The Smalltalk Report

The International Newsletter for Smalltalk Programmers

June 1994 **Volume 3 Number 8**

A BRIEF LOOK AT INHERITANCE METRICS

by Mark Lorenz

In my book OBJECT-ORIENTED SOFTWARE METRICS,[1] I divide static metrics into two categories:

- *Project metrics.* A group of metrics that deals with the dynamics of a project. Used for estimating work effort and progress.
- *Design metrics.* A group of metrics that looks at the quality of the project's design at a particular point in the development cycle.

As we all know, there are fundamental concepts underlying O-O software systems, including the use of inheritance. These differences from function-oriented development result in the necessity for a different set of metrics to measure the quality of designs. In this article, I'm going to discuss a couple of *design* metrics dealing with the use of inheritance. In upcoming articles, I'll take a look at some other O-O metrics.

INHERITANCE HIERARCHY NESTING

The deeper a class is nested in the inheritance hierarchy, the more methods there are available to the class and the more chances for method overrides or extensions. This all results in greater difficulty in testing a class.

Our experience has been that large nesting numbers indicate a design problem, where developers are overly zealous in finding and creating objects. This will usually result in subclasses that are not specializations of all the superclasses. A subclass should ideally extend the functionality of the superclasses. If you look around your everyday life, you see that specialization goes only so far. For example, if you look at a transportation domain, you might find a hierarchy similar to Figure 1.

Figure 2 shows some project results for nesting levels. Our rule of thumb is six levels as a threshold for identifying possible anomalies. Frameworks are a significant exception to this heuristic.

As we see in Figure 3, someone using the *View* framework will start at a nesting level of three. This will affect the maximum nesting level for the domain classes. In this case, we offset the heuristic, counting the nesting levels from the bottom of the framework instead of the top of the hierarchy.

Action plans

So what do you do if your nesting levels are beyond the rule of thumb threshold? First of all, the threshold is a heuristic, so it is possible that nothing is wrong in a particular case. There may be good reasons why you see classes nested nine-levels deep, for example. The point is to make a conscious decision about the anomaly rather than to ignore it. Assuming you decide that an action would improve your design, you can:

continued on page 4

Contents:

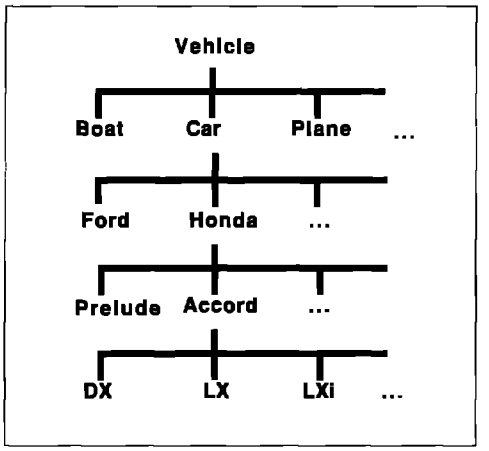


Figure 1. Example transportation domain hierarchy.

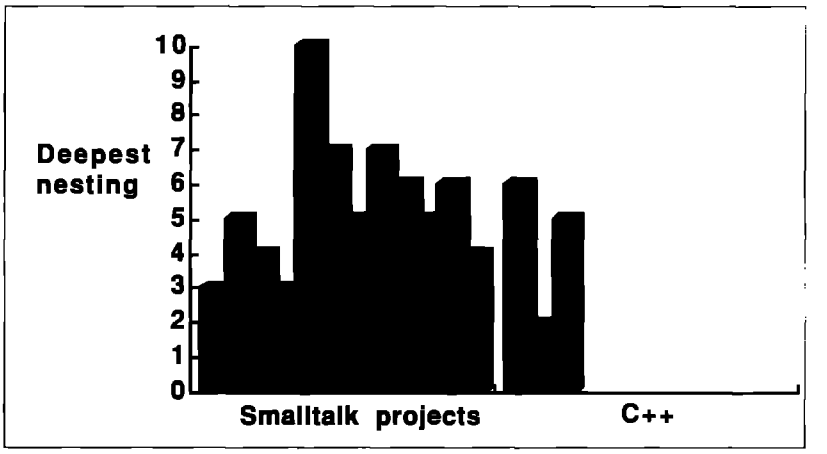


Figure 2. Project maximum nesting levels.

- Move subclasses that are not specializations of all their superclasses to another location in the hierarchy.
- Merge subclasses and superclasses that were created for expediency during rapid prototyping but that logically belong together.
- Consider whether a subclass should be a peer of its superclass.
- Factor out new classes that should have reused base classes, such as String or Stream.

METHOD OVERRIDES

A large number of overridden methods indicates a design problem. Because a subclass should be a specialization of its superclasses, it should primarily extend the services of the superclasses. This should result in unique new method names. Numerous overrides indicate subclassing for the convenience of reusing some code and/or instance variables when the new subclass is not purely a specialized type of its superclasses.

Figure 4 shows some project averages for method overrides. Our rule of thumb anomaly threshold is three method overrides by a class. We also weight this threshold by the *inheritance hierarchy nesting* measurement, so that more deeply nested subclasses have lower thresholds.

There are a number of affecting factors when considering the meaning behind overridden methods:

- *Framework.* Classes that are a part of a framework provide some functionality that is meant to be finished by application developers. One mechanism is used to define some methods that are *meant* to be overridden. Creating a method of the name defined by the framework architecture is not really an override—it is merely filling in the blanks and should not be considered for this measurement.
- *Abstract class.* Abstract classes often act as miniframeworks, providing method templates to be filled in by subclasses. Selectors such as implementedBySubclass and subclassResponsibility are used to note where subclasses are supposed to override methods in the abstract class. Again, these should not be included in a measurement of method overrides.
- *Invocation of superclass method.* Methods in subclasses can include the statement

```
super <methodName>
```

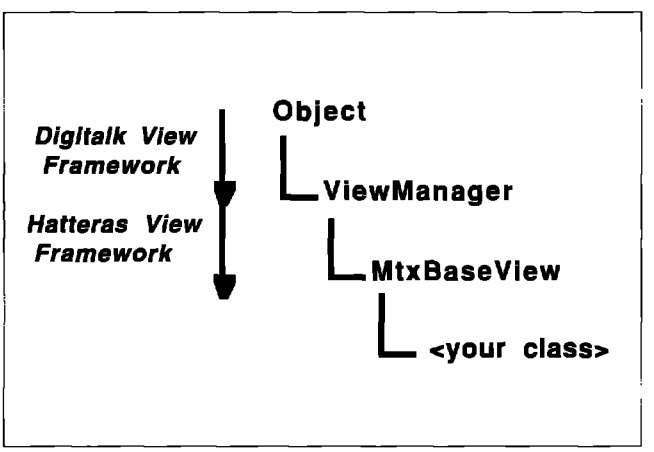


Figure 3. Framework example.

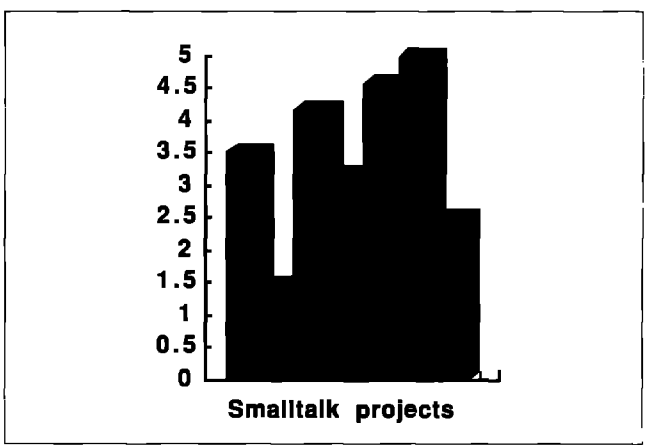


Figure 4. Average number of methods overridden per class.

GLOSSARY

Framework. A set of prebuilt classes and methods that define the basic structure of some end-user functions, leaving the application-specific details to be filled in by developers.

Heuristic. A guideline based on trial-and-error usage. A rule of thumb.

Measurement. The determination of the value of a metric for a particular object.

Metric. A standard of measurement. Used to judge the attributes of something being measured, such as quality or complexity, in an unbiased manner.

Specialization. An extension of the behavior of a type of object.

Threshold value. A measurement value that has been determined through project experience to be significant in terms of desirable or undesirable design, with some margin of error. Generally, these will be valuable over time to your gain or perform specific to your business and teams.

where <methodName> is the selector of the method being overridden. If this statement is always executed, then the subclass is specializing the behavior. This is not the same as a complete override, and should be weighted differently in measuring the method override metric.

Action plans

So what do you do if you have a class with a large number of method overrides (that weren't planned to be overridden)? Again, treat the threshold as a heuristic and make a conscious decision about the anomaly. A suggested action plan: Move the ill-fitting subclass to another place in the hierarchy. Look for a superclass in which the subclass is the same kind of thing. This means the expected behaviors should be similar, with few needed overrides. If you don't find such a superclass, move the former subclass under Object.

SUMMARY

We have taken a brief look at O-O metrics dealing with the use of inheritance. In particular, we examined a class' position in the inheritance hierarchy and a method's use of overrides. We have seen that there are factors that affect the meaning of these measurements and have taken a look at possible action plans for anomalies. ■

References

1. Lorenz, M. Object-Oriented Software Metrics: A Practical Guide, Prentice-Hall, Englewood Cliffs, NJ, 1994.

Mark Lorenz is founder and president of Hatteras Software, Inc., a company that specializes in helping other companies use object technology effectively. He welcomes questions and comments via email at 71214.3120@compuserve.com or voicemail at 919.851.0993.

VisualWorks List Components

William Kohl & Tim Howard

List components provide a flexible means for displaying a collection of objects and allowing the user to make a selection. What many developers may not realize, however, is that there are a variety of ways in which a list component can display the objects in this collection. In this article, we will discuss the collection on which a list component operates, and we will offer several techniques for displaying that collection's elements in the list.

A list component displays a collection and allows the user to select elements within the collection. For a single selection list, the aspect model is a SelectionInList, and the widget is a SequenceView. For multiple selections, the aspect is a MultiSelectionInList and the widget is a MultiSelectionSequenceView. The examples in this article concern a single selection list component, but all the material applies to multiple selection list components as well.

THE COLLECTION

The SelectionInList will only operate on sequenceable collections. This is because a SelectionInList tracks the current selection by its index—not by referencing the object directly. Sequenceable type collections include: OrderCollection, List, SortedCollection, and Array among others. Nonsequenceable collection types include: Set, Bag, and Dictionary among others.

It is important to remember that the objects in the collection can be of any type, and we are not restricted to just using textual type objects such as String and Text. Every object knows how to represent itself textually, whether it is a bitmap, a geometric, or a *VisualWorks* tool, because all objects understand the message printString. Any type of object can go into a list component's collection. Furthermore, because Smalltalk is typeless, we can have a heterogeneity of objects in the same collection, and the list component will perform admirably. In the example to be used throughout this article, we will create a list component to display a collection of all current instances of ApplicationWindow. To do this, paint a canvas with a single list whose aspect and ID are both #windows. Install this canvas as the application model ListExamples:

```
ApplicationModel subclass: #ListExample
    instanceVariableNames: 'windows'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'UIApplications-New'
```

Edit the windows aspect method to look like the following:

```
windows

    windows isNil
        ifTrue: [windows := SelectionInList with: ApplicationWindow
            allInstances]
        ifFalse: [windows]
```

Now open the application and your list will display 'an ApplicationWindow' for each of the instances of ApplicationWindow currently in your image (note that browsers, inspectors, workspaces, and debuggers do not live in ApplicationWindows, so don't expect to find them in your list!).

You should be aware that the collection that is displayed by a list component incurs a dependent. The collection contained by the SelectionInList is also known to the widget; this is a SequenceView. The SequenceView registers itself as a dependent of the collection. The reason for this is that if the collection object is a List, it can notify the SequenceView of a change of its contents. However, even if the collection object is not a List, the dependency is established. In such a case, a dependency is established on an object that is ill equipped to behave as a model. This dependency can cause problems when trying to make persistent a collection that is currently being displayed in a list component.

LIST CLASS

The List class was created specifically for operating in tandem with a SelectionInList. The List object behaves as a model. Whenever an element is added or removed, it notifies its dependents of this change. Because the SequenceView is a dependent, it redraws itself to reflect this change in the collection. This is not the case for other sequenceable collection types such as OrderedCollection and SortedCollection. Although these other collection types do acquire the SequenceView as a dependent, their behavior definition does not include model-type behavior, so they are ill equipped to know about internal changes and broadcast updates. Any changes to these collections will not automatically update the list component. In such cases, the burden is on the developer to send invalidate to the list component. This will cause the list widget to redraw itself based on the new information in the collection maintained by the SelectionInList.

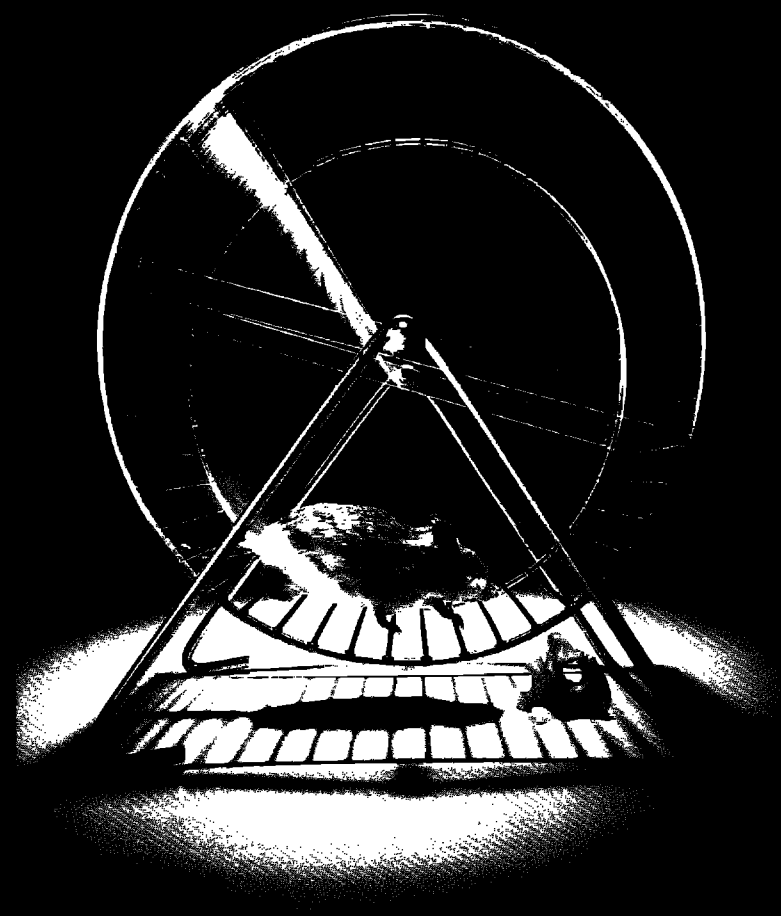
PRINT STRING METHOD

There are several ways to display items in a list. They can be represented as strings, formatted text, or as visual components. Furthermore, each object can have a different appearance for different list components. The remainder of this article explores these many possibilities.

By default, the SequenceView displays each object in the collection by sending it the message displayString. All objects know how to respond to this message, and the default behavior is to return the printString value. Every object in Smalltalk also knows how to respond to printString. For most objects, printString returns a string describing the object's type, such as 'a CompositePart' or 'an ApplicationWindow'. The printOn: method is the method actually responsible for constructing the string returned by printString. If you want to change the way an object displays itself as a string, edit the printOn: aStream

ParcPlace International
Users Conference. July 31-Aug 2
Santa Clara, CA. Call for Details 1-800-543-8096

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



VisualWorks

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

ParcPlace®

method. As an example, to make our list of windows appear more informative, add the following instance method to ApplicationWindow under the printing protocol:

```
printOn: aStream
    aStream
        nextPutAll: self label;
        nextPutAll: ' at ';
        nextPutAll: self globalOrigin printString
```

Now open ListExample again and you will see a much more informative list of your windows. We strongly recommend that you define a printOn: method for each kind of object you create. Even if a particular kind of object never appears in a list component, the dividends will pay off enormously in debugging and inspecting. It is the printString message that displays an object in a debugger or inspector, and having an informative printString can often mitigate the tedium of using these tools.

DISPLAY STRING METHOD

You can also specify how an object will appear in a list component by implementing the displayString method directly. This allows an object to be displayed one way in general (printString) and another way in a list component (displayString). For ApplicationWindow, add the following instance method under the printing protocol:

```
displayString
    ^'ApplicationWindow labeled: ' , self label
```

Now open ListExample to see that the list component is using the displayString message in favor of the printString message.

ARBITRARY DISPLAY STRING SELECTOR

To add some flexibility, you have the option of telling the SequenceView which selector to use to retrieve a printable representation of the objects in its collection—displayString is merely the default. Do this by sending the SequenceView the displayStringSelector: aSymbol message as part of the post build operation. The argument aSymbol is a message selector understood by each of the objects in the collection. To illustrate this, first add the following instance method to ApplicationWindow in the printing protocol:

```
displayOpenStatus
    ^self label , ' is ' , (self isOpen ifTrue:['open'] ifFalse: ['closed'])
```

Now add the following method to ListExample in the interface opening protocol:

```
postBuildWith: aBuilder
    (aBuilder componentAt: #windows) widget
        displayStringSelector: #displayOpenStatus
```

Open ListExample and the windows will be displayed in the list using the displayOpenStatus method.

FORMATTED TEXT

The display string selector does not necessarily have to return a string. It can also return formatted text, or a Text object. This allows you to have certain items in the list appear italicized, bold, in color, or even in a different character size or font. In this next example, we only display the window's label, but all closed windows will appear in normal text, collapsed (iconized) windows will be italicized, and expanded windows will appear bold. Add the following instance method to ApplicationWindow in the printing protocol:

```
displayLabelAndStatus
    self isOpen ifFalse: [^self label asText].
    ^self isCollapsed
        ifTrue: [self label asText emphasizeAllWith: #italic]
        ifFalse: [self label asText allBold]
```

and edit the postBuildWith: method in ListExample to read:

```
postBuildWith: aBuilder
    (aBuilder componentAt: #windows) widget
        displayStringSelector: #displayLabelAndStatus
```

Open ListExample to verify these changes.

GRAPHICAL REPRESENTATION

The objects in a list component do not even have to display themselves in a textual manner at all. The objects can be displayed visually, although this does require a certain amount of setup. Visual display requires defining the SequenceView's two visual blocks: visualBlock and selectedVisualBlock. These blocks determine how the items in the list are represented visually (including both text and graphics).

Each block takes two arguments: the SeqeunceView itself and the index of the selection currently being drawn. Both blocks should evaluate to a visual component. The blocks are set by sending the messages visualBlock: and selectedVisualBlock: to the SequenceView. This procedure should be performed in a post build operation. In addition, the visual blocks can be set at any time during runtime, and they will take effect immediately upon the next display of the widget. For our example, we will have our list component display the label of each window and prefix the label with the window's icon. First, we must add the instance method below to ApplicationWindow (put it in the printing protocol, although it does not really belong there):

```
icon
    ^icon mask
```

Now we will install the appropriate visual blocks to achieve the desired look. Edit the postBuildWith: method in ListExample to read:

```
postBuildWith: aBuilder
    | sequenceView |
    sequenceView := (aBuilder componentAt: #windows) widget.
    sequenceView visualBlock:
        [:sv :i | | window labelAndIcon icon |
        window := sv sequence at: i.
        icon := window icon.
        labelAndIcon := (LabelAndIcon with: window label offset: 4@0)
          icon: icon.
        BoundedWrapper on: labelAndIcon].
    sequenceView selectedVisualBlock:
        [:sv :i | | reversingWrapper window labelAndIcon icon |
        window := sv sequence at: i.
        icon := window icon.
```

continued on page 12

Alan Knight

Still more frequently asked questions

Commercial use of Smalltalk continues to increase, resulting in a constant stream of new users with questions about Smalltalk. Although there are many new user questions on comp.lang.smalltalk, this is only a small fraction of the people discovering Smalltalk. Many of them don't have a good way to get accurate information about Smalltalk. They are left with rumors, misinformation, overhyped marketing literature, and salespeople (sometimes the salespeople are disguised as consultants).

In this column, I attempt to provide some simple answers, free of propaganda, to some of the most frequently asked questions. These are a lot less technical than the questions normally discussed in this column, and I hope that the answers are more or less what any knowledgeable Smalltalk person would say. One word of warning: I've not had much opportunity to work with Enfin or Smalltalk/X yet, so it's possible I have inadvertently failed to mention some of their capabilities.

Are people using Smalltalk for real projects?
Yes. The primary uses of Smalltalk used to be in academic or research projects, but applications have been changing very rapidly. Smalltalk is now used extensively in financial and MIS applications, particularly in updating or providing graphical interfaces to legacy systems. Beyond that, it's possible to find Smalltalk projects in almost any application area. In fact, the use of Smalltalk is increasing so rapidly and the commercial opportunities are so numerous that many research projects have trouble keeping their Smalltalk programmers.

Can I interface Smalltalk to my relational database?
Yes. Most of the legacy systems mentioned previously have a database component, and a good interface from Smalltalk to the database is essential. Many different interfaces are available. Some Smalltalks come with database access built in or available as an option. There are also various products available from third parties.

It's important to note that the biggest difficulty is not connecting to the database but overcoming the impedance mismatch between the relational model and the object model. A naive interface can have much worse performance than either the relational or object models separately, so the interface should be carefully thought out.

Can I interface Smalltalk to code written in other languages?
The answer is a qualified yes: it depends on the language.

All implementations provide mechanisms to call C. Some of these read C header files and automatically generate Smalltalk classes and methods. If the other language (e.g., FORTRAN) can be called from C, then it should be possible to call a C routine, which in turn calls the other language. MS-Windows and OS/2 implementations usually support calling code stored in DLL's, which could be written in language that can produce a DLL.

Some implementations are starting to support languages other than C. IBM's VisualAge can call COBOL as well as C. Digitalk's PARTS can create wrappers for other languages, including COBOL. That isn't quite the same as being able to call COBOL from Smalltalk, but should be close enough for most purposes.

It would be particularly nice to be able to call other object-oriented languages from Smalltalk. Most O-O languages support a C interface, but that only allows calling C functions, not sending messages to objects. This situation should improve soon. Most Smalltalk vendors have announced their intention to support one or more of the emerging interlanguage object communication standards (CORBA, SOM, OLE, etc.).

These mechanisms should also allow Smalltalk to be called from other languages. Current Smalltalk implementations like to be in charge and only support calling back to Smalltalk when Smalltalk is the initiator of the computation.

What tools do I need?
Smalltalk already comes with a complete programming environment, eliminating the need for many of the traditional development tools. This doesn't mean that tools aren't necessary, and there are two categories that are particularly common:

1. ***Window layout.*** Even in Smalltalk, laying out Windows manually is tedious, error prone, and unnecessary. Tools for window layout and (more or less) visual programming have been available for some time now. Some, like Easel's Enfin, IBM's VisualAge, and ParcPlace's VisualWorks are more or less bundled with Smalltalk. Smalltalk/V can be used with both Digitalk's own PARTS product and Objectshare's WindowBuilder.
2. ***Team programming.*** Smalltalk was originally designed as a single-user single-machine development environment. A

number of additional tools are available for dealing with team programming, version control, and configuration management issues. The most widespread is OTI's ENVY/Developer system, which works with Smalltalk/V, VisualAge, and VisualWorks. For Smalltalk/V, Digitalk makes Team/V. These are both relatively expensive packages, but there are a number of other lower-priced packages available as well.

> "For the vast majority of applications, garbage collection should not pose any problems."

How can I find out what's available?
Rather than give contact information for all the products mentioned here, I'm providing pointers to some general resources for finding commercial products.

- ***The Smalltalk Report.*** This magazine is a useful resource, as it has a lot of information about commercial products in reviews, advertisements, and new product anouncements.
- ***The Smalltalk Resource Guide.*** Creative Digital Systems publishes this; it attempts to be a complete listing of Smalltalk-related products and resources. (293 Corbett Avenue, San Francisco, CA 94114 v/f: 415.621.4252, email: 72722.3255@compuserve.com or cds.sem@applelink.-apple.com.)
- ***The Smalltalk Store.*** This is a mail-order source for Smalltalk products. (405 El Camino Real, #106, Menlo Park, CA, 94025. v: 415.854.2557, f: 415.854.2557, email: 75046.3160@compuserve.com or info@smalltalk.com.)

Isn't Smalltalk too slow for commercial applications?
In general, no. Smalltalk does have overhead in both speed and space relative to assembly language or optimized C, particularly for very small programs. This has restricted its use in the low-end shrink-wrapped software market, where development time is much less important than the ability to run on low-end hardware.

These obstacles have been diminishing for some time. Many of Smalltalk's advantages don't show up in small benchmarks but can be very valuable in larger programs. For example, consider garbage collection. On a small benchmark, a C program can usually allocate all of its storage on the stack and avoid any storage management overhead. For larger programs, some form of storage management becomes essential, and often ends up being implemented using a simple but inefficient technique such as reference counting. Smalltalk's memory allocation is much more efficient than malloc, and its garbage collection is much more efficient than most manually implemented techniques. This and other factors can result in large Smalltalk programs outperforming similar programs in other languages.

In fact, Smalltalk has been and is being successfully used in many areas where many people had thought it was completely unsuitable. Hard real-time systems are often cited as an area where Smalltalk could not be used, yet Smalltalk has been successfully used on a number of commercial real-time systems, including a line of oscilloscopes from Tektronix.

Another factor is the use of operating system facilities. In modern programs with graphical user interfaces, the main bottleneck is often calls to the windowing system. When this is the case, the efficiency of the remaining code is much less important. This factor has made interactive development environments like Smalltalk and Visual Basic much more widespread.

How portable is Smalltalk code?
There are two main questions here—portability between platforms and portability between Smalltalk vendors. The first might concern, for example, portability between Smalltalk/V Mac and Smalltalk/V Windows. The second might concern portability between Smalltalk/V Mac and VisualWorks for the Macintosh.

In both cases, code that does not involve the user interface should be very portable. With a few minor tweaks, it is usually possible to load non-GUI code directly into any version of Smalltalk. The exceptions occur when using classes that do not exist in the other version or have different semantics. This is usually not a big problem.

User interface code is more difficult to port, particularly between dialects of Smalltalk. VisualWorks is strongest in this area, as it can normally use identical code on any supported platform. Digitalk code is much less portable, because it uses native platform facilities and often has different user interface frameworks on different platforms. None of the other vendors support multiple platforms yet, but both IBM and QKS are promising to be very portable.

How do I destroy an object in Smalltalk?
You don't destroy objects in Smalltalk—the garbage collector destroys them automatically. This is a great mental leap for people used to programming languages with manual storage allocation. Once there are no more references to an object, it will be disposed of. It's that simple. . . most of the time.

Problems can arise when there is cleanup that needs to be done when an object is destroyed. One example of this is the global Dependents dictionary. At least some objects implement dependents by adding themselves to this dictionary. When the object is no longer referenced, that dictionary entry needs to be removed.

