



Editors

John Pugh and Paul White
Carleton University & The Object People

SIGS Publications Advisory Board

Tom Atwood, *Object Design*
François Bancelhon, *O₂ Technology*
Grady Booch, *Rational*
George Bosworth, *ParcPlace-Digital*
Jesse Michael Chonoles, *Lockheed Martin ACC*
Stuart Frost, *SELECT Software*
Adele Goldberg, *ParcPlace-Digital*
Thomas Keffer, *Rogue Wave Software*
R. Jordan Kriendler, *IBM Consulting Group*
Thomas Love, *Consultant*
Bertrand Meyer, *ISE*
Mellir Page-Jones, *Wayland Systems*
Cliff Reeves, *IBM*
Bjarne Stroustrup, *AT&T Bell Labs*
Dave Thomas, *Object Technology International*

The Smalltalk Report

Editorial Board

Jim Anderson, *ParcPlace-Digital*
Adele Goldberg, *ParcPlace-Digital*
Reed Phillips
Mike Taylor, *ParcPlace-Digital*
Dave Thomas, *Object Technology International*

Columnists

Jay Almarode, *GemStone Systems Inc.*
Kent Beck, *First Class Software*
Juanita Ewing, *ParcPlace-Digital*
Bob Hinkle, *Consultant*
Tim Howard, *FH Protocol, Inc.*
Ralph E. Johnson, *University of Illinois*
Alan Knight, *The Object People*
Mark Lorenz, *Hatteras Software, Inc.*
Jan Steinman, *Bytesmiths*
Rebecