The Smalltalk Report

The International Newsletter for Smalltalk Programmers

October 1991 **Volume 1 Number 2**

A MATTER OF STYLE

By Ed Klimas & Suzanne Skublics

Contents:

CONSISTENCY? WHY BOTHER?

In an era where the software industry is concerned with reliability, standard interfaces, duplication of effort, maintenance cost, feature creep and increasingly shorter market windows, code reuse is proving to be a solution. Smalltalk is a language that promotes code reuse. The industry is recognizing Smalltalk's potential for reducing development costs, improving reliability, improving productivity, and improving a company's software competitive edge.

Code reuse does not come automatically by simply using Smalltalk. Just because code can be used does not mean it can be reused. Applying a clear and consistent programming style will help make code easier to read, maintain, and subsequently reuse. Strictly following a set of style *rules* is not necessary; however, a set of simple and consistent style *guidelines* should help a programmer design reusable code. A guideline is simply a recommended practice. A set of style guidelines is important: experience shows that without a sound and consistent coding style, dirty code gets dirtier!

In this article, we present a sample set of guidelines to help make object-oriented source code easier to read, maintain, and reuse. The guidelines are based upon existing good software engineering practices that are used by developers of commercial software. In many object-oriented programming language reviews, Smalltalk is used as the basis of comparison because of its relatively complete set of features; it is a good language to use to discuss object-oriented programming concepts. We use Smalltalk terminology and examples to present our guidelines; however, the principles of style apply to all object-oriented languages.

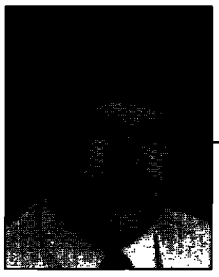
REAL PROGRAMMERS CAN WRITE FORTRAN IN ANY LANGUAGE

Many procedural programmers new to Smalltalk tend to program with the style of their mother tongue programming language. This usually leads to convoluted code that does not take advantage of the features of Smalltalk that make it suitable for rapidly building high-quality systems. Unfortunately, a set of guidelines to follow when programming in Smalltalk has yet to be presented. We attempt to fill that void with our "Smalltalk with Style" column. In this, our first column, we address source code presentation from a Smalltalk perspective. Future columns will cover naming conventions, standard protocols, frameworks, good object-oriented Smalltalk programs, quality assurance and testability, and software metrics.

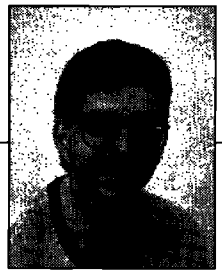
GUIDELINES FOR EVERYONE!

The guidelines are intended for people involved in the development of software systems written in an object-oriented language such as Smalltalk. Readers with different levels of Smalltalk experience and different roles in a software project will use the guidelines in different ways. We do not necessarily recommend that you follow the guidelines strictly but, rather, that you adapt them to suit your particular project or organization. The most important idea to keep in mind is *consistency*.

continued on page 4...



John Pugh



Paul White

EDITORS' CORNER

It seems very strange to be writing the editorial for our second issue without the benefit of feedback from you on our premiere issue. Such are the vagaries of publication deadlines! In any event, we look forward to receiving your letters and comments on the contents and format of *The Smalltalk Report*.

As the lead article in this issue, "A Matter of Style," we focus on style in Smalltalk programming. Many organizations have successfully developed projects using Smalltalk technology, and it is important that the experience gained and lessons learned from these projects be passed on to the organizations that are just now migrating to Smalltalk. In the first of their regularly appearing "Smalltalk with Style" columns, Ed Klimas and Suzanne Skublics offer stylistic guidelines for the presentation of Smalltalk source code. In upcoming issues, they plan to address topics such as reusability, naming conventions, finding application frameworks, and software metrics. This column will be a "must read" for both beginning and experienced Smalltalk programmers.

An exception-handling mechanism for gracefully taking corrective action when unusual and unanticipated conditions arise has long been a feature of languages such as Ada. ParcPlace first introduced facilities for handling exceptions in release 2.4 of Smalltalk-80, but no such facility has been added to Digitalk's Smalltalk/V. In his article, "Exception Handling in Smalltalk," Boris Gärtner looks at a number of potential ways in which a simple mechanism could be implemented for Smalltalk/V. He first describes a method for adding a handler for a newly defined application class and then expands his implementation to handle exceptions generated by class methods. Finally, he proposes a more general approach.

In the continuation of their first column, Greg Hendley and Eric Smith complete their study of how to give application windows dialog box functionality in Smalltalk/V PM with a discussion of window ownership in OS/2. Also in this issue, Juanita Ewing makes suggestions as to how Smalltalk programming teams might organize their applications.

Complementing the review of the WindowBuilder/V product from Acumen, which appeared in our premiere issue, Cahan O'Ryan reviews the Tigre Programming Environment for Objectworks\Smalltalk. Also, Dan Lesage reviews Dan Shafer and Dean Ritz's book entitled *Practical Smalltalk*.

This is a busy time of the year for conferences of interest to Smalltalkers. *The Smalltalk Report* will have a presence at the first Digitalk Developers' Conference in Los Angeles, OOPSLA '91 in Phoenix, and the first Eastern European Conference on Object-Oriented Programming in Bratislava, Czechoslovakia. We look forward to seeing you there!

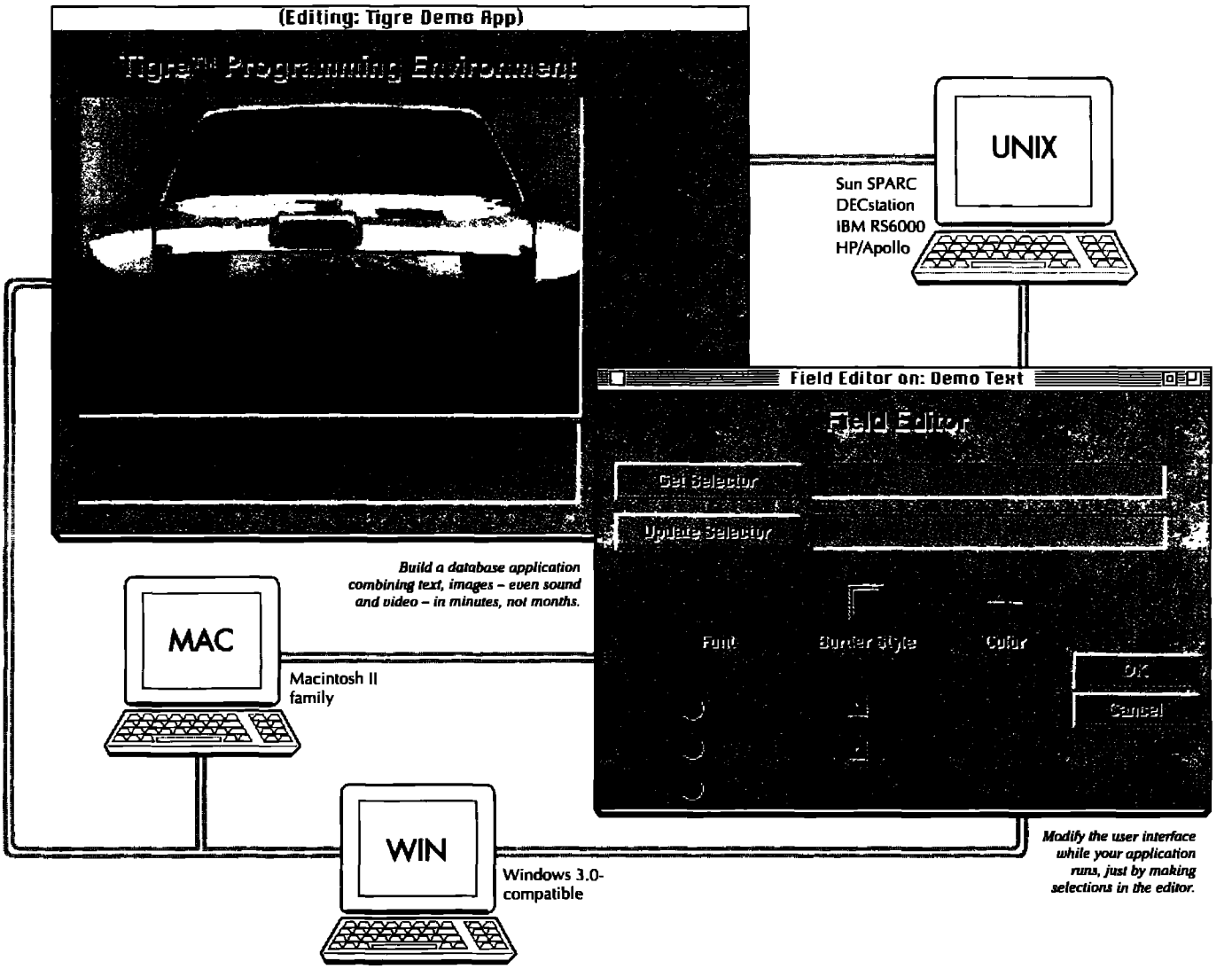
Enjoy the second issue!

John Pugh and Paul White
Editors

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 COOT, Inc., a member of the SIGS Publications Group, 588 Broadway, New York, NY 10012 (212)274-0640.

Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90, Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: THE SMALLTALK REPORT, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada, (613) 230-6897, or fax (613) 235-8256.

Fastest Path to Platform Independence.



Leap free of platform limitations and deliver full-color GUI applications in half the time...with Tigre™.

Introducing an incredible OOP breakthrough: A complete development environment that lets you create object-oriented, multi-user applications that run across all major platforms and networks. *And* lets you deliver them up to 80% faster than ever before.

Tigre™ Programming Environment, running with Objectworks® \ Smalltalk Release 4, offers a set of tools that turns a major hassle into a quick drag. Literally. Because it lets you build customized, color GUIs just by dragging and dropping.

You'll choose from a large library of user interface components. Objects like scrolling text fields, check boxes, radio buttons and more.

Drag them from the palette onto your application screen. Move and resize them as often as necessary. No recompiling needed. And virtually no code to write. Tigre's Interface Designer automatically creates the Smalltalk GUI for you.

Give the interface your unique imprint by clicking selections to change color, font, borders, icons, etc. And you can add your own custom GUI creations to the library for reuse.

Use Tigre's multi-user, object-oriented database manager, to provide network-compatible access to text, images, icons, sounds – any type of stored data.

Phone now for a complete package of information on Tigre. There's never been a faster track to freedom.

TIGRE OBJECT SYSTEMS, INC.

Call: (408) 427-4900, Fax: (408) 457-1015
3004 Mission Street, Santa Cruz, CA 95060

continued from page 1...

If you are a novice programmer learning Smalltalk, you will gain an advantage by following guidelines and methods early in your programming career. This will help you develop a clear programming style that effectively exploits whatever languages you will be using.

If you are an experienced Smalltalk programmer, you probably already develop code that conforms to many of the guidelines. The set of guidelines in this column embody a widely accepted approach that will make your code more consistent and easier to reuse.

If you are a technical manager, you probably already follow a set of corporate standards. Guidelines will help technical managers ensure that the software produced during a project is correct, reliable, easy to maintain, and reusable. The difficulty lies in creating a project-wide commitment to adhere to a set of guidelines.

SOURCE CODE PRESENTATION

In an environment such as Smalltalk, more time is spent reading code than writing it. The physical layout of source code on the page or screen strongly influences how easy it is to read. "A program is not only a set of instructions for a computer, but a set of instructions that must be understood by a human, especially the one who reads it the most—the programmer."[1] In the OO community, easy-to-read code relates directly to greater code reuse: it is more likely that someone will be able to reuse code if they can at least read and understand it. It is important for code to be well structured, for example, in terms of indentation and naming standards. However, defining a good structure is subjective. We present a number of guidelines that define general principles of a good layout but do not prescribe a particular layout style. The decisions about how to apply these principles are better left to the project leader or organization.

