEN SYSTEMS TODAY, 7/20/92

END-USER DEVELOPERS

. . . No fundamental change in the pace of software development can occur until there is a significantly higher level of application development. In other words, end users must become developers. Object-oriented programming could allow end users to do just that. The ideal application development environment would consist of enormous libraries of prefabricated, modular program parts (super high-level objects). These modules could be configured and combined in virtually unlimited combinations to build complete applications across the entire spectrum of software use. Applications would be built exclusively in a high-level tool of this sort. Conventional code-level programming would focus on creating object components. . . . End users would have unprecedented programming opportunities.

The new shangri-la?, Joseph Firmage, SOFTWARE MAGAZINE, 7/92

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

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

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

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

ONE-STOP SHOPPING.

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

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

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

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

IMMEDIATE RESULTS.

Digitalk's training gives you practical information and techniques you can put to work immediately on your project. Just ask our clients like IBM, Bank of America, Progressive Insurance, Puget Power & Light, U.S. Sprint, plus many others. And Digitalk is one of only eight companies in IBM's International Alliance for AD/Cycle—IBM's software development strategy for the 1990's. For a full description and schedule of classes, call (800) 888-6892 x412.

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

100% PURE OBJECT TRAINING.

DIGITALK