Another example is that of objects that refer to non-Smalltalk storage. For example, FileStreams may refer to operating system file handles, which should be closed if the object is garbage collected.

There are three ways you can deal with this:

- *Avoid it.* ParcPlace provides a class Model with a dependents instance variable. This removes the need to do any cleanup on objects inheriting from Model. This is good if you can do it, but it doesn't solve the problem for other classes, and it won't work when dealing with operating-system storage.
- *Make the programmer do it.* Many objects support a message like release, which does any necessary cleanup. The programmer is expected to send this message when the object is no longer needed. This works, but it rather defeats the purpose of garbage collection, because the programmer must now know when the object is not needed.
- *Finalization.* It's possible to provide hooks into the garbage collection mechanism to run code when a particular object is garbage collected. This is, in my opinion, the ideal solution, but it is currently only supported by SmalltalkAgents and VisualWorks.

Isn't garbage collection too slow for real applications?
No. Garbage collection algorithms have been the subject of a great deal of research and are very carefully tuned. Most operate incrementally, so the system pauses only for extremely short intervals unless it is critically short of space. For the vast majority of applications, garbage collection should not pose any problems at all. Most large applications need to do some form of storage management, and a built-in garbage collector can be much more efficient than one written from scratch for the application.

Does the garbage collector work if...?
For example, what if I have code like this:

```
| garbage1 garbage2 |
garbage 1 := Array new: 1.
garbage2 := Array with: garbage1.
garbage1 at: 1 put: garbage2.
```

Now there are two objects, each of which has a reference to the other but which no other object knows about. Both objects should be garbage, but the garbage collector can't tell that by looking at either of them individually. Will the garbage collector work?

Yes. This is known as cyclic garbage, and any garbage-collected system should be able to handle it. This is only a problem for systems using reference counting. Very few systems with built-in garbage collectors use reference counting. Even without the cyclic garbage problem, it is much less efficient than other methods. ■

Alan Knight is a consultant with the Object People. He can be reached at 613.225.8812 at by email at knight@acm.org.

```
        labelAndIcon := (LabelAndIcon with: window label offset: 4@0)
                icon: icon.
        reversingWrapper := ReversingWrapper on: labelAndIcon.
        reversingWrapper reverse setValue: true.
        BoundedWrapper on: reversingWrapper].
    sequenceView lineGrid: 32
```

The example above used LabelAndIcon objects, VisualBlock objects, and ReversingWrapper objects, which are often very instrumental in defining a list widget's visual blocks. Also, we had to adjust the grid size of the SequenceView. The grid is the total height, measured in pixels, allotted to each element in the SequenceView. Open ListExample to verify that the code works.

MULTIPLE SELECTION LIST EXAMPLE

As one final example, edit the specs in your list such that it is now a multiple selection list. Now edit the windows method so that it reads as follows:

```
windows
    windows isNil
        ifTrue: [windows := MultiSelectionInList with:
        ApplicationWindow allInstances]
        ifFalse: [windows]
```

Normally, a multiple selection list uses a check mark to indicate a selection. To make a multiple selection list use highlighted selections instead and send normalSelection to the MultiSequenceView in a post build operation. Edit the postBuildWith: method to read as the follows:

```
postBuildWth: aBuilder
    (aBuilder componentAt: #windows) widget normalSelection
```

This will cause the unselected items to be drawn regularly and selected items to be drawn highlighted—no check marks are used. Open ListExample to verify this.

CONCLUSION

List components display to the user the information in a sequenceable collection. The aspect model is a SelectionInList (MultiSelectionInList), and the widget is a SequenceView (MultiSelectionSequenceView). VisualWorks makes the collection a dependent of the SequenceView. The List class was created to operate specifically with list components. Whenever its content changes, it uses this dependency relationship to notify the SequenceView, which redisplays itself. The elements in the collection can be displayed in the list component in a variety of ways—string, formatted text, or some arbitrary visual component. A SequenceView can use printString, displayString, or any arbitrary display selector to represent the elements in the collection. The SequenceView object's visual blocks can be redefined such that the elements in the collection are represented in the list as visual components. ■

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

Tim Howard has been developing application software for the past eight years and holds an MBA and a Masters degree in industrial engineering. He can be reached at 74213.1517@compuserve.com.

Kent Beck

Birds, bees, and browsers—obvious sources of objects

This is the fourth and final installment in my series on where objects come from. I deliberately started with the unusual and difficult ways of finding objects. There are lots of books that will tell you how easy it is to find objects. Just underline the nouns! The fatuous phrase that keeps popping up is, "...there for the picking." Or maybe it's "plucking." In any case, none of the objects you'll find with Objects from States, Objects from Variables, Objects from Collections, or Objects from Methods is there for the picking. They are, rather, deep, powerful objects that will change the way you see and structure your systems.

That's not to say that program-derived objects are the only important objects. There are a couple of kinds of objects that are necessary for a well-structured application. They just aren't sufficient to take full advantage of all the benefits objects can offer. Here are two patterns that capture the way I think about obvious objects: Objects from the User's World and Objects from the Interface.

Pattern: Objects from the User's World

Problem: What are the best objects to start a design with?

Constraints: The way the user sees the world should have a profound impact on the way the system presents information. Sometimes a computer program can be a user's bridge to a deeper understanding of a domain. However, having a software engineer second guess the user is a chancy proposition at best.

Some people say, "I can structure the internals of my system any way I want to. What I present to the user is just a function of the user interface." In my experience, this is simply not so. The structure of the internals of the system will find its way into the thoughts and vocabulary of the user in the most insidious way. Even if it is communicated only in what you tell the user is easy and what is difficult to implement, the user will build a mental model of what is inside the system.

Unfortunately, the way the user thinks about the world isn't necessarily the best way to model the world computationally. In spite of the difficulties, though, it is more important to present the best possible interface to the user than to make the system simpler to implement. Therefore...

Solution: Begin the system with objects from the user's world. Plan to decouple these objects from the way you format them on the screen, leaving only the computational model.

Comment: This is a pattern Ward Cunningham and I named years ago when we first began exploring patterns. Looking at it again was interesting. I was reminded why having the user's objects at the center is so important. So many effects flow subtly from the object model to the user. I always know a project is in trouble if I come in and the Parsers and ProcessSchedulers are in the middle of the table.

Pattern: Objects from the Interface

Problem: How can you best represent a modern interface with objects?

Constraints: A natural tendency is to want to make big user interface objects—an entire table, a row of buttons, etc. This may be a legacy from procedural programming, wherein separating functionality into pieces is difficult. The problem with this approach is that the result is inflexible. If you want to add another button to the row or change the way the table behaves, you may have to touch many parts of the code.

A better approach is to make many smaller-grained user interface objects. The more you can compose your user interface out of small objects, the more flexibility you have, and flexibility is at a premium in user interface design and implementation. The user interface is the part of the system that will remain unstable longest, long after the underlying model has shaken out.

Solution: Find objects in the user interface. As much as possible, make each identifiable thing in the interface into an object and build larger entities by composing them together. The lowest level user-interface objects become like the tokens in a programming language.

Comments: I sort of like the way Objects from the User's World turned out, but I think Objects from the Interface isn't very good. Actually, the design of objects to support user interface is the result of a whole system of patterns. I think I succumbed to "big patternitis," the disease in which you want to look comprehensive and you end up saying

continued on page 17

GETTING REAL

Juanita Ewing

Return values

Every method has one return value. A return value can be a Boolean indicating the success or failure of the operation, a new object that is the result of the operation, or an existing object such as the receiver of the method. The default return value of a method is self, the receiver of the method.

In this column, we examine some common return values from methods. Evolution of interfaces can result in an ad hoc variety of return objects that are more difficult for a client to use. A better alternative is specialized objects that encapsulate return values.

WHAT HAS RETURN VALUES?

In Smalltalk, there are two structures composed of a sequence of statements, that have a return value: blocks and methods. These structures have two kinds of returns, implicit and explicit returns. This column discusses method returns in detail.

An *explicit* return consists of a return statement and causes an immediate return from the method context, even if the return statement is inside a block. The return value is the value of the expression to the right of the return operator (^). The expression ^nameCollection includes: myName returns true or false, depending on whether myName is included in nameCollection.

Another form of return is an *implicit* return. Implicit returns are performed when no explicit return is performed. An implicit return is performed when execution "falls off the end" of the method.

Blocks and methods have different implicit return semantics. For methods, the implicit return value is the receiver of the method, otherwise known as self. For blocks, the return value is the value of the last statement in the block, or nil if the block has no statements.

EXAMPLES

Smalltalk class libraries are filled with examples of implicit and explicit returns. The method displayOn: has an implicit return. The implicit return is performed after the last statement in the method:

```
Wedge
displayOn: aGraphicsTool
    "Graphically display the receiver on <aGraphicsTool>."

    self fillOn: aGraphicsTool
```

The truncated method for Line has an explicit return, but it is the same as the implicit return. Sometimes developers use an explicit return for emphasis, even though it is not necessary. In this case, it is probably because the method, unlike other similar line methods, returns the receiver rather than a copy of the receiver:

```
Line
    truncated
        "Answer the receiver with the coordinates of its end points
            truncatedto integers."

        self start: self start truncated.
        self end: self end truncated.
        ^self
```

Other common return objects are nil, true, false, and strings. By convention, a return value of nil usually indicates an exception condition or an error. Many older class libraries written in the days before exception handling had old methods that started out with a return of self. Later, some error checking might be added and another return value might be used to indicate the error. The other return value was typically nil, which can be easily tested by clients. The method operationFooOn illustrates a conditional return of nil:

```
operationFooOn: anObject
    "Perform the operation foo on <anObject>. Return nil if error."

    | fooizedObject |
    (self compatibleWith: anObject)
        ifFalse: [^nil].
    fooizedObject := self prepare: anObject.
    self foo: fooizedObject
```

Another common way to indicate an error is to return a string describing the error. The class method bindTo: checks the return value for the bindTo: instance method:

```
ObjectLibraryBind class
bindTo: aDLLName
    "Bind the ObjectLibrary with <aDLLName> into the current image."

    | result |
    result := self new bindTo: aDLLName.
    result isString ifTrue: [ self error: result ].
    ^result
```

A BAD IDEA

As operations grow more complex, there is a tendency for the

interface to operations to become broader. Sometimes broadness takes the form of disparate return values, each returned under a different condition. Methods with widely differing return values require the client to execute conditional code or perform a kind of case statement in order to use the result of the method invocation.

Let's look at an example of a complex operation. The operation has these characteristics:

- It might not succeed.
- The operation has a second chance of success—it can be retried, with some input ignored.
- If the operation fails, it might be because of an internal error or because an external function failed. For debugging purposes, it is desirable to distinguish between the two.
- Another effect of the operation is the creation of an OrderedCollection of strings containing result data from the operation.

One possible solution to the problem of how to return this information is to use different return values to indicate how the operation proceeded. This solution uses the following set of return objects (with their interpretation):

- *self.* Operation completed without error and return object represents success.
- *nil.* Operation completed but some input data was ignored. Return object represents conditional success. If the invocation of the method has user interaction available, it would be appropriate here.
- *string.* Operation failed due to internal constraints, and return object represents error message.
- *integer.* Operation failed due to external constraints, and return object represents error code. Client must look up message in error-code table. Desirable to have error code in case error-code table is inconsistent with external interface.

This solution also makes use of a global variable to contain the collection of result strings. The operation has a side effect of setting the global variable to the collection.

Clients of this method must test for the kind of return value to interpret the result of the operation. The testing of the return value and its interpretation looks like a secret code between the method and the client. Occasionally, a developer has to break the secret code to maintain the application. Client code might look this:

```
invokeOperation
    "Invoke the operationWithPoorInterface. Interpret the result and
    conditionally return the global variable <GlobalResult> if the
    operation succeeded. If it failed return an empty collection. If the
    invocation was interactive, notify the user of the operation results."

    | result errorMessage |
    result := self operationWithPoorInterface.
    result == self
        ifTrue: [self invocation isInteractive
                ifTrue: [self notifySuccess].
                ^GlobalResult].
        result isNil
            ifTrue: [self invocation isInteractive
                    ifTrue: [self notifyDataIgnored].
                ^GlobalResult].
        result isString
            ifTrue: [self notifyError: result.
                ^OrderedCollection new].
        result isInteger
            ifTrue: [errorMessage := (self class errorTable at: result ifAbsent:
    ['unknown error'].
                self notifyError: result printString,': ', errorMessage.
                ^OrderedCollection new]
        ^self error: 'unable to interpret result of
            operationWithPoorInterface'
```

The messages that can be sent to the result depend on the type of object. If the return values from a method are highly polymorphic, then the client for the method can use the result without testing for the kind of object.

This solution suffers from several problems:

- *Ease of Use.* The client must map the return value to the correct action, based on the kind of return object. The kind of return object can be arbitrary and therefore difficult for the developer to interpret correctly.
- *Maintenance.* Requires the client to have a "case" statement that is error prone during evolution and maintenance—if the set of return values changes, then all clients must be updated.
- *Encapsulation.* A global variable that is set as a side effect of the operation is problematic because it is not protected from modification outside of the operation.

ANOTHER BAD IDEA

The other bad idea is to package multiple return objects into one generic return object, such as an array. We can rework the above example to use an array for a return value. The array contains much of the information above, but in a different form:

- *Element 1.* Success Boolean. Set to true if the operation succeeded. Set to false if data was ignored or the operation failed.
- *Element 2.* Data-ignored Boolean. Set to true if the operation succeeded by ignoring some input data. Set to false if otherwise.
- *Element 3.* Error message. Set to an empty string if the operation succeeded. Set to a descriptive string if it failed.
- *Element 4.* Error code. Set to nonzero if external operation failed. Error message is also set.
- *Element 5.* Collection of result strings. Set to an empty collection if operation failed.

In this form, we do not use a global variable to return the collection of result strings. Now the client must interpret the contents of the array instead of the set of return values. The code for the client might look like this:

```
invokeOperation
    "Invoke the operationWithPoorInterface. Return a collectionof
    strings if the operation succeeded. If it failed, return an
    emptycollection. If the invocation was interactive, notify the user of
    theoperation results."

    | result errorMessage errorCode |
    result := self operationWithPoorInterface.
    (result at: 1)  "success"
        ifTrue: [self invocation isInteractive ifTrue: [self notifySuccess].
            ^result at: 5].
    (result at: 2)  "data ignored"
        ifTrue: [self invocation isInteractive ifTrue:
                [self notifyDataIgnored].
            ^result at: 5].
    errorMessage := result at: 3.
    errorCode := result at: 4
    errorCode > 0
        ifTrue: [errorMessage := (self class errorTable at:
errorCode ifAbsent: ['unknown error']), ': ',errorMessage].
    self notifyError: errorMessage.
    ^OrderedCollection new
```

This solution does not have a side effect of setting a global variable. But, there are three reasons why capturing a set of return objects in an array is not a good choice:

- *Ease of Use.* Clients of the method returning an array need to use arbitrary indices to obtain the data instead of sending messages with meaningful names.
- *Encapsulation.* Multiple return objects form a coherent set and should have behavior to represent operations on these objects. An array has no way to encapsulate behavior, and, consequently, clients of the method need to write much more code to duplicate the behavior of the return set of objects. Each client writes the same code over and over.
- *Information Hiding.* With an array representing multiple return objects, the constituent data, including private data, is accessible to all clients. Also, the formation of the objects within the array cannot be changed without affecting all clients.

For a more complete discussion of these issues, see the article "Don't Use Arrays?" SMALLTALK REPORT (2[7]).

A GOOD IDEA

There is an alternative to packaging different return values in a generic data structure or returning several kinds of objects. The alternative is a single instance of specialized return object. A specialized return object can encapsulate the success or failure of an operation, error descriptions, and multiple results. Independent of the success of the operation, all clients can send the same messages to the result. A single return object is easier for clients to use and encapsulates appropriate behavior.

Using the same problem description, here is a solution that uses a specialized return object. A partial description of its protocol follows:

```
SpecializedReturnObject
    wasSuccessful "Return true if the operationsucceeded."
    wasDataIgnored  "Return true if the operation ignored some input
        data, false otherwise."
    errorMessage "Return an empty string if the operation succeeded,
        a descriptive string if it failed. If the failure is external, the
        message includes the error code."
    errorCode  "Return nonzero if external portion failed, zero otherwise."
    stringCollection  "Return a collection of result strings empty if the
        operation failed."
```

Using a specialized return object, the client of the operation sends messages to interpret results instead of testing for arbitrary return values or access arbitrary elements in an array. Client code might look like this:

```
invokeOperation
    "Invoke the operation WithPoorInterface. Return a collection of
    strings if the operation succeeded. If it failed, return an empty
    collection. If the invocation was interactive, notify the user of the
    operation results."
| result errorMessage |
result := self operationWithPoorInterface.
result wasSuccessful
    ifTrue: [self invocation isInteractive ifTrue: [self notifySuccess].
        ^result stringCollection].
result wasDataIgnored
    ifTrue: [self invocation isInteractive ifTrue:
            [self notifyDataIgnored].
        ^result stringCollection].
self notifyError: result errorMessage.
^OrderedCollection new
```

In this example, special code that dealt with the error code was dropped out. It is not necessary for the client to know the error code for the error message to be composed. Instead, it is composed by the return object. During a debug session, the error code can still be individually accessed.

A REAL LIFE EXAMPLE

Compilation is a complex operation with multiple results. The solution used by Smalltalk/V for compilation results is a specialized return object, an instance of CompilationResult. The behavior for CompilationResult from Smalltalk/V for Win32 (simplified for presentation—see sidebar). The behavior can be divided into several categories:

- methods for determining the success of the operation and dealing with errors
- methods for returning the results of successfully compiling some source code
- methods for returning the results of parsing some source code, subdivided into:
  1. local names
  2. selector
  3. miscellaneous results, such as the messages sent by the method.