GUIDELINE 1: BE CONSISTENT

The formatting of Smalltalk source code affects how the code looks, not what it does. The most important guideline to follow is to be consistent throughout the application and the project.

GUIDELINE 2: USE A CONSISTENT SPACING STRATEGY

Spacing makes source code easier to read because it emphasizes the delimiters on a source statement line. Constructs are easier to recognize irrespective of where they occur in program text. A consistent spacing strategy applies to many facets of a piece of source code. For example, delimiters such as binary operators and parenthesis are easier for the reader to parse when separated from other programming constructs.

Examples:

```
3 + 4 * 36 >= 32 + (-32).
#(1 2 3), #(4 5 6).
'hello', 'there'.
#((2 3) (3 4) (4 5))
#( red )
```

GUIDELINE 3: INDENT AND ALIGN NESTED CONTROL STRUCTURES AND CONTINUATION LINES CONSISTENTLY

Source code that is consistently indented is easier to read because the structure and flow of a program are easier to see. Nesting levels can be clearly identified by indentation. Cascaded messages, for example, are easier to follow when the receiver object is separated from the messages, each indented on a separate line. The fact that the code is indented consistently is more important than the number of spaces used. An indentation of four spaces or a tab is typical.

Related to indentation is alignment. Alignment makes it easier to see the position of the operators and, therefore, places visual emphasis on what the code is doing. Statements that include nested control structures or long expressions that continue over more than one line are easier to read and parse if they are aligned on separate lines. The flow of control of a program can also be reflected by alignment. The particular style used is not as important as applying a consistent alignment. We recommend an alignment style in the examples presented, such as matching alternative cases in an ifTrue:ifFalse: message.

Examples:

```
"Single alternative statements on the same line as the condition."
(self includesKey: aKey) ifTrue: [^self].

"Blocks with short expressions contained on single lines."
aBooleanExpression
    ifTrue: [aShortExpression]
    ifFalse: [aShortExpression].

"Blocks with long expressions contained on several separate lines."
aBooleanExpression
    ifTrue: [
        aLongExpression.
        aLongExpression]
    ifFalse: [
        aLongExpression.
        aLongExpression].

"Enumeration messages indented and aligned to reflect control
flow."
self value
    ifTrue: [
        aBlock value.
        self whileTrue: aBlock].

"Indented cascaded messages on separate lines."
self
    updateFather;
    updateMother.
```

```
"Indented long key word messages to avoid line wraps."
magnifiedForm
    displayOn: aDisplayMedium
    at: absolutePoint - alignmentPoint
    clippingBox: clipRectangle
    rule: ruleInteger
    mask: aForm

"Long, cascaded key word messages with a blank line between each
message."
anOrderedCollection
    replaceFrom: 2
    to: 3
    with: #(a b c d e f g)
    startingAt: 3;

    replaceFrom: 7
    to: 8
    with: #(a b c d e f g)
    startingAt: 5.
```

GUIDELINE 4: START EACH STATEMENT ON A NEW LINE; NO MORE THAN ONE SIMPLE STATEMENT PER LINE

It is easier to locate variable assignments when they are aligned along the left margin. A single statement on each line makes statements easier to distinguish. Similarly, the structure of a compound statement is clearer when its parts are on separate lines. If the statement is longer than the remaining space on the line, continue it on the next line or restructure the code so that it can be cascaded onto separate lines.

Examples:

```
"Follows guideline..."
compositionRectangle := compositionRect copy.
text := aText.
textStyle := aTextStyle.
firstIndent := textStyle firstIndent.
rule := DefaultRule.
mask := DefaultMask.

"Does not follow guideline..."
compositionRectangle := compositionRect copy.
text := aText.  textStyle := aTextStyle.
firstIndent := textStyle firstIndent.
rule := DefaultRule.  mask := DefaultMask.
```

GUIDELINE 5: BREAK COMPOUND STATEMENTS (LONG KEY WORD MESSAGES) OVER MULTIPLE LINES

Examples:

```
"Follows guideline..."
aMenu
        "Answer a menu with a list of miscellaneous operations."
    ^(Menu
        labels: 'Clear\Copy\Paste\Fonts...\Pen size\BitEdit'
withCrs
        lines: #(4)
        selectors: #(clear copyGraph pasteGraph changeFont
                changeSize bitEdit))
            owner: self;
            title: '&Options';
            yourself

"Does not follow guideline ..."
aMenu
        "Answer a menu with a list of miscellaneous operations."
    ^(Menu labels: 'Clear\Copy\Paste\Fonts...\Pen size\BitEdit'
            withCrs lines: #(4)
        selectors: #(clear copyGraph pasteGraph changeFont
                changeSize bitEdit))
            owner: self; title: '&Options'; yourself
```

GUIDELINE 6: USE BLANK LINES TO CONVEY SEMANTICS

Blank lines should be used to separate chunks of code that perform different tasks. It is easier to read and understand code that is semantically grouped. The need for blank lines is not as great in Smalltalk because methods group logical chunks of code. However, this guideline should not be ignored simply to make the code fit on the screen.

CONCLUSIONS

Object-oriented programming languages such as Smalltalk support many of the modern software practices that can significantly improve the productivity and quality necessary to meet today's shorter commercial market development windows. We have addressed how applying a clear and consistent programming style can promote the software practice of code reuse. We have presented a set of guidelines that will help an object-oriented programmer develop code that easier to read, maintain, and reuse. We emphasize that the guidelines need not be strictly followed but, rather, that the strategies of each be applied consistently. Applying source code presentation guidelines does not guarantee that code will be reused, but it is an important step toward obtaining the full benefits of highly reusable code development. ✣

REFERENCE

[1] Legard, H., P. Magin and J. Hueras. *Pascal with Style: Programming Proverbs*, Hayden Book Co., Hasbrouck Heights, NJ, 1979, p. 2.

Ed Klimas has been involved with the implementation and application of industrial software in industry as well as modern object-oriented programming practices in commerical real-time industrial control systems.

Suzanne Skublics is the Education Manager at Object Technology International. She has been involved with the Smalltalk/object-oriented programming community for several years, particularly with Carleton University.

Ed and Suzanne are co-authors of Smalltalk with Style, *a forthcoming book to be published by Addison-Wesley. The guidelines and examples for this column series are excerpts from this book.*

GETTING REAL

Juanita Ewing

How should teams organize their applications?

IN MY PREVIOUS COLUMN, I began to address some important issues for teams of Smalltalk programmers. Teams of programmers are important because large complex applications cannot be built by a single programmer. Continuing with issues relating to teams, this column will present heuristics for organizing applications. Organizational units can be the basis of work assignments for team members and the basis for distributing completed portions of an application. An additional benefit is that organizational units tend to represent reusable units.

HOW SHOULD TEAMS ORGANIZE THEIR APPLICATIONS?

When a team of programmers implements an application, the development work needs to be structured and distributed among the team members. Without some kind of organization, development would be a free-for-all, and no schedule would be possible. The most obvious organizational technique is to partition an application along class lines. In this organizational scheme, each member of the team would be responsible for implementing and maintaining a group of classes. In Smalltalk, a class is a unit that encapsulates the behavior and data specifications for a particular kind of object. An organization based on classes has the advantage of being built on an existing supported Smalltalk unit and is able to use many of the existing Smalltalk tools. But, classes don't exist in isolation.

SHOULD HIERARCHICALLY RELATED CLASSES BE ORGANIZED TOGETHER?

Classes are usually part of a hierarchy in which superclasses also specify data and behavior. The behavior of an object is defined by the behavior in its own class and the behavior of its superclasses. Since a class requires its superclass to function, it is desirable to organize both classes together. This desire is the basis of our first heuristic.

This kind of grouping usually involves several classes since an inheritance tree is frequently larger than just two classes. Entire trees of hierarchically related classes might be grouped together to satisfy this heuristic, but it cannot be followed blindly. If it were, most of the classes in an image would be grouped together.

HOW DO YOU LIMIT THE HIERARCHICAL GROUPS?

If classes were grouped strictly by inheritance, the size of groups would not be reasonable. Use the additional heuristic that hierarchically related classes performing a similar function should be grouped together.

For example, suppose you are developing an application that has a plumbing system. The plumbing system is composed of plumbing components such as valves, spigots, and pipes. All of the plumbing components are subclasses of an abstract class, PlumbingComponent. PlumbingComponent is a subclass of Object. A group based on function would contain PlumbingComponent and all its subclasses, but would not contain the superclass Object because it does not fulfill the same function as a plumbing component.

SHOULD COLLABORATING CLASSES BE ORGANIZED TOGETHER?

Frequently, an application contains several classes that send messages back and forth. These classes collaborate. Collaborating classes require each other to function. Because these individual classes don't stand alone, it is desirable to organize these classes together.

The degree of collaboration affects this organizational heuristic. If two classes collaborate with just one message, then the degree of collaboration is small. Many messages indicate a large degree of collaboration and a stronger reason to organize the classes together.

Suppose our plumbing system contains a water heater. A water heater has a water tank, a heating element, and a thermostat. The heating element must be turned on and off when the water temperature, as sensed by the thermostat, reaches upper and lower limits. The thermostat sends messages such as

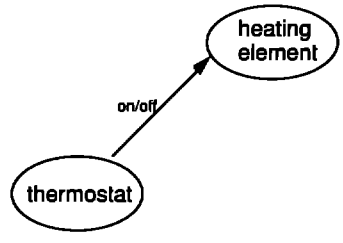


Figure 1. Grouping collaborating objects.

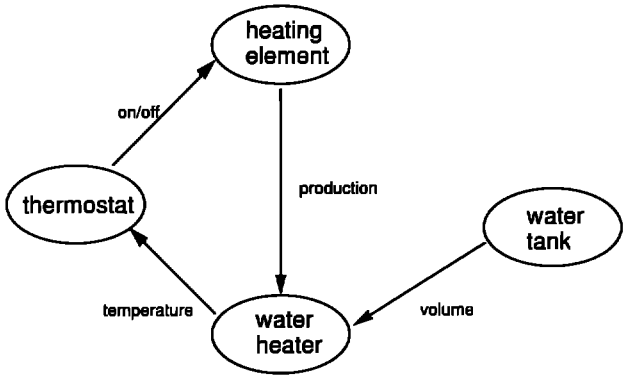


Figure 2. Determining which objects should be included in the water heater group.

turnOn and turnOff to the heating element. These two classes collaborate and, therefore, should be grouped together (Fig. 1).

We haven't addressed the issue of how to organize the water tank class in our example. The heating element would send messages indicating how much heat it has produced, and, based on the volume of water, a temperature rise could be calculated. Does the water tank perform the temperature rise calculation? No. The water tank is responsible for knowing its volume of water. However, nothing about a generic water tank suggests that it be able to calculate temperature rises. (We will ignore volume fluctuations based on temperature variations.)

Our system also needs to include a water heater object that performs operations specific to a water heater, such as calculating the temperature rise. The heating element communicates with the water heater object to pass on heat production, and the water heater tells the thermostat the current temperature. (See Fig. 2.)

Because of the collaboration between the water heater and both the heating element and the thermostat, the water heater should also be included in the organization based on collaboration. The water tank collaborates with only one of the other classes in this example. Because of the small degree of collaboration and also because the information doesn't take an active role in the primary calculation, we leave it out of the water heater group.

Our group contains three classes: heating element, thermostat, and water heater. This organization is based solely on collaboration (Fig. 3).

WHAT IF YOUR CLASSES ARE IN A HIERARCHY AND COLLABORATING?