Depending on the client, results of compiling could be used to create methods for classes, to build static databases of messages,

COMPILATION RESULT

success	
wasSuccessful	"Answer <true> if the receiver represents a successful compilation."
error	"If the compilation was unsuccessful then return the Compilation Error object that describes the error that terminated compilation."
errorMessage	"If the compilation was unsuccessful, then return a string describing the error that terminated compilation, otherwise return an empty string."
compilation results	
association	"Answer an association between the method selector and the compiled method that was created. Answer nil if the compilation was unsuccessful or if what was compiled was not a method"
method	"Answer the compiled method that was created by the compilation. Answer nil if the compilation was unsuccessful."
parse results–local names	
argumentNames	"Answer a list of the names of all method arguments defined within the source."
localNames	"Answer a list of the names of all variables defined within the source."
nonLocalNames	"Answer a list of the names of all variables referenced but not defined within the source."
parse results–selector	
selector	"Answer the method selector by which the compiled method would normally be invoked. Answer nil if the expression was compiled or if the compilation was unsuccessful."
parse results–other	
messages	"Answer a list of the messages sent within the source."
primitiveSpecification	"Answer a string containing the primitive specification found in the source code, including the enclosing brackets ('<' and '>'). If the source code had no primitive specification, the answer is an empty string."
sourceCode	"Answer the source code that was parsed. This may be different from the original source code if the source was modified by an error handler."
temporaryNames	"Answer a list of the names of all temporary variables and block arguments defined within the source."

or to collect data for metric and productivity measurement. Typical use of the interface to the compiler looks like this:

```
invokeCompiler
    "Invoke the compiler. Return a new compiled method if the
compilation was successful, nil otherwise."
    | compiler compilerResult |
    compiler := self compilerClass forClass: self classToCompileFor.
    compilerResult := compiler compile: initialSource.
    compilerResult wasSuccessful ifFalse: [ ^nil ].
    self install: compilerResult association withSource:
      compilerResult sourceCode.
    self buildCallTableFrom: compilerResult selector to: compilerResult
      messages.
        ^self
```

ADVICE

When designing systems with complex operations, pay attention to the interface between the client and the operations. If there is a requirement for multiple return values, consider the use of specialized return objects.

Analyze interfaces to existing complex operations. Be wary of:

- sets of return objects,
- the use of "case" statements in client code to analyze return values, and
- generic data structures, such as arrays, which are analyzed by the client.

These are signs of code that could benefit from specialized return objects. The results of rework with specialized objects will be more understandable, extensible, maintainable, and reusable. ■

Juanita Ewing is a senior staff member of Digitalk, Inc. 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. Previously, at Textronix Inc., she developed class libraries for the first commercial-quality Smalltalk-80 system. She can be reached via email at juanita@digitalk.com or by mail at Digitalk, Inc., 7585 SW Mohawk Drive, Tualatin, OR 97062.

■ SMALLTALK IDIOMS

continued from page 13

something so vague as to be unusable. I'll leave that pattern there, though, as an example of how not to do it.

This concludes a four-part series on where objects come from. Looking back, I can see I have wandered pretty far afield from the hard, practical information I wanted to present in this column. I'm not sure what I'll do next, but I think I may even give patterns a rest for a while. Maybe something about what goes on under the hood in VisualWorks and V. Maybe some virtual machine secrets. What do you think? Let me know at 70761.1216@compuserve.com. ■

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

GUIs

Kyle Brown

GUI Smalltalk: The VisualWorks UIBuilder

VisualWorks allows programmers to easily generate complex user interfaces in a highly automated way. The same capabilities that make it possible to draw windows using the point-and-click UIPainter tool are also used to generate the resulting windows at runtime. By taking a peek under the covers of how VisualWorks constructs its windows, we can gain a greater understanding of how to apply the user hooks that VisualWorks provides and start to see how to use this knowledge to extend VisualWorks.

A VISUALWORKS WINDOW FROM THE OUTSIDE IN

Let's start by looking at a simple VisualWorks window (constructed using the UI Painter) containing an ActionButton, an InputBox, and a Label. (See Fig. 1.)

Figure 2 shows the objects that make up the visuals seen in the window. They proceed inward (in an "is-part-of" relationship) from the ApplicationWindow.

What the pieces do

ApplicationWindow—The application window is the topmost layer of the VisualWorks window system. It represents the operating system window that you see. It is different from the rest of the components of the window in that the class ApplicationWindow is not a subclass of VisualComponent—it is not something that can represent itself with a graphics context or be a subcomponent of something else. This is why it is so difficult (maybe impossible!) to do MDI under VisualWorks. Instead, ApplicationWindow is a subclass of DisplaySurface. This means it possesses a graphics context that it can pass to other objects with which they can display themselves.

KeyboardComposite—A KeyboardComposite is a special subclass of CompositeView that connects to a KeyboardProcessor through its controller. A KeyboardProcessor coordinates keyboard focus for all the Views contained in the KeyboardComposite.

CompositeViews are a subclass of CompositePart. A CompositePart is a visual component that holds a group of other visual components within its bounds.[1] For instance, when you "group" things in VisualWorks, a CompositePart is created to hold the things in the group.

SpecWrapper—The SpecWrapper binds together the component represented on the window and the spec that created it. SpecWrappers are returned when you ask for "componentAt:" to a builder. SpecWrappers are a subclass of WidgetWrapper that manage the visual state of the component—that is, is it visible or invisible, what LookPreference does it use, and so forth.

BoundedWrapper and BorderedWrapper or LayoutWrapper—These wrappers set the position and size within the window of the component that they contain. If you set a component to have a border in the UIPainter, then it is wrapped in a BorderedWrapper; otherwise it is wrapped in a BoundedWrapper. In some cases, another wrapper, a *LayoutWrapper*, is used. A LayoutWrapper works from an instance of LayoutFrame or LayoutOrigin rather than from a rectangle.

At the very bottom of the stack are the UI components—the subclasses of View that actually accomplish something like editing text or displaying graphics, and so forth.

THE BUILD PROCESS

Now that we've seen what a VisualWorks window is composed of, the questions remains: How do the pieces of a window we have just seen get from the spec generated by the UIPainter onto the window? This is the result of the VisualWorks interface building process. This process goes through several steps, with user hooks at each level to allow the process to be modified or enhanced.

First Step: Setup

When a VisualWorks window is created, the message open is sent to an instance of ApplicationModel. Note that this occurs *after* the instance of ApplicationModel is created, so any initialization that took place in the initialize method of that particular ApplicationModel subclass has already occurred.

The open method sends the openInterface: message, with the particular windowSpec symbol (i.e., #windowSpec) as a parameter.*

```
openInterface: aSymbol
    "Open the ApplicationModel's user interface, using the specification
    named."

    | spec |
    builder := UIBuilder new.
    builder source: self.
    spec := self class interfaceSpecFor: aSymbol.
    self preBuildWith: builder.
```

```
builder add: spec.
self postBuildWith: builder.
builder window model: self.
builder openWithExtent: spec window bounds extent.
self postOpenWith: builder.
^builder
```

This method first creates a new instance of UIBuilder, then creates an instance of FullSpec by sending its class the message interfaceSpecFor: with the windowSpec symbol as a parameter. This method turns over the job of creating a new interface spec to the class UISpecification. The class UISpecification then reads the literal array version of the specification from the windowSpec method, decodes the literal array format of the UI Specification, and returns an instance of FullSpec. Here is what a FullSpec consists of:

A WindowSpec represents the attributes of the ApplicationWindow, that is, its min and max size, its bounds, its label, and so forth. The component attribute of a FullSpec contains a SpecCollection that contains (in its collection attribute) the individual specs of each of the components of the window to be built.

As the next step in the build process, the first user hook, preBuildWith: is called. At this point, both the UIBuilder and FullSpec exist, but nothing has been done with them. You can modify the UIBuilder in this method, but unfortunately, the FullSpec is not passed as a parameter. (Of course, you could always change the method openInterface: to correct this oversight).

Second Step: Creating the components

The next message sent by the openInterface: method is the add: message sent to the UIBuilder with the newly created FullSpec as the parameter. This simple little message accomplishes most of the work of putting together the VisualWorks window.

```
add: aSpec
    "Reset current internal state and build within the current composite
    according to aSpec."

    self startNewComponent.
    self addSpec: aSpec.
    ^wrapper
```

The message startNewComponent cleans out some instance variables of the UIBuilder and sets it up to begin work on a particular component. This message then sends addSpec:, which sets the UIBuilder's spec attribute to be the new specification passed in and turns the process over to the specification by sending it the addTo:withPolicy: message.

At this topmost level, aSpec is an instance of FullSpec. FullSpec implements addTo:withPolicy: by telling the builder to add: its windowSpec and ComponentSpec in turn. Similarly, SpecCollection implements this message by iterating over its collection of specs and telling the builder to add: each of them in turn. However, the superclass UISpecification implements the method this way:

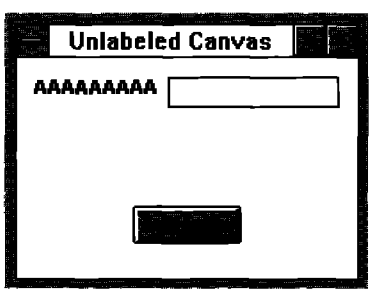


Figure 1. VisualWorks window.

```
addTo: builder withPolicy: policy

    self dispatchTo: policy with: builder.
    self finalizeComponentIn: builder
```

The method dispatchTo:with: is overridden in all concrete subclasses of UISpecification. As an example of how this process works, let's trace down the adding of one of the specs, the InputFieldSpec, and see how the methods are implemented.