It is likely that your application contains classes that belong to a hierarchy and also collaborate with other unrelated classes. Both hierarchical and nonhierarchical relationships should be taken into account. Classes in the hierarchy should be organized together, and tightly coupled classes in the application should be organized together.

Let's examine the water heater example. Some of the objects in this system are hierarchically related. The thermostat and the heating element are part of an electrical component hierarchy, and the tank is part of the plumbing component hierarchy. Yet we also want to capture the relationships based on collaboration, as depicted in Figure 2. There is a desire to associate the heating element with other electrical components as well as with the other classes comprising the water heater.

HOW DO YOU ORGANIZE A CLASS IN MORE THAN ONE WAY?

We have discussed a unit that captures a single organization. Let's call this unit the *primary organizational unit*. To represent multiple overlapping associations, we need another type of organization. Configurations are another type of organization that is used to represent secondary relationships. *Configurations* refer to other organizational units. As such, they are another level of organization. They can be nested, so that one configuration may refer to another configuration or simply to a group of primary organizational units.

In most cases, a hierarchical relationship forms the basis for the primary organizational unit. This unit can then be combined with other units via configurations. This tactic reflects the point of view that the hierarchical relationship is tighter and more stable than collaboration-based relationships.

Another way to think about different organizations is to imagine scenarios for reuse and maintenance. If developers are more likely to reuse a hierarchy of classes than a group of collaborating classes, then the primary organization should be based on inheritance. If a group of classes will be maintained as a unit, this means that they are closely related and should be grouped together.

In the plumbing example, all the plumbing components can be organized into a primary unit. This unit contains classes related by inheritance. The same should be done for the electrical components. A configuration representing the water heater would contain three primary units:

- the plumbing components unit (for the water tank)

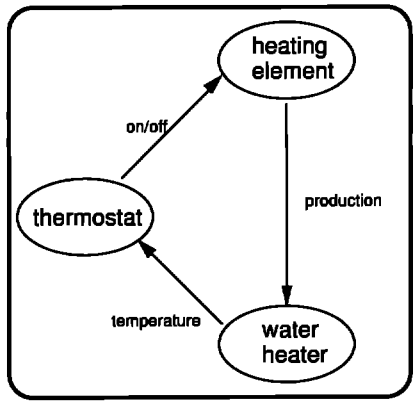


Figure 3. The water heater group.

- the electrical components unit (for the thermostat and heating element)
- the water heater unit consisting of only the water heater class

With inheritance as the organizational basis for primary units, collaboration-based relationships can be represented by configurations. In our example, this organization is useful because the plumbing components can exist in different systems. You can imagine the configurations and primary organizational units needed to represent a well and pump or a solar hot water heater.

WHAT ABOUT RELATED CODE IN OTHER CLASSES?

All parts of an application might be neatly contained in classes. Frequently, though, methods will be sprinkled throughout the class library.

Suppose a way to distinguish between other objects and plumbing components is needed. It is reasonable for a developer to define a method in Object that answers whether the receiver is a plumbing component (isPlumbingComponent). This method returns false. A similar method implemented in PlumbingComponent returns true. All classes inheriting the method from Object will answer false when asked if they are a plumbing component. Subclasses are free to override the method.

SHOULD CODE ORGANIZATION BE BASED ON CLASSES?

In our example, a single method in an unrelated class has functionality that relates to another class. How should this method be organized? Should the method in Object be associated with the class Object or should it be part of the plumbing component unit? Obviously, this method relates to plumbing components and not to the generalized behavior of objects in a Smalltalk system. It should be associated with the plumbing component classes.

Organizing strictly along class boundaries is not flexible enough. I advocate a flexible grouping scheme in which classes and methods can be organized together into primary units. (We will refer to classes and methods as definitions.) Use the heuristic that functionally related definitions should be organized together without regard to class boundaries.

In the plumbing system example, the classes composing the plumbing component hierarchy and the method Object>isPlumbingComponent should be bundled into one primary unit. Both implementations of isPlumbingComponent would be contained by the same primary unit. It is likely that many of the definitions would be used together and maintained together. In particular, if the meaning of Object>isPlumbingComponent is changed, then PlumbingComponent>isPlumbingComponent is also likely to change.

Use these heuristics to organize your application:

- organize hierarchically related classes together
- use functionality to limit the size of hierarchically based groups
- organize collaborating classes together
- put functionally related definitions together

Two types of organization are required to represent different kinds of relationships and to retain the flexibility required in a highly productive environment like Smalltalk. With the primary organization units, a developer can bundle definitions together that are closely related—either through inheritance or collaboration. These definitions are maintained together and reused together. Primary organizational units contain logical groups of definitions that cannot stand alone.

Configurations are another level of organizational structure that represent secondary relationships. The components of a configuration stand alone and are more likely to be used in other situations. Developers should be encouraged to mix and match different organizational units to extend the usability of a set of definitions. Secondary relationships, represented by configurations, are the basis of mixing and matching.

Primary organizational units are suitable for organizing the development of an application. Each team member should be assigned to implement one or more primary organizational unit, and the implemented units should be distributed to other team members. Configurations are used to build up the various subsystems in an application and ultimately to specify the application itself. ✤

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

EXCEPTION HANDLING IN SMALLTALK

Boris Gärtner

EXCEPTION HANDLING is a method to be used whenever program crashes may not be tolerated in an important application. Most well known are the methods included in the languages Ada[1] and CHILL.[2] Recently, ParcPlace decided to incorporate exception-handling facilities into its Smalltalk-80 version 2.5 and Objectworks\Smalltalk release 4 systems.[3]

The features of Objectworks\Smalltalk are implemented with newly introduced primitives. For the user of Smalltalk/V, it is a challenge to implement exception handling in the language itself. This contribution sketches two methods based solely on elements of Smalltalk as defined in the "blue book."[4] The simpler method is suitable if exception handling should be limited to some classes, whereas the more sophisticated method offers as much generality as possible.

A SIMPLE METHOD

EXCEPTIONS IN NEWLY DEFINED CLASSES

Implementation of exception handling is simple if it can be done from the very beginning. All you need to do is to provide an instance variable where the exception handler can be stored. A suitable handler will be provided during instance creation.

A handler can be any of the following blocks:

```
[ <any sequence of expressions>
    ^self]

[:message | <any sequence of expressions>
    ^self]

[:sender :message | <any sequence of expressions>
    ^self]
```

To raise an exception, one has to write:

```
<exceptionHandler> value.
<exceptionHandler> value: <argument>
<exceptionHandler> value: self value: <argument>
```

The essential point is that a block is bound to the method of its definition. Consequently, the execution of ^self will terminate the execution of the method containing the block definition. The ^self is nothing but a GOTO in disguise.

All three forms of exception handlers can be intermingled. But, for uniformity as well as for clarity, it is convenient to use only handlers with two arguments. The value of the first argument shall always be the signaling instance; the value of the second argument shall be a symbol communicating the kind of exception.

EXAMPLE: STACKS AND QUEUES

The class Storage implements the common protocol of its subclasses — Stack and Queue. The exception handling is part of the common protocol of both classes.

```
Object subclass:                    #Storage
       instanceVariableNames:       'field exceptionHandler '
       classVariableNames: ''
       poolDictionaries: ''
```

Storage Class Methods

```
new: amount atError: aBlock

    ^super new
        exceptionHandler: aBlock;
        init: amount
```

Storage Methods

```
capacity

    ^ field size

exception: aSymbol

    exceptionHandler notNil
        ifTrue: [exceptionHandler   value: self
                                    value: aSymbol]
        ifFalse:
            [super error: 'unhandled exception: ', aSymbol asString]

exceptionHandler: aBlock

    " aBlock is expected to be a block with two arguments. The first
      argument should take the sender (i.e., self); the second
      argument should take the message symbol. "

    exceptionHandler := aBlock

isEmpty
```

```
    ^self size = 0

isFull

    ^self size = self capacity

Storage    subclass:                    #Queue
           instanceVariableNames:       'input output count '
           classVariableNames: ''
           poolDictionaries: ''
```

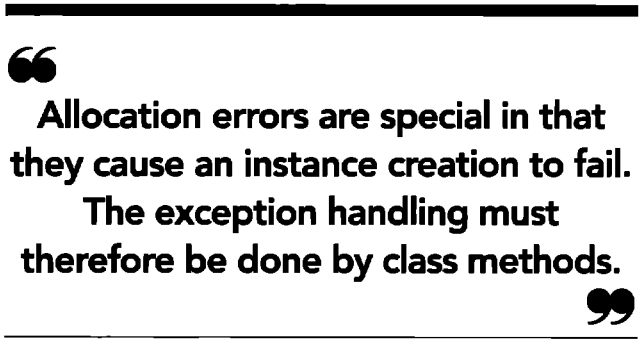
Queue Methods

```
init:  amount
    field := Array new: amount.
    input := 1.
    output := 1.
    count := 0

pop
    | var |
        self isEmpty
            ifTrue:    [self exception: #isEmpty]
            ifFalse:   [var := field at: output.
                        count := count - 1.
                        output := output \\ |self size| + 1.
                        ^ var]

push: x
    self isFull
        ifTrue:    [self exception: #isFull]
        ifFalse:   [field at: input put: x.
                    input := input \\ |self size| + 1.
                    count := count + 1.
                    ^x]

size

    ^count

top
    self isEmpty
        ifTrue:    [self exception: #isEmpty]
        ifFalse:   [^field at: output]

Storage    subclass:                    #Stack
           instanceVariableNames:       'stackpointer '
           classVariableNames: ''
           poolDictionaries: ''
```

Stack methods

```
    "code omitted for brevity "
```

Obviously, it would be nice to apply the technique just demonstrated to ReadStreams. The possibility of providing an exception handler for the exception "end-of-stream" avoids checking the end-of-stream every time a value is fetched. This could amount to considerable savings if the stream to be processed contains many items. Regrettably, it is not possible to redefine class Stream because there are always two instances of the subclass FileStream.



HANDLING ALLOCATION ERRORS

Allocation errors are special in that they cause an instance creation to fail. The exception handling must therefore be done by class methods. The essential problem is to catch the error communicated by either Behavior>>new or Behavior>>new: if the instance creation failed.

In the following example, the class method new:atError: is called with the handler. The handler is stored in the class variable ErrorBlock, where it can be fetched from the class method error:. The class method error: is sent by the calls Behavior>>new: and Object>>primitiveFailed.

```
Object variableSubclass:          #Matrix
       instanceVariableNames:     'numberOfRows numberOfColumns'
       classVariableNames:        'ErrorBlock '
       poolDictionaries: ''
```

Matrix Class Methods

```
new: aPoint atError: aBlock

    " Creation of a matrix with aPoint x rows and aPoint y columns.
      The block will be called if it is not possible  to create the instance.
      This is an allocation error. "

    | aMatrix |

    ErrorBlock := aBlock.
    aMatrix := super new: aPoint x * |aPoint y|
    aMatrix rows: aPoint x;
              columns: aPoint y.
    ErrorBlock := nil.
    ^aMatrix

error: aString

    " This method overrides the method in class Object. It will be called
      from Object>>primitiveFailed if the method Behavior|new:
      cannot allocate memory for a matrix of the specified size. "

    | handler |

    (handler := ErrorBlock) isNil
        ifTrue: [super error: aString]
        ifFalse: [ErrorBlock := nil.
                  handler value: self
                        value: #allocationError]
```

Matrix Methods

```
columns: anInteger
    numberOfColumns := anInteger

rows: anInteger
    numberOfRows := anInteger
```

It is an interesting exercise to reimplement the classes Stack and Queue as variable classes and to revise the exception handling in such a way that allocation failures can be handled.

Obviously, there are two problems remaining:

1. The methods described so far are not applicable to classes like Collection or Stream because there are always instances of some subclasses of these classes in the image.
2. It is not feasible to add an instance variable to every instance of a number.

In this situation, it is sometimes feasible to store the exception handler in a class variable. This will work in very much the same way as demonstrated in the matrix example

HANDLING ARITHMETIC EXCEPTIONS

```
Number    variableByteSubclass:    #Float class
          VariableNames:           'ErrorHandler '
          poolDictionaries: ' '
```

Float Class Methods

```
exceptionHandler

    ^ ErrorHandler

exceptionHandler: aHandler

    " aHandler is expected to be a block with two arguments. The first
      argument should take the sender (i.e., self); the second
      argument should take the message symbol. "

    ErrorHandler := aHandler

floatError
        "Query the floating-point coprocessor as to the type of
         exception and report it."
    | status message |
    status := self status.
    message := 'Float undefined exception'.
    (status bitAnd: 8) ~= 0
        ifTrue: [message := 'Float overflow exception'].
    (status bitAnd: 16r10) ~= 0
        ifTrue: [message := 'Float underflow exception'].
    (status bitAnd: 4) ~= 0
        ifTrue: [message := 'Float divide by zero exception'].
    (status bitAnd: 2) ~= 0
        ifTrue: [message := 'Float denormalized operand'].
    (status bitAnd: 1) ~= 0
        ifTrue: [message := 'Float invalid operation'].
    (status bitAnd: 16r80) ~= 0
        ifTrue: [message := 'Math coprocessor missing'].
    ErrorHandler notNil
        ifTrue: [^ErrorHandler value: self
                              value: message]
        ifFalse: [^ super error: message]
```

A MORE GENERAL APPROACH TO EXCEPTION HANDLING

Examining the handling mechanisms used in Ada,[1] one observes that exception handlers are a part of the activation record of the segment where they are defined. Administration of exception handlers can be done with very little overhead together with the administration of the activation records.

It is convenient to review the most important features of the exception concept used by Ada:

1. Exceptions are identified by name.
2. One or more exception handlers may be attached to a sequence of statements.
3. Exception handlers may be nested, and exceptions may be propagated from an inner handler to an outer one. Propagation follows the dynamic reference chain.
4. Handler nesting and exception propagation together permit a kind of distributed exception handling, where several handlers contribute to prepare the continuation of a program after the occurrence of an exception.
5. An exception is served by the innermost handler declared for that exception. The execution of the handler completes the execution of the sequence of statements it is attached to.

Our concept of exception handling is illustrated by the following similarity:

```
begin                                ExceptionManger new
    <statement sequence>                 exception: #<exception name>
    exception                            handler:
        when <exception name>                [:sender :handler |
            => <handler actions>                 <handler actions>];
end;                                     execute: [<statement sequence>]
```

The resulting value of the expression of the right-hand side is:

- The result of the block [*<statement sequence>*] if no exception was raised; or
- The result of the handler [:sender : handler | *<handler actions>*] if the exception *#<exception name>* was raised.

Earlier, we proposed handlers that contained a return statement, generally the statement ^self. The jump statement abandons the execution of a sequence of statements that, due to an exception, cannot be completed. As we will see, our implementation of the Ada-like exception handling is also based on return statements. However, these return statements are merely implementation details of class ExceptionManager. The programmer of an exception handler will not use any return

statements; in fact, he does not even need to know that jump instructions are part of the Smalltalk language.

Trying to utilize as much as possible from Ada and realizing the impossibility of modifying class Process, we decided to administer exception handlers in a separate stack to be held in synch with the stack of the current process.

The top element of this stack is referenced by the global variable CurrentHandler; the stack itself is implemented as a list of instances of class ExceptionManager. Each instance of class ExceptionManager administers a dictionary of specific handlers and, optionally, a catch-all handler.

Using a global list of handlers, this approach avoids the introduction of instance variables. The error block is now defined to take the following arguments:

```
<error block> value: <sender>
              value: <instance of ExceptionHandler>
```

The block can communicate with the instance of ExceptionManager. The following methods are provided:

- exception — inquire about the raised exception
- restart — restart the block associated with the handler

EXAMPLES OF USE

It is assumed that the calls of method #error: are replaced by calls of the method #exception: (described later in this paper).

```
"Read the contents of a stream until encountering end-of-stream. "

| x |
ExceptionManager new
    exception: #endOfStream
        handler:   [:sender :handler | self];
        execute:   [ x := ReadStream on: #(4 65 7 8 4 2).
                     [Transcript show: x next printString; cr.
                      true ]
                      whileTrue: []
                   ]

"Reprompt until you receive a usable answer. "

| x |
ExceptionManager new
    defaultHandler: [:sender :handler | handler restart];
    execute:   [x := Prompter prompt: 'value '
                              defaultExpression: '4'.
                1.0 / x]

"Issue a message and abandon the current statement. "

ExceptionManager new
    exception:    #invalidMessage
        handler:  [:sender :handler |
                   Transcript show: handler exception printString.];
    execute: [ String new add: 34 ]

"Reenter the current statement with a value to be used in the
 place of the result the statement failed to produce. "
(3 negated to: 3 by: 1) collect:
    [:i | ExceptionManager new
              exception: #zeroDivision
                  handler:   [:sender :handler | 0.0];
          execute: [ 1.0 / i]
    ]

"Propagation and restart used in conjunction: the outer block will
 be restarted as long as i is less than 4. "

| i |
i := 0.
ExceptionManager new
    exception: #test
        handler:   [:sender :handler |
                    Transcript  show: 'outer Handler entered ';
                                show: i printString; cr.
                    (i := i + 1) < 4
                        ifTrue:
                            [Transcript show: 'restart from outer block';
                                        show: i printString; cr.
                            handler restart
                            ]
                   ];
    execute: [Transcript show: 'enter outer block'; cr.
              ExceptionManager new
                  exception: #test
                      handler:
                          [:sender :handler |
                              Transcript show: 'propagate the
                                                exception'; cr.
                              sender exception: handler exception
                              ];
                  execute: [Transcript show: 'enter inner block'; cr.
                            self exception: #test
                           ]
             ]
```

The piece of code to be executed under the control of these handlers is a block. This block shall not contain any return statements (^*<expression>*). If it is necessary to leave a method from within an exception handler, one has to write something like this:

```
methodSelector
    ExceptionManager new
          exception: #exitMethod
          handler: [:s :h | <return value>];
    execute:
          [(ExceptionManager new
              exception: #myException
              handler: [:s :h | <actions>
                                self exception: #exitMethod];
              execute: <aBlock>
    ]
```

The method will be left by signaling an exception to the outermost exception manager of the method. This technique guarantees that the necessary reduction of the stack of exception managers will be done.

Class Object is augmented by the following instance method, which is used to raise an exception.

Object Methods

```
exception: aSymbol

    " Exception handling: first, a handler for the signaled exception is
      searched. If no such handler exists, it will try to find a handler
      for the exception #unhandledException. The method #error: will
      be called if there is no reasonable way to handle the exception. "

    | handlerToBeUsed |
    CurrentHandler isNil
        ifTrue: [self error: 'unhandled exception: ', aSymbol asString]
        ifFalse: [(handlerToBeUsed := CurrentHandler
                                        searchHandlerFor: aSymbol)
                notNil
            ifTrue: [^handlerToBeUsed from: self exception: aSymbol]
            ifFalse:
                [(handlerToBeUsed :=
                    CurrentHandler
                        searchHandlerFor: #unhandledException)
                    notNil
                        ifTrue:
                            [^handlerToBeUsed from: self
                                    exception: #unhandledException]
                        ifFalse:
                            [CurrentHandler := nil.
                            self error:
                                'unhandledException: ', aSymbol
                                                    asString]
                ].
            ]
```

The class ExceptionManager is defined as a subclass of class Dictionary. The inherited dictionary features are used to store exception handlers for named exceptions. Additional instance variables are used to store the catch-all handler, jump blocks, and some state informations. The instance variables are used as follows:

- cdr — a list of all previously created exception managers
- catchAllExceptions — a block that is used as a catch-all handler
- reenterBlock — a block implementing a jump into method **#doProtected:**
- exceptionSignaled — a boolean value indicating whether an exception was signaled
- exceptionName — name of the signaled exception of nil
- restartFlag — a boolean value indicating whether the exception handler requested the restart of the protected block
- restartBlock — a block implementing a jump into method **#execute:**
- sender — a reference to the instance that raised the exception

The method **execute: aBlock** creates two additional segments. The first segment created is that of method **doProtected:**. This segment is the reentry point for program continuation after an exception is raised. Furthermore, this segment provides a block for immediate return into method execute:, where the restart facility is implemented. The second segment created is method **execStatement:**. This method prepares a block for immediate return into the segment of method doProtected: and then executes the protected statement.

```
Dictionary subclass: #ExceptionManager
    instanceVariableNames: 'cdr
                            catchAllExceptions
                            reenterBlock
                            exceptionSignaled
                            exceptionName
                            restartFlag
                            restartBlock
                            sender '
classVariableNames: ''
poolDictionaries: ''
```

ExceptionManager Class Methods

```
accepts: aSymbol
    " Ask whether the current handler administrater provides a specific
      handler for the exception aSymbol or whether it is capable of
      handling all exceptions. "
    ^ CurrentHandler isNil
        ifTrue: [false]
        ifFalse: [CurrentHandler accepts: aSymbol]
```

ExceptionManager Methods

```
accepts: aSymbol
    " Ask whether this instance provides a specific handler for the
      exception aSymbol or whether it is capable of handling all
      exceptions. "

    ^ catchAllExceptions notNil
        or: [self includesKey: aSymbol]

cdr
    " List of previous exception managers."

    ^cdr

defaultHandler: aBlock
    " Define a handling block that will handle all exceptions that
      cannot be handled by specific handlers. "

    catchAllExceptions := aBlock

doProtected: aBlock
    " Provide the restartBlock and execute the protected block. "

    | result |

    restartBlock := [^self].
    result := self execStatement: aBlock.
    exceptionSignaled
        ifTrue:
            [CurrentHandler := cdr.
            result :=
                (self at: exceptionName
```

```
                ifAbsent: [catchAllExceptions])
                    value: sender
                    value: self.
            ^result]
        ifFalse: [^result].

exception
    " With this method, the handling block may ask for the name of
      the exception that it should handle. "

    ^exceptionName

exception: exception handler: aBlock
    " Store a specific handle block – aBlock. The value of parameter
      exception may be the name of an exception or a collection of
      exception names. "

    (exception isMemberOf: Symbol)
        ifTrue: [self at: exception put: aBlock]
        ifFalse:
            [exception do: [:aSymbol | self at: aSymbol
                                        put: aBlock]
            ]

execStatement: aBlock

    reenterBlock := [^ self].
    ^aBlock value

execute: aBlock
    " Execute the statements of aBlock under the protection of this
      handler and all handlers accessible via instance variable cd. "

    | result |

    cdr := CurrentHandler.
    CurrentHandler := self.
        [exceptionSignaled := restartFlag:= false.
        result := self doProtected: aBlock.

        restartFlag]
        whileTrue: [].

    CurrentHandler := cdr.
    ^result

from: sendingInstance exception: aSymbol
    " This method is send from Object>>exception. It handles a raised
      exception. The exceptionName is stored and method
      execStatement is left. The exception block itself will be called in
      method doProtected. "

    exceptionName := aSymbol.
    exceptionSignaled := true.
    sender := sendingInstance.
    reenterBlock value.   " return to method doProtected: "

restart

    " Restart the ProtecedBlock. The restartFlag is set and the
      restartBlock is evaluated. The CurrentHandler is reactivated. (It
      was deactivated by method execHandlerFor:sender:.) The restart
      itself occurs in method #execute. "

    restartFlag := true.
    CurrentHandler := self.
    restartBlock value.   " return to method execute: "

searchHandlerFor: aSymbol
    " Search for a handle block that is capable of handling the
      exception aSymbol. Search begins in this instance. It will be
      continued in the instance referenced by cdr if the instance self
      cannot provide an usable handler block. This method is called
      from Object>>exception:. "

    | ptr |

    ptr := self.
    [(ptr accepts: aSymbol)
        ifTrue: [^ptr]
        ifFalse: [ptr := ptr cdr].
    ptr notNil
    ] whileTrue: [].

    ^ nil " no handler "
```

This method can be made usable for processes. All that is necessary is a globally declared dictionary, where suspended processes can be stored together with their handler stacks. The process switching has to be modified in the following way.