InputFieldSpec implements dispatchTo:with: in this way, by using double-dispatching:

```
dispatchTo: policy with: builder

    policy inputBox: self into: builder
```

The class UILookPolicy and its subclasses implement many methods that are of the form component:into:, in which component is the name of the UI component being added (listView, textEditor, etc.). These methods actually create the instances of the correct View subclasses for each specification. If you wanted to extend out the VisualWorks palette with your own View, one of the first steps (which is not mentioned in the relevant chapter of the VisualWorks user's manual) should be to implement these methods for each UILookPolicy subclass in the system.

Coming back to our example, the class UILookPolicy implements inputBox:into: in this way:

```
inputBox: spec into: builder

    | component model menu performer alignment |
    model := spec modelInBuilder: builder.
    component := self inputBoxClass new.
    spec type == #password ifTrue: [component displayContents:
        PasswordComposedText new].
    component model: model.
    self setStyleOf: component to: spec style.
    component controller: self inputBoxControllerClass new.
    (menu := spec getMenuIn: builder) == nil
        ifFalse: [component controller menuHolder: menu].
    (performer := spec getPerformerIn: builder) == nil
        ifFalse: [component controller performer: performer].
    builder component: component.
    component displaySelection: false.
    spec tabable
        ifTrue: [component widgetState isTabStop: true.
                builder sendKeyboardTo: component]
        ifFalse: [component widgetState canTakeFocus: true.
                component controller keyboardProcessor: builder
                    keyboardProcessor.
                builder component controller dispatchOn: Character cr
                    to: #acceptKey:.].
    spec isReadOnly ifTrue: [component controller readOnly: true].
    spec numChars == nil ifFalse: [component controller maxChars:
      spec numChars].
    (alignment := spec alignment) == #left
        ifFalse: [self setAlignmentOf: component displayContents to:
          alignment].
    builder wrapWith: ScrollWrapper new.
    spec decorationType == #bordered
        ifTrue: [builder wrapWith: self borderedWrapperClass new.
                builder wrapper inset: 0.
                builder wrapper border: self inputFieldBorder]
        ifFalse: [builder wrapWith: self boundedWrapperClass new].
    builder applyLayout: spec layout
```

The aforementioned methods (in boldface) indicate most of the steps necessary to create the component. In general, most component:into methods do all or most of the following six steps:

- create the component's model (if necessary) by calling modelInBuilder:
- create the component
- set the component of the builder to be the new component
- do any necessary initializations and setup
- wrap the new component in an appropriate wrapper.
- set the layout of the new component's wrapper from the specification.

If the component is one that requires a model, then it will be obtained (in most cases) in the modelInBuilder: method by getting the return value of sending the aspect message that was defined in the spec to the builder's ApplicationModel.

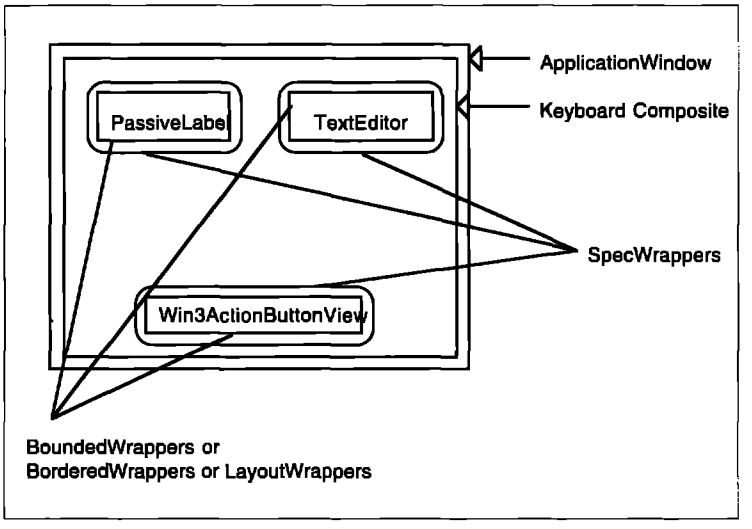


Figure 2. VisualWorks objects.

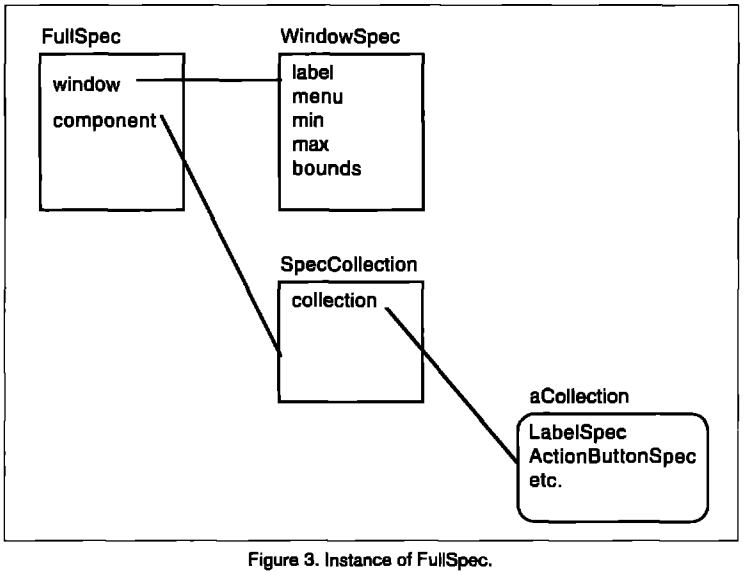


Figure 3. Instance of FullSpec.

Usually, obtaining the correct wrapper for the component is done by sending a particular UILookPolicy the manufactureGeneralWrapperFor:into: method. This method creates a wrapper based on the type of the layout attribute of the spec and the value of the decoration attribute. This allows a component to be bordered, unbordered, or to have scroll bars based on the decoration flags. There are a few exceptions to this rule that need to specify their wrappers differently, as does inputBox:into: above.

Finally, the finalizeComponentIn: message adds the SpecWrapper around the newly created wrapper and adds the resulting wrapper into the builder's dictionary and the builder's composite.

At this point, most of the work of creating the window is finished. By now, all the window components have been created and connected to each other. All that remains is for the window to be opened.

Last Step: Opening the window

Back up at openInterface: events quickly reach an end after the specs have been added. The user hook postBuildWith: is called next. Any changes to the layout of the UI components or special initializations outside of those done in the build process can be done in this user-defined method. Additionally, any keyboard hooks (which trap keys before they are passed to lower-level components) can be defined in this method.

The message postBuildWith: is followed by actually opening the window itself (UIBuilder openWithExtent:). This message sends the openWithExtent: message to the window held by the UIBuilder, which opens and schedules the window. Finally, the user hook postOpenWith: is called and the openInterface: method returns.

That just about does it. We've managed to trace down from the spec built up by the UIPainter all the way to the Objectworks window objects actually built by the UIBuilder. In the next article of this series, we'll explore how to use some of the information we've gained to build an OS/2-like notebook pane in VisualWorks. ■

References

1. Liebs, D., and K. Rubin. Reimplementing model-view-controller, The Smalltalk Report (1)6:1–7, 1992.

Kyle Brown is a Senior Member of Technical Staff at Knowledge Systems Corp. He has been developing custom views for Objectworks/Smalltalk for over three years. As part of his consulting practice, he has developed custom graphical interfaces for applications in Engineering, MIS, and scientific computing. Since joining KSC, Kyle has enjoyed teaching the principles of reuse and good O-O design to a variety of clients through the KSC Smalltalk Apprentice Program.

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.

EASEL CORPORATION ANNOUNCES NEW VERSION OF ENFIN APPLICATION DEVELOPMENT ENVIRONMENT AND TEAMBUILDER

Easel Corporation announced an upgrade of its application development environment, ENFIN Release 4.0. The company also introduced TeamBuilder, a development tool that enables groups of developers to simultaneously build applications. Both are members of Easel 's Object Studio products family.

ENFIN Release 4.0 offers a new development workspace. All object icons are now organized by functional group and are color-coded to help developers visualize functions. It also includes a Main Desktop Apprentice feature that provides online training so developers can become more comfortable with the drag-and-drop environment. For both Windows and OS/2 developers, ENFIN templates can be double-clicked on, as well dragged and dropped.

Other features include a new SQL Editor that increases a developer's ability to build applications that access SQL databases by supporting aliases and calculated columns and providing a nonparsed "direct" mode feature. ENFIN Release 4.0 has added connectivity features, including Oracle support of stored procedures and enhanced DDCS/2 support. A new generic EHLLAPI interface that supports all 3270 emulators that are fully EHLLAPI compliant, including DCA's IRMA Workstation, Wall Data's Rumba, and Attachment's Extra.

TeamBuilder enables a networked software development team to jointly build object-oriented client/server applications. The tool provides check-in/check-out capabilities of classes and files and works within the ENFIN Class Browser. With TeamBuilder, a developer can group objects together as a project, each having a separate revision level, date-stamp and label. This easily allows a developer to reuse other developers' objects while still keeping their projects organized and maintainable. TeamBuilder integrates with Intersolv's PVCS product, providing additional functionality, including archiving older classes and files, comparisons between revisions, and labeling.

Easel Corp., 25 Corporate Drive, Burlington, MA 01803, 617.221.2100 (v), 617.221.6899 (f)

RECRUITMENT

To place an ad, call Shirley Sax at 212.274.0640

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

Smalltalk Designers and Developers

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

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

Smalltalk Developers and Consultants

In order to keep up with the explosive growth in the object technology market, Versant Object Technology, the industry's leading provider of object database management systems (ODBMS) for multi-user, distributed environments has immediate openings for the following positions.

Join the team that offers the industry's most complete product line including the VERSANT ODBMS, application development and database administration tools, multiple programming language interfaces, and a family of support services.

Project Lead (Job Code: eng 11)

Responsible for leading and driving the development of the Smalltalk interface to the VERSANT ODBMS in an aggressive and dynamic engineering organization.

You'll need a minimum of 2 years experience implementing production Smalltalk applications on either UNIX or PC platforms. A working knowledge of C, C++, and UNIX is also required. Experience with either object or relational databases is desirable. An MS in computer science or equivalent experience is required.

Consultant (Job Code: tco 42)

You will be responsible for consulting with our customers throughout their software lifecycle enabling them to leverage Object Oriented Technology, complemented by a suite of Versant tools to solve complex application problems. Position also requires you to interface with Marketing and Engineering to feed customer input to the product planning process. You will also provide training to customers. Required to travel extensively within the US and overseas.

A minimum of 5 years application development experience with 2 years experience focused on Smalltalk. A working knowledge of either object or relational databases is required. UNIX experience mandatory, and PC experience a plus. A BS degree in engineering or computer science required.

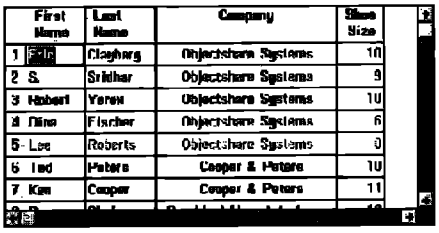
VERSANT offers an excellent compensation and benefits package including a stock option plan, 125K and 401(K) plan. Versant is an EEO/affirmative action employer. For immediate consideration please mail or fax a cover letter and a resume to:

Versant Object Technology, Attn: Personnel (JOB CODE:), 1380 Willow Road, Menlo Park, CA. 94025, or FAX: (415)-325-2380

VERSANT
The Database For Objects™

Get Powerful New Controls for Smalltalk/V®

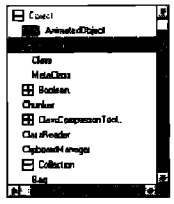
Subpanes™/V is a library of unique controls for Smalltalk/V. Place and edit them interactively with WindowBuilder™ Pro/V. When you use the right controls, your applications will be easier to use. And you'll save time because you won't need to fight controls that aren't right for the job.



A Table of Editable Cells

TablePane provides a scrollable grid of editable cells. In addition to handling a matrix of strings, it can manage a collection of objects. Users edit cells in-line by selecting them with the mouse or keyboard.

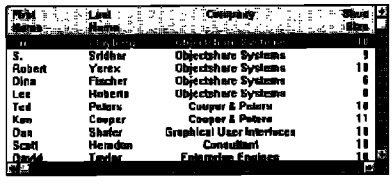
Hierarchical List Box



HierarchicalListBox extends a normal listbox to view a hierarchical group of objects. Collapse or expand the hierarchy, use icons, use indentation to show the relationships. Display any objects that have hierarchical relationships.

A List Box with Columns

ColumnarListBox displays multiple pieces of information about each object of a collection. You control headers, justification, color*, multiple select* and more.



Bitmap Panes, 3D Frames, & More

Subpanes/V also includes BitmapPane, 3D frames, ValueSet, Gauges, date, number, and time editors, BitmapButton, and more.

No Runtime Fees

No runtime fees for applications developed with Subpanes/V. It includes complete documentation, full source, free support to registered users for the first 90 days, and a 30-day money-back guarantee.

SUBPANES/V

NEW! **For OS/2 ..... $235** *(v2.0)*
For Win ..... $129 *(v1.0)*
For Win32 ..... $195 *(v1.0)*

*These features in version 2.0 only. Version 2.0 for Win and Win32 will ship in 3Q94.

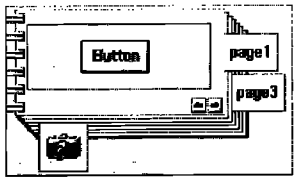
Subpanes/V requires WindowBuilder Pro/V. Subpanes /V is compatible with Team/V and ENVY/Developer. Subpanes is implemented in Smalltalk, as subclasses in Digitalk's Subpane hierarchy. Support subscription available.

...And CUA'91 Controls Are Easy Too!

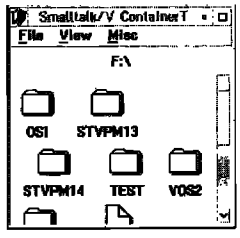
WidgetKit™/CUA'91 is a library of CUA'91 controls for Smalltalk/V. CUA'91 controls provide a distinctive and powerful user interface. WidgetKit/CUA'91 makes them easy to use and portable. Place and edit the controls interactively with WindowBuilder™ Pro/V. WidgetKit/CUA'91's specialized editors give you easy access to all of the control's attributes.

Notebooks, Cached for Performance

CachedNotebooks provide the CUA'91 notebook control. Performance is dramatically improved by dynamic page loading. You get complete control of orientation, tabs, alignment, color, binding, and caching.



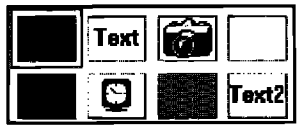
Containers



CuaContainers provide text or icon representations of items they contain. Items can be dragged and dropped between containers. Supports icon, name, text, tree, and detail views. CuaContainers can hold objects of any type.

Value Set and More

CuaValueSet provides a way for users to select from icon and text choices with a mouse click. WidgetKit/CUA'91 also provides full support for the rest of the CUA'91 controls, including slider and spin button.



For WindowBuilder Pro/V

WindowBuilder Pro/V lets you build Smalltalk/V user interfaces fast. Place the controls and edit them interactively. Increase consistency, ease maintenance. Call for a free brochure.

No Runtime Fees

No runtime fees for applications developed with WidgetKit/CUA'91. It includes complete documentation, full source, free support to registered users for the first 90 days, and a 30-day money-back guarantee.

WIDGETKIT/CUA'91

NEW! **For OS/2 ..... $295**
For Win ..... $295 *(3Q94)*
For Win32 ..... $295 *(3Q94)*

WidgetKit/CUA'91 requires WindowBuilder Pro/V. WidgetKit/CUA'91 is compatible with Team/V and ENVY/Developer. Includes DLLs. User interfaces built using WidgetKit/CUA'91 are portable to supported platforms. Support subscription available.

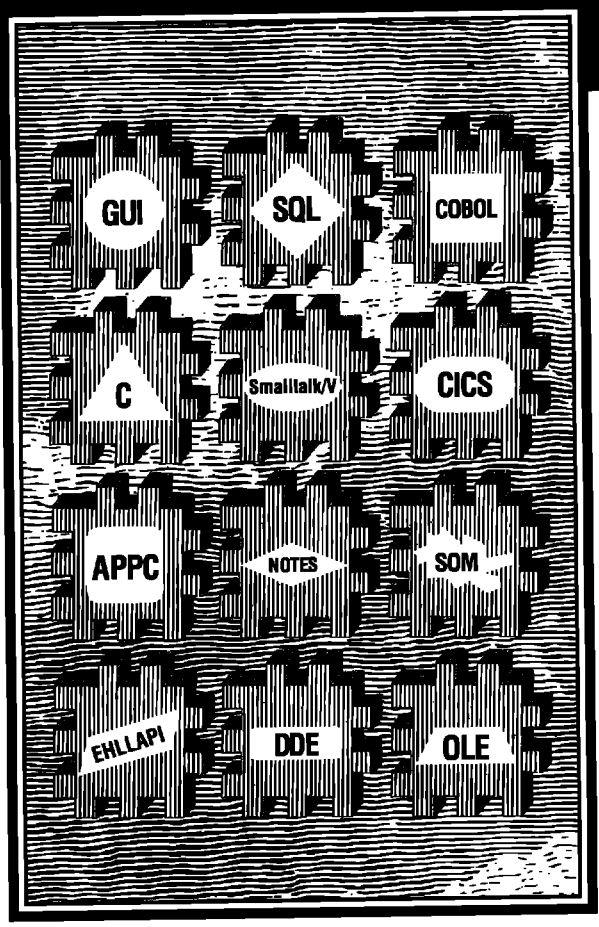
SHARE OBJECT SYSTEMS INC.

Objectshare Systems, Inc.
5 Town & Country Village, Suite 735
San Jose, CA 95128-2026
Fax 408-970-7282
CompuServe 76436,1063

Call to order today (408) 970-7280

9 AM to 5 PM PST, Monday through Friday
30 day money-back guarantee

FINALLY, CLIENT/SERVER INTEGRATION.



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

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

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

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

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

RATED #1 – TWICE.

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

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

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

10 YEARS EXPERIENCE.

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

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

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

PARTS. THE CLIENT/SERVER INTEGRATION TOOL. DIGITALK