Process Instance Method

```
    "This method is for Smalltalk/V 286 only."

resume
    "Resume the receiver process. Store the handler list of the current
     process "

    SuspendedHandlers at:     CurrentProcess
                      put:    CurrentHandler.
    CurrentHandler := SuspendedHandlers at: self
                                    ifAbsent: [nil].
    SuspendedHandlers removeKey: self
                      ifAbsent: [nil].
    CurrentProcess := self.
    self resume: 0
```

ProcessScheduler Instance Methods

```
    "This method is for Smalltalk/V 286 only."

initialize
    " Initialize the receiver by discarding all processes and then
      creating a new user interface process modified to implement
      exception handling. "

    Process enableInterrupts: false.
    readyProcesses := nil.
    readyProcesses := Array new: self topPriority.
    1 to: self topPriority do: [ :index |
        readyProcesses
            at: index
            put: OrderedCollection new].
    CurrentProcess := Process new.
```

```
" beginning of insertion "
CurrentHandler := nil.
SuspendedHandlers := Dictionary new.
" end of insertion "

CurrentProcess makeUserIF.
Terminal initialize.

Process enableInterrupts: true.
```

Using this method of exception handling, one has to accept one restriction: the exception handling mechanism cannot be used with the restart facility of the debugger. ✤

REFERENCES

[1] *The Programming Language*, Ada ANSI/MIL-STD-1815A, 1983.

[2] CCITT. *CHILL recommendation Z.200*, May 1984.

[3] ParcPlace Systems. *Objectworks for Smalltalk-80: Advanced User's Guide*, ParcPlace Systems, Mountain View, CA, 1989, Chapter 3.

[4] Goldberg, A. and D. Robson. *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

Boris Gärtner studied computer science, mathematics, and Bulgarian at a university in Munich. He has studied numerical programming, language implementation, and AI languages. Presently, he works on a development project in the field of database access. Since 1989, Smalltalk has become his favorite language. Boris can be reached at DATEX AI, Sandstrasse 41, D-8000 Muenchen 2, Germany (phone: (+8) 72 31 312).

Greg Hendley and Eric Smith

Giving application windows dialog box functionality in Smalltalk/V PM, part II

IN THE LAST ISSUE, we began the process of creating DialogApplicationWindow, a subclass of ApplicationWindow, with the essential behaviors unique to the class DialogBox. We identified two behaviors, modality and ownership. We investigated the definition of modality, how it is implemented in DialogBox, and how to implement it in DialogApplicationWindow, our own subclass of ApplicationWindow. In this issue, we will investigate the concept of ownership and then tie everything together.

OWNERSHIP

WHAT OWNERSHIP IS

Ownership is a window relationship supported by OS/2 Presentation Manager (PM). The ownership relationship is described on page 62 of the *Microsoft OS/2 Programmer's Reference, Vol. 2*. For PM windows in general there are no predefined rules for how the owner and owned windows are supposed to interact. PM does, however, provide a set of ownership properties that are specified for a frame window that is an owner of another window. All Smalltalk/V PM application windows have a PM frame window. The properties of frame window ownership are the remaining behaviors of DialogBox that we will add to our DialogApplicationWindow class. They are:

- The owned window always appears in front of the owner.
- The owned window closes when the owner closes.
- The owned window hides when the owner is minimized.
- The owned window shows when the owner is subsequently restored.
- The owned window normally moves when the owner moves. A constant relative position is maintained between the owned and owner windows.
- When either the owner or owned window is brought to front, both come to front.

At first glance, this behavior looks much like the parent-child window relationship. The difference is that child windows are always clipped by their parent. Owned windows may be completely separate, unless of course its owner also happens to be its parent.

HOW DIALOGBOX DOES IT

DialogBox supports frame window ownership by having OS/2 do the work. Part of opening a dialog is specifying the dialog's owner. The method used for opening dialogs in Smalltalk/V PM is the following:

```
fromModule: aModuleHandle id: anInteger
        "Open the dialog box whose id is anInteger
        contained in the module identified by aModuleHandle."
    owner isNil ifTrue: [
        owner := Notifier activeMainWindow.
        owner notNil ifTrue: [owner := owner frameWindow]].
    owner isNil ifTrue: [owner := WindowHandle queryActive].
    self handle: (WindowHandle fromBytes: (PMWindowLibrary
loadDlg: HwndDesktop
        owner: owner asParameter
        dlgProc: PM dlgProc
        hmod: aModuleHandle
        idDlg: anInteger
        createParams: nil)).
    handle = NullHandle ifTrue: [self class tooManyWindows].
    Notifier add: self.
```

The owner is set in two places: (1) in the first several lines and (2) in the message owner:dlgProc:hmod:idDlg:createParams: sent to HwndDesktop. You may recognize the first several lines as the same code we copied for use in the method findAndSetOwner for DialogApplicationWindow. These lines establish the owner in Smalltalk, but have nothing to do with PM ownership. PM ownership is established in the method sent to HwndDesktop. Here the PM owner is established as part of the PM call that creates the PM dialog.

ADDING PM OWNERSHIP TO APPLICATIONWINDOWS

SETTING THE PM OWNER

The first step in adding PM ownership functionality to ApplicationWindows is setting the PM owner of an application window. We can't just copy this code from DialogBox, so we need to do some searching. (The method in DialogBox that sets the PM owner won't work for application windows since it also creates the PM dialog, not a frame window.)

Page 356 of the *Microsoft OS/2 Programmer's Reference, Vol. 1* describes the PM function WinSetOwner, which tells one PM window to own another. The Smalltalk/V PM class PMWindowLibraryDLL provides the protocol for calling such PM window dynamic link library functions. In it, we find the method setOwner:owner:, which calls WinSetOwner.

Now we need a method in DialogApplicationWindow to make the owner be the PM owner. Setting a window's owner in Smalltalk/V PM has no effect on who the window's PM owner is. So, we need a method in the class DialogApplicationWindow for setting a window's PM owner to its Smalltalk owner. Note the check for an application window owning itself. If the application window owns itself, there is no need to set the PM owner. More importantly, the system will hang if you set a window's PM owner to itself.

```
makeOwnerPMOwner
    "Set my owner to be my PM owner so
    I can follow my owner around and stay above
    it just like a dialog would.
    Note: PM does not like my owning myself.
    GLH 5 July 1991."
    (self handle parentHandle == self owner handle)
        ifFalse: [PMWindowLibrary
            setOwner: self handle parentHandle
            owner: self owner handle].
```

THE OPENING METHODS

From the last issue, we have the method openModal. Now we will add two more methods: openOwned and openAsDialog. The first method is simply the code we used to test ownership:

```
openOwned
    "Open with the application window that
    opened me as my owner and PM owner."
    self
        findAndSetOwner;
        open;
        makeOwnerPMOwner.
```

Test the method by opening a workspace; this will be the application window for the dialog. From the workspace do:

```
DialogApplicationWindow new openOwned
```

Notice that the dialog remains above and follows the workspace. Go ahead and try minimizing, restoring, and closing the workspace.

The second method opens the DialogApplicationWindow as owned and modal to its owner:

```
openAsDialog
    "Open like a normal dialog. Open modal to
    the application window that opened me and
    be modal to my owner. GLH 5 July 1991."
    self
        findAndSetOwner;
        open;
        makeOwnerPMOwner;
        processInput.
```

The message order matters in these methods: findAndSetOwner must be sent before the message open. This is because findAndSetOwner makes the currently active window the owner. If the dialog has just opened, the dialog will be the currently active window. So, sending the findAndSetOwner after open would make the dialog be owned by itself instead of by the window that opened it.

The message makeOwnerPMOwner needs to be sent sometime after findAndSetOwner. Setting the PM owner to the Smalltalk owner makes sense only after the Smalltalk owner has been set. The method makeOwnerPMOwner also needs to be sent after the message open. The method makeOwnerPMOwner relies on both the dialog and the owner window having valid window handles. These are not valid until both the owner window and the dialog are open.

The message processInput has to be sent last since it blocks the method until the dialog is closed. An exception is when you want the dialog to return a value when it closes, as Prompter and MessageBox do.

AN EXAMPLE

The following is the code for a dialog built entirely within Smalltalk/V PM. The dialog contains the same controls used in a PM dialog. The dialog asks the user to choose between three buttons. The dialog returns the label of the button. This example shows ways to handle the problems in implementing a dialog.

First, define the class MyTestDialog. Give it an instance variable so it can remember the user selection for its answer:

```
DialogApplicationWindow subclass: #MyTestDialog
    instanceVariableNames: 'answer '
    classVariableNames: ''
    poolDictionaries: 'PMConstants '
```

The basic opening method is

```
openWithFirst: string1 second: string2 third: string3
        "Open and ask the user to choose
        between three labeled buttons.
        Answer the user's choice."
    self addSubpane: (Button new
        contents: string1;
        owner: self;
        when: #clicked perform: #getAnswerFromButton:;
        framingRatio: (0.1@0.1 corner: 0.3@0.4);
        yourself).
    self addSubpane: (Button new
        contents: string2;
        owner: self;
        when: #clicked perform: #getAnswerFromButton:;
        framingRatio: (0.4@0.1 corner: 0.6@0.4);
        yourself).
    self addSubpane: (Button new
        contents: string3;
        owner: self;
        when: #clicked perform: #getAnswerFromButton:;
        framingRatio: (0.7@0.1 corner: 0.9@0.4);
        yourself).
    self addSubpane: (StaticText centered
        contents: 'Pick a number';
        framingRatio:(0.1@0.5 corner: 0.9@0.9);
        yourself).
    self addSubpane: StaticBox new.
    self openAsDialog.
    ^answer.
```

The method getAnswerFromButton: is referenced in openWithFirst:second:third:, so we define it here:

```
getAnswerFromButton: aButton
        "The user has made a choice. Set the
        answer based on the label of the button
        and close."
    answer := aButton contents.
    self close.
```

Now the class is ready to be tried out; try it out by "showing:"

```
MyTestDialog new
    openWithFirst: 'one'
    second: 'two'
    third: 'three'.
```

There are several methods you may want to add to make instances of MyTestDialog look more like a normal dialog. To give the dialog a dialog border and get rid of unnecessary clutter, override the method defaultFrameStyle to answer only FcfDlgborder:

```
defaultFrameStyle
        "Private - Answer the default PM frame style for the receiver.
    ^FcfDlgborder
```

Override buildMenuBar to do nothing:

```
buildMenuBar
    "Don't build a menu bar for this dialog."
```

Override initSize to answer the size and location you want the dialog to open in:

```
initSize
        "Private - Answer default initial window extent."
```

> **"You now know how to make application windows modal to one another."**

```
    ^100@100 extent: 200@150!
```

You now have the basic functionality of dialog windows in a subclass of ApplicationWindow. You know what modality is. You now know how to make application windows modal to one another. You also know what PM ownership means and how to use it to make one window always appear above another. With these tools, you can make floating tool pallets. You can also group application windows so that, when you activate any one window, all windows in the group come to front. Most importantly, you can create custom dialogs without going outside of Smalltalk/V PM. ✣

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/V PM.

Eric Smith is a member of the technical staff at Knowledge Systems Corporation. His specialty is custom graphical user interfaces using Smalltalk (various dialects) and C.

They may be contacted at Knowledge Systems Corporation, 114 MacKenan Drive, Cary, NC 27511, or by phone at (919) 481-4000.

BOOK REVIEW

reviewed by Dan Lesage

PRACTICAL SMALLTALK

by Dan Shafer and Dean A. Ritz
Springer-Verlag, New York, 1991

YOU CAN'T JUDGE A BOOK by its cover. This is certainly true of *Practical Smalltalk* by Shafer and Ritz. This book contains useful technical information for novice Smalltalk/V 286 programmers. However, the practicality of the text is limited to single programmer development of applications and components within the V 286 environment. To more correctly reflect the book's contents, the title might have been something like "Writing Components and Applications in Smalltalk/V 286" or "How to Use the Model-Pane-Dispatcher Paradigm." Bookstore browsers might misinterpret the existing title and back cover notes as addressing practical design issues for large-scale applications running on a variety of Smalltalk implementations.

The authors state that the purpose of the text is to instruct the reader in using Smalltalk for real-world application development. This is done by leading the reader through a series of five small applications of increasing complexity, focusing on those aspects of the system that are deemed to be the important ones for application development.

The first two chapters of the book act as supplemental and review material to the Digitalk Smalltalk/V manuals. The supplemental material is of two types. The first type consists of "tricks" to get around problems that new Smalltalk users might have when interacting with the system. Most of the first chapter is devoted to the use of the debugger. The second type of supplemental material consists of class definitions that could be used as the V 286 class comments. These definitions consist of behavioral descriptions and indications of other classes that cooperate with the defined class. Chapter 2 can also be used as a secondary index for the text since each class description indicates the chapter in which the class is used in one of the example applications.

The next nine chapters serve to describe the five example applications. Each subsequent pair of chapters consists of a chapter that introduces the appropriate classes and methods to design the application and then a chapter that describes the application design in detail. The five applications consist of a browser that prompts the user to prioritize text entries, a counter widget, a multiselection list pane, a bar graph editor/displayer, and a fill-in-the-blank form widget. Each application highlights some aspect of Smalltalk/V 286. The first three applications focus on the model-pane-dispatcher (MPD) paradigm. The bar graph editor highlights the graphics capabilities. The form widget highlights the use of the text manipulation classes.

The code presented in the book is well written, and the text associated with the method selectors in the studied classes does provide greater insight than the Digitalk comments. The authors are careful to document the stumbling blocks that they ran into when learning Smalltalk, allowing readers to avoid some of the subtle pitfalls. The text is easy to read and is written in a down-to-earth manner. The code for the applications represents good Smalltalk style.

The text also contains a few shortcomings. The authors indicate that the described techniques can easily be used in Smalltalk dialects other than V 286. Unfortunately, V PM and Objectworks\Smalltalk have different paradigms for building applications. Much of the text is concerned with extending and overriding the appropriate methods pertaining to MPD realization of an application. This is great for V 286 programmers; however, it is not clear exactly which of the described techniques are portable to the other Smalltalk implementations.

Second, although I realize that it is very difficult to choose simple examples to demonstrate the system components from an application programmer's point of view, only the first and fourth application projects address issues that an application programmer might consider. The other three "applications" are really application components in the form of interaction widgets. Application programming in any language consists of coupling existing code while attempting to reuse as many existing components or functions as possible. The design of reusable components is quite different; the priorities are different. Typically, a component programmer is focused on the creation of good reusable components, while the application programmer is focused on solving the problem at hand given the tools available. The examples in the text present an opportunity to highlight the distinction between these two disciplines.

A third concern is the notion that practical applications are still created by a single programmer developing code in isolation. The book does not address the problems associated with sharing or distributing code. Once the sample applications are complete, how do I share them with my colleagues?

Practical Smalltalk could have increased its appeal to novice programmers by providing references and a bibliography. For example, Chapter 4 defines the term *object responsibility*. It

continued on page 23...

SOFTWARE REVIEW

reviewed by Cahan O'Ryan

Tigre: an interface builder for Objectworks\Smalltalk

THE TIGRE PROGRAMMING ENVIRONMENT is a graphical interface-building kit (Tigre Interface Designer) bundled with a persistent object storage mechanism (Tigris Database) that runs on top of ParcPlace's Objectworks\Smalltalk and, therefore, supports the "instant" portability of programs to a large list of hardware platforms without any changes in code. The purpose of this software system is to provide Objectworks\Smalltalk developers with the capability to quickly create and test the graphical user interface component of their Smalltalk applications. The Tigris database is tightly integrated with and inseparable from the interface designer since the interface designer uses the database to store the screen descriptions.

The intended audience for this software product is developers who may or may not already be using Objectworks\Smalltalk and are interested in speeding up their development cycle by using a tool that allows them to quickly create their interface screens visually. Developers, using the rich library of icons and color patterns that are provided in the Tigre system, also gain the advantage of being able to easily create screens that are visually stunning. Developers who are not currently using Objectworks\Smalltalk and have decided against it because of the lack of tools for visually building graphical interfaces should reconsider using Objectworks in conjunction with the Tigre Programming Environment. This product also allows novice Smalltalk programmers to create sophisticated graphical user interfaces much more quickly than if they had to learn all the ins and outs of the Smalltalk class library. Tigre makes it easier to reuse the code of more advanced Smalltalk programmers who create the plug-and-play interface elements known as *widgets*.

An interesting aspect of the Tigre system is that all the screens in the system used for the various utilities available were built with the Tigre system itself, providing a level of uniformity throughout the system. This means that any screen in the Tigre system can be easily modified or extended by the customer in the same way that the customer's own screens can be modified and extended.

The package comes with several utilities that help you organize and browse through your Tigre applications. There are also several sample applications and a tutorial to help you get up to speed in developing Smalltalk applications with Tigre.

TUTORIAL

The tutorial that comes with Tigre is an excellent example of how easy it is to put together a simple and colorful graphical interface. The tutorial, called Tidepool, is a browser for a database of tidepool creatures. The database contains a full-color picture as well as a textual description for each creature. The tutorial takes you step by step through creating your own tidepool application. The complete application only requires about fifteen minutes to create, although this does not count the time it took to put together the Tigris database of graphics and text.

The process starts by creating and opening a new screen using the Program Editor by selecting commands found in pop-up menus. When a new screen is created, you will usually create a new Smalltalk class to go with it, a subclass of ScreenAgent. An instance of ScreenAgent acts as the interface model for the Tigre screen, and all commands that take place in the Tigre screen are dispatched through the ScreenAgent. The user is prompted for the name of the ScreenAgent subclass, and then a blank window opens. By using a pop-up menu from this blank window, you put the screen in edit mode, clearly indicated by a change in the window label. Once you are in edit mode, you can then add widgets to the screen.

The Tidepool screen shown in Figure 1 contains a total of five widgets: an image, a selection list, two text widgets, and a button. Each widget is created via a pop-up menu command and then positioned by dragging and resizing with the mouse. Widgets are chosen from a list of types that are currently in the system. If a standard widget does not fit your needs, a Smalltalk developer can create custom widgets (see the discussion on custom widgets below). The widget attributes are then edited by opening a dialog, again from a pop-up menu command. The attributes include any text that is displayed, the font and style it should be displayed in, the foreground and background colors or patterns, the border style (embossed, raised, two-dimensional, or none), and/or the icon that is displayed.

More important are the method selectors assigned to a widget that determine exactly how that widget will interact with a ScreenAgent. For example, a selection list requires one selector that will be used to get the list of items to display and another selector that will be used to inform the ScreenAgent of a selection change caused by the user. It is up to the developer, then, to implement the Smalltalk methods that get invoked by these selectors. There are a handful of methods in-

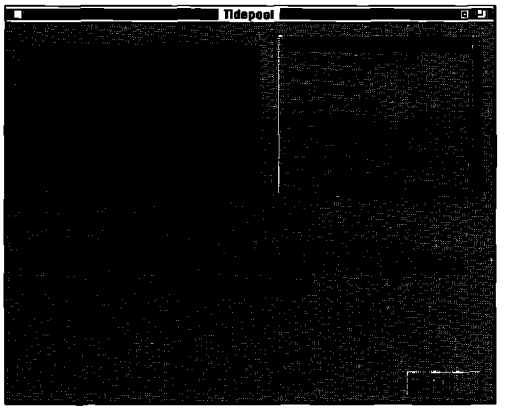


Figure 1. Tidepool.

herited from ScreenAgent that can be used without writing any additional Smalltalk code. For example, the method *accept* will close the window. Because the widgets interact with the ScreenAgent through method selectors rather than deferred evaluation code blocks, the widgets are better able to be reused later for other ScreenAgents that understand the same selectors.

Very conveniently, there is a pop-up menu item in all Tigre screens that allows you to open a code browser on the ScreenAgent for the window. This allows you to go back and forth quickly between working graphically with the screen and editing Smalltalk code, giving the developer a very tight development cycle. There is also a pop-up menu command that opens an inspector on the ScreenAgent or on any widget in the screen, providing quick access to debugging information.

Once the widgets are in place and their attributes set, it is a simple matter of changing the screen into user mode through another pop-up menu command to test the screen.

TIGRE LAUNCHER

The Tigre Launcher is a screen containing icons linked to the standard utilities that come with the package. There are also icons linked to the sample applications provided and space to add your own icons for other applications. Figure 2 shows the Tigre Launcher with some of the icons added that are included with the system.

PROGRAM EDITOR

The Program Editor is a launcher and browser for all the Tigre screens that exist in the system, organized by application and displayed in list format. The Program Editor is the tool used for creating, opening, and editing Tigre screens.

PALETTE

The Tigre system comes with a sample *palette*, shown in Figure 3, that has many sample widgets of various types. The standard widgets that come with Tigre include buttons, calendars (for viewing and setting dates), checkboxes, group boxes, lines, images, labels, pop-up lists, selection lists, switches, and text. Widgets can be "cloned" from this palette and placed into another screen. This supports easy reuse of widgets, and the developer can easily create his or her own palettes of often-used widgets. When copying a widget from one screen to another, the destination ScreenAgent may or may not understand the same method selectors that the original widget used. If it does, then the clones can be used instantly without any new code being written. If not, then the methods for the selectors must be implemented in the destination ScreenAgent.

TIGRIS DATABASE

The Tigris database system is a persistent object storage system for Objectworks\Smalltalk that is an enhancement of binary object streaming service (BOSS) technology provided by ParcPlace in their Objectkit for Smalltalk. The difference between standard BOSS and Tigris is that Tigris stores the objects by keys, allowing objects in a data file to be accessed like a Smalltalk dictionary using messages such as at:, at:put:, and removeKey:. This is much more convenient than a standard BOSS file, which stores the objects linearly. Tigris also includes a simple file-locking system that makes Tigris multiuser compatible with the help of networking systems such as NFS or AppleShare. The result is not a full-fledged object database management system (ODBMS) since the Tigris multiuser sys-

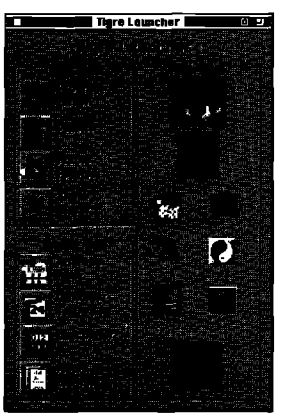


Figure 2. Tigre Launcher.

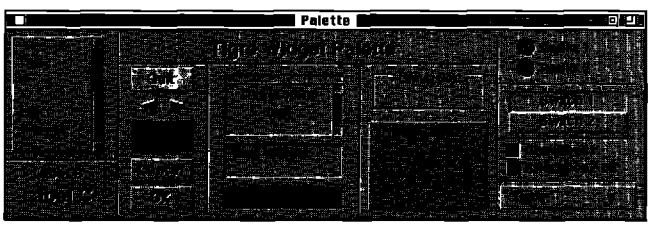


Figure 3. Tigre widget palette.

tem does not include the ability to send messages directly to persistent objects stored on a server. The objects must first be loaded into the Smalltalk system before a message can be sent to them. Tigre does have plans to release a Tigris server in the future.

A fundamental difference between the Tigre Programming Environment and other interface designers is that Tigre is not a code generator. The screens that Tigre creates are not translated into Smalltalk code that is later compiled and executed to open the screens. Rather, the screen descriptions are stored as binary objects in Tigris databases. This approach implies that the screens do not have to be algorithmically generated every time they are opened — they are simply loaded from the database. The quantity of source code in the final application is thereby drastically reduced, making the final application simpler to maintain. This design is similar to the way traditional Macintosh applications are built. On the Macintosh, the screen descriptions are stored in *resources* that are created graphically with a resource editing tool, and the resource manager is invoked to load the data at runtime. Tigre works in a very similar way, with the Tigris database system playing the role of the resource manager.

The Tigris database, however, has much more general use than the Macintosh resource manager. Tigris allows the storage of arbitrary objects keyed by strings. It also supports the storage of data in *frame* format, with multiple levels of keys. The first level is the primary key, one per frame. Each frame can have multiple slots, each keyed by a unique string. Each slot can be further keyed by multiple *facets*. A facet is one of potentially multiple values of a single slot. This framework allows the Tigris database a great deal of flexibility. One use of frames is to simulate a relational database, if desired.

Tigre also includes a frame browser and a frame mover. The Frame Browser allows the user to browse through the objects in Tigris databases. The Tigris Mover allows the user to move objects from one Tigris database to another. This tool can be used to share Tigre screens between applications and developers.

TIGRE AND MVC

The Tigre ScreenAgent is similar to a Smalltalk model in the model-view-controller (MVC) paradigm, except that a ScreenAgent has direct access to the user interface. In a typical Smalltalk application, one or more pure (noninterface) Smalltalk models could be connected to a Tigre screen by using the ScreenAgent as an access path between the models and the screen that interacts with the user. The ScreenAgent serves as a bottleneck for all message sending to and from a Tigre screen. This helps in the maintenance and analysis of the application code since there is only one place to look to find the code that gets executed when users interact with a screen.

The Tigre widgets, on the other hand, correspond to MVC view-controller pairs. The widget encapsulates both a view and a controller — the widget both displays itself to the user and interacts with the user through an input device.

SOFTWARE REUSABILITY

Tigre makes simple interfaces extremely easy to generate through the use and reuse of the standard widgets, which are easily cloned and adapted for applications that differ widely. Tigre also promotes reusability of new code that must written by allowing new interface devices to be implemented as custom widgets that can be used in a plug-and-play fashion. This will allow other users to easily take advantage of the effort of others in the area of user interface development.

CUSTOM WIDGETS

Since any set of standard widgets will never be able to handle all the needs for all projects, no interface designer would be complete without support for allowing developers to create their own custom widgets. The implementation of a Tigre widget is based closely enough on Smalltalk's MVC paradigm that existing Smalltalk interface code can be used as a foundation for a custom Tigre widget. Some additional code and modifications, however, will be required to fit the classes into the Tigre framework. An experienced Smalltalk programmer should have no trouble creating custom widgets and, by doing so, will make his or her interface elements much easier for others, perhaps less experienced Smalltalk programmers, to reuse.

OTHER FEATURES

Tigre also supports *modal* and *child* screens. A modal screen is one that takes control until the user accepts or cancels the screen. A child screen is one that is opened by the *parent* screen. If the parent screen is closed while the child screen is still open, then the child will be closed automatically.

Tigre also has support for a number of utility dialogs that are quite useful, including a File Chooser, a File Saver, various notifiers, a selector dialog, a prompter, and a confirmer. Some of these dialogs already have similar implementations in the standard Smalltalk system, but Tigre either extends their functionality or makes them easier to use. The File Saver and File Chooser, however, are not provided by Smalltalk and provide a user-friendly way of choosing and saving files in the underlying file system. These dialogs were much needed and are much appreciated.

SHORTCOMINGS

One disappointment in the Tigre system is that the ability to align widgets with one another on the screen is fairly cumbersome. Tigre provides pop-up menu commands for aligning one widget with another, matching two widget's edges, heights, and/or widths. There is no support for centering or aligning the centers of two widgets, aligning several widgets at once, or moving widgets in groups.

An issue addressed by other interface-building tools is how the widgets are affected when the window that contains them

is resized. Currently, Tigre widgets will always resize and position themselves proportional to the amount the window is resized. What is needed is the ability to specify whether the widget should resize with the window, have a fixed size and position, or some combination of the two. Tigre plans to incorporate more flexibility in this area in a future release.

Another potential problem for some developers is that not all the source code is provided with the software automatically — only most of it. There are five core classes in the system that the source code is not provided for unless a special source code license agreement is signed. The Tigre system can be used with the standard widgets provided without the source code license, but custom widgets cannot be created without it. This is only a minor inconvenience if you are willing to sign the extra license agreement.

PERFORMANCE

I have found the performance of the Tigre system to be quite good, provided I give it enough memory to work with. Eight megabytes of RAM are recommended for the system. One time I tried running a Smalltalk image with Tigre installed in "only" a 6,000-K memory partition on a Macintosh, and the system performed rather poorly. By increasing the partition size to 6,500 K, however, the speed-up was dramatic and the system performed reasonably well. One reason for the large memory requirement is the large number of 8-bit deep color images that are loaded into memory at one time.

RUNTIME LICENSE

A runtime license for the Tigre system requires that the location and size of widgets in a Tigre screen not be modifiable by the end user. The cost of such a license is negotiable with Tigre.

CONCLUSION

In my opinion, the Tigre Programming Environment is the one thing that Objectworks\Smalltalk most needed to compete with other interface development tools that are now on the market such as Smalltalk/V in conjunction with Acumen's WindowBuilder/V, Apple's MacApp with ViewEdit, or Neuron Data's Open Interface with OpenEdit. The capability of visually creating a graphical interface was an obvious missing feature in Objectworks until now. Objectworks\Smalltalk in conjunction with Tigre is easily the most interactive, richest, and portable development environment currently available that I've encountered. ✤

PRODUCT INFORMATION

TIGRE PROGRAMMING ENVIRONMENT
RETAIL PRICE: $2,900
SYSTEM REQUIREMENTS: OBJECTWORKS\SMALLTALK RELEASE 4.0, ANY PLATFORM THAT OBJECTWORKS RUNS ON, I.E., MACINTOSH, MS/WINDOWS, AND UNIX WITH THE X WINDOW SYSTEM
8 MB RAM

TIGRE OBJECT SYSTEMS
3004 MISSION ST.
SANTA CRUZ, CA 95060
(408) 427-4900
TIGRE!SUPPORT@UCSCC.UCSC.EDU

Cahan O'Ryan is the author of Arbor's two Object Bridge products. He's also a part-time graduate student at the University of Michigan, where he's working on projects involving GemStone. Cahan is a senior software engineer at Arbor Intelligent Systems, Inc. He can be reached there at 506 North State St., Ann Arbor, MI 48104, (313) 996-4238, or at oryan@eecs.umich.edu.

continued from page 19...

would have been natural for the authors to reference *Designing Object-Oriented Software*[1] by Wirfs-Brock and Wilkerson at this point to ease the reader into the realm of object-oriented design methods. Many of the problems with newcomers to Smalltalk (and OOP in general) consist of where to obtain reliable technical information. The book exemplifies techniques for good coding style issues in several places. Unfortunately, these are located in various parts in the text, rather than being consolidated for easy reference. The text also lacks a summary section in the last chapter.

Despite some of the above shortcomings, I recommend the book as an application sampler for novice Smalltalk/V 286 users who wish to become familiar with MPD. Since there now exist several paradigms for application development within various Smalltalk implementations, novice programmers should first understand one and then tackle the others. Although MPD is not the most elegant of these paradigms, it is the most natural for programmers who are already familiar with the basics of Smalltalk/V 286. In this regard, *Practical Smalltalk* can be used as the first step for leading new Smalltalk programmers into the realm of "real-world" application development. ✤

REFERENCE

[1] Wirfs-Brock, B. and B. Wilkerson. *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ, 1990.

Dan Lesage has been involved with object-oriented programming since 1986 and Smalltalk since 1988. Currently, he is the Project Manager, Turnkey Systems at Object Technology International in Ottawa, Canada. His current interests include distributed computing, data communications, and object-oriented analysis/design. He can be reached at Object Technology International, (613) 228-3535, or dan@oti.on.ca.

WHAT THEY'RE SAYING ABOUT SMALLTALK

Excerpts from industry publications

... We hope that in 2001, objects will be boring. In comparison, radical ideas of past decades — that system software should be written in higher-level languages or in languages with strong type systems, and that computers can and should be seamlessly networked — are thoroughly accepted today. Whether to implement them is almost never an issue now, even though there is still plenty of discussion about how to implement them well. In the same vein, we expect that 10 years from now, the object-oriented approach to software design and implementation will be an accepted, standard technique used in every language, library, database system, and operating system and will be taught in undergraduate computer science courses at every university. This is an issue of moving the technology further out into the world, and no major new thinking will be needed to accomplish it. One significant technological advance will be that we will free ourselves even further from equating objects with the nouns in the problem domain. Some of the most remarkable advances in the usability of computer systems have come from recognizing that processes, as well as things, can and should be described, modeled, and manipulated. Therefore, we will see software objects being used to model time, places, actions, and events. We believe that this will lead to usability advances almost as dramatic as those resulting from the now-established window/icon/mouse/pull-down interfaces that were to a large extent inspired by the original Smalltalk work of the 1970s and 1980s.

Smalltalk: Yesterday, Today, and Tomorrow,
L. Peter Deutsch and Adele Goldberg, **BYTE**, 8/91

... "Smalltalk lets me concentrate on solving higher-level problems; I don't have to fight the language," [Abdul Nabi of Knowledge Systems Corp.] says. "At the end of the day I've made progress toward building the client's application instead of just tracking down pointer errors." As a result, Nabi says, he's more productive than other consultants. "I can charge two to three times what other consultants charge ... because on an average day I'm five times more productive"...

... The availability of Smalltalk implementations on a variety of platforms is a key benefit for clients, Nabi says: "Cross-platform development has become a big issue for our clients. They give developers high-power workstations, but the applications they write have to run on inexpensive PCs. Because Smalltalk is highly portable — much more portable than C — this arrangement is feasible." More portable that C? Sure. Any nontrivial C applicationis based on assumptions about the environment where the code will run. Memory addresses, device naming conventions, and other details vary from platform to platform. At the very least, C applications must be recompiled when ported from one platform to another. In Smalltalk, hardware and operating-system dependencies are dealt with by the runtime system. Platform dependencies are encapsulated, hidden from programmers and applications. Programs run in a "virtual Smalltalk machine," and can therefore be ported from one operating system to another without recompilation ...

... There's a downside, too: Customers can be so pleased with working prototypes that they're sometimes reluctant to pay for further development. "Software isn't automatically good just because it's written in Smalltalk," Nabi admits. "Thrown-together prototypes often contain really terrible code, tossed in for demonstration purposes. It's important to write the production version of the software using stricter rules"...

A Competitive Edge, J.D. Hildebrand, **UNIX Review**, 7/91

... Most of the C developers we've talked to ... admit they're looking at both C++ and Smalltalk, but are waiting for more standardization of object classes. Some shops, because of existing code and skill sets, find it advantageous to use precompilers for COBOL, Pascal, and Fortran ...

Front-end Application Development Tools Come of Age,
Karen Watterson, **Data Based Advisor**, 8/91

... Arguably, the first major OOP language was Smalltalk, invented by Xerox at their Palo Alto Research Center (PARC) during the development of the workstation Dynabook, technology later drawn on by Steve Jobs for the Lisa and Macintosh. Based on the language Flex, it is similar in appearance to C and Pascal, but was created fresh by PARC. In it, everything is an object, unlike virtually all other OOP languages ...

Notes from Swan Lake: Software That Uses Object-Oriented Programming, Jason Goertz, **The HP Chronicle**, 8/91

... A handful of companies are now selling object-oriented languages that include much of the technology pioneered by the Smalltalk language originally developed at Xerox Corp.'s Palo Alto Research Center. Updated or written specifically for Windows, these languages are considered by some as the forerunners of tools to come for graphical user interface environments. For now, most of these tools present developers with a relatively steep learning curve, often requiring retraining and several months' work beforedevelopers can become productive ...

Choosing the Right Windows Tool,
Paul Pinella, **Datamation**, 8/1/91

... In many respects, Smalltalk is almost the opposite of C++: It is a pure OO environment. Even performing arithmetic is the manipulation of objects ... Smalltalk in its purest form is a self-contained environment, where the environment itself is constructed from objects that can be modified. In fact, Smalltalk belongs to a class of products known as OOPS (Object Oriented Programming Systems), along with others such as Actor ...

Mission Critical View: Object Orientation,
Martin Butler and Robin' Bloor, **DBMS**, 7/91

... The Tigre Interface Designer is a valuable extension to Smalltalk in and of itself. When combined with the second component, Tigris, the Tigre Programming Environment empowers the developer with the capabilities to create multi-user object-oriented database applications. Tigris implements a shared, distributed, persistent object store ... The Tigre Interface Designer provides a convenient means of defining connections between interface elements and Tigris persistent database elements. In brief, the Tigre Programming environment blows the

doors off the Hypercard/Oracle combo for multi-user database application development ...

Object-Oriented Programming: OOPSLA/ECOOP Reflections and New Products, Jim Salmons and Tim Lynn Babitsky, **MacTech Journal**, Spring 1991

... The Look and Feel Kit, with its so called wires, externalizes language-level objects. By grabbing a component, you immediately see its graphical elements. You also "see" its message capabilities — what are usually the conceptual, nongraphical aspects of an object. In a programming world where code is invoked through the passing of messages, the wires provide a literal, visual representation of the OOP procedural model. By drawing connections, you can create complex applications with a minimum of coding ... After the current Windows release of the Look and Feel Kit, Digitalk plans to ship an OS/2 version in October or November. This powerful, interesting development tool should add momentum to the OOP movement. And it may win some converts to the Smalltalk cause.

OOP Made Visual: Digitalk's Look and Feel Kit, Ellen Ullman, **BYTE**, 8/91

... Both Windows versions of Smalltalk maintain a text log of changes to the Smalltalk "image" (i.e., the Smalltalk gestalt of any moment). You can view the Smalltalk/V version of the log with the File utilities. With Objectworks/Smalltalk, you can view the change log as an object with a hierarchy that has separate instances for changes to classes, to methods, and to the system. Both products provide a method for applying the changes of one project to another, a necessary operation if the system is to follow the objective of reusability. Both products also have an excellent debugger, as well as tools for file management, view management, and text management. As with all things, their styles differ: Objectworks maintains its own style, and Digitalk adopts the style of Windows.

Smalltalk About Windows, Ben Smith, **BYTE**, 8/91

Servio Corp., developer of the GemStone object database system, has just announced object database access for Macintosh C applications with support of Symantec Corp.'s THINK C software development environment. Servio says new and existing Mac client applications written in Symantec's THINK C can now access objects stored in the GemStone object server. Think C is Symantec's C programming language for the Mac, implemented with object-oriented extensions. GemStone is a client/server object database management system with interfaces to support Smalltalk, C and C++ applications. Access to GemStone objects facilitates the development of more advanced applications, including multimedia, complex modeling, and arrays ...

Software Industry Report, 6/17/91

Sybase Inc. is moving on several fronts to strengthen its technology base as the company continues a two-year metamorphosis from database vendor to broad-based tools and services provider. The company has been workng with a London-based tools vendor to improve support for graphical environments from within Sybase's APT Workbench development software, a key feature demanded by the user community. Also, Sybase is looking into supporting the Smalltalk object-oriented language as part of a forth-coming repository-based development environment, code-named Comet .... The moves, which follow the acquisition this spring of computer-aided software engineering tools developer Deft Inc., come as Sybase is preparing for an initial public offering to begin as early as August, according to sources in the financial community ...

Sybase Adds Database Tools, Joshua Greenbaum, **Computer Systems News**, 6/17/91

PRODUCT ANNOUNCEMENTS

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

Servio announces the first commercially available Kanji object database

Servio Corporation has announced the shipment of a Kanji object database management system (ODBMS). Servio's GemStone now supports manipulation of extended UNIX code (EUC) standard Japanese character strings. Kanji support is immediately available in Japan and will be made available worldwide this fall. The initial Kanji release supports the Japanese Industrial Standards (JIS) character set, using the EUC representation. GemStone's Kanji capabilities provide support for storage, retrieval, indexing, concatenation, and all other functions normally associated with ASCII string manipulation. By supporting full indexing of Kanji character strings, GemStone assures high performance for applications handling complex multimedia data including Kanji text. In addition to Kanji capabilities, Servio will Japanize GemStone documentation, the developer interface, and other product features.

For more information, contact Servio Corporation, 1420 Harbor Bay Pkwy, Alameda, CA 94501, (415) 748-6200, or fax (415) 748-6227.

Digitalk named member of the IBM International Alliance for AD/Cycle

IBM announced that Digitalk, Inc., has become a member of the IBM International Alliance for AD/Cycle. The move underscored IBM's commitment to Smalltalk/V with AD/Cycle and is seen as providing key tools for delivering Common User Access (CUA '91) compliant applications within AD/Cycle. The Smalltalk/V products are used to complement host applications by producing System Application Architecture (SAA) and CUA compliant graphical interfaces for cooperative applications.

For more information, contact Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045, (213) 645-1082, or fax (213) 645-1306.

Digitalk announces Smalltalk/V PM release 1.3 and the Database Interface for Smalltalk/V PM

Digitalk's Smalltalk/V PM release 1.3 fully supports the new Common User Access architecture, known as CUA '91, which includes the new advanced controls IBM intends to ship with OS/2 2.0. The Database Interface provides simplified access to IBM's OS/2 Extended Edition Database Manager and the Microsoft SQL Server.

For more information, contact Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045, (213) 645-1082, or fax (213) 645-1306.

Digitalk and IBM sign letter of intent to market Smalltalk/V products worldwide

IBM and Digitalk have worked closely together to focus Digitalk's enhancement of Smalltalk/V PM. Once the proposed marketing agreement is signed, Digitalk products will reach a broader audience of corporate developers who need high-leverage tools to develop applications under OS/2 and Windows. Under the proposed agreement, IBM can market Smalltalk/V PM, Smalltalk/V Windows, Smalltalk/V DOS, and Smalltalk/V 286.

For more information, contact Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045, (213) 645-1082, or fax (213) 645-1306

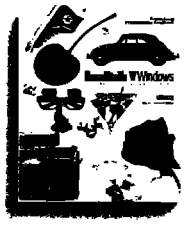
ParcPlace Systems' Objectworks\Smalltalk will support Information Builders' new Enterprise Data Access/SQL product family

ParcPlace Systems announced that it will support Information Builders' (IBI) Enterprise Data Access/SQL product family. EDA/SQL provides direct access to information in corporate databases, including IBM's DB2 and IMS, Sybase, Oracle, Informix, and IBI.

ParcPlace Systems intends to extend its suite of Objectworks\Smalltalk Portable Objects (object-oriented class libraries) to provide a common interface for applications that use IBI's EDA/SQL product. By using IBI's EDA/SQL product to support the common interface, Smalltalk developers will gain easy access to numerous corporate databases from within the Smalltalk environment. In addition, ParcPlace is working with its partners in the Smalltalk community to provide support for additional products and tools using this interface.

ParcPlace also announced that they have signed a strategic marketing partner agreement with IBI. ParcPlace will use EDA/SQL as its primary database connectivity solution, while IBI has agreed to support Objectworks\Smalltalk in its marketing efforts.

For more information, contact ParcPlace Systems, 1550 Plymouth St., Mountain View, CA 94043, (415) 691-6700, or fax (415) 691-6715.



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

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

With Smalltalk/V, you don't.

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

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

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

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

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

Smalltalk/V

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

LOOK WHO'S TALKING

HEWLETT-PACKARD

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

NCR

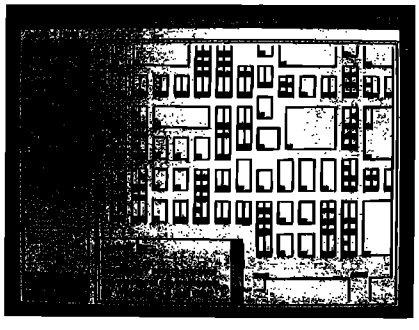
NCR has an integrated test program development environment for digital, analog and mixed mode printed circuit board testing.

MIDLAND BANK

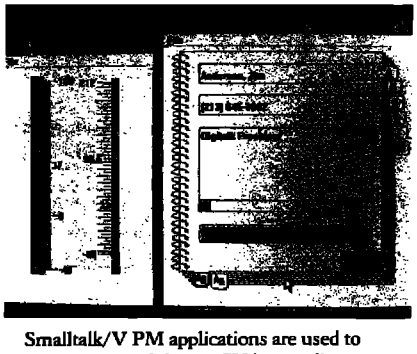
Midland Bank built a Windowed Technical Trading Environment for currency, futures and stock traders using Smalltalk V.

KEY FEATURES

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



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



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

Putting Smalltalk To Work!

1980	Smalltalk Leaves The Lab.	We were there.
1984	First Commercial Version Of Smalltalk.	We were there.
1985	First Industrial Quality Smalltalk Training Course.	We were there.
1987	First Fully Integrated Color Smalltalk System.	We were there.
1988	Responsibility-Driven Design Approach Developed.	We were there.
1991	Smalltalk Mainstreamed in Fortune 100 Applications.	WE ARE THERE.

Smalltalk Technology Adoption Services

Technology Fit Assessment
Expert Technical Consulting
Object-Oriented System Design/Review
Proof-of-Concept Prototypes
Custom Engineering Services & Support

Smalltalk Training & Team Building

Smalltalk Programming Classes:

Objectworks Smalltalk Release 4
Smalltalk V/Windows V/PM V/Mac
Building Applications Using Smalltalk

Object-Oriented Design Classes:

Designing Object-Oriented Software: An Introduction
Designing Object-Oriented Systems Using Smalltalk

Mentoring:

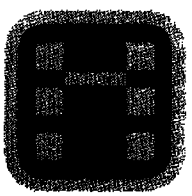
Project-focused team and individual learning experiences.

Smalltalk Development Tools

Application Organizer Plus™ Code Modularity & Version Management Tools

See our new Multi-User/Shared Repository Team Tools At OOPSLA 91!

Smalltalk! Nobody Does It Better.



Instantiations, Inc.

1.800.888.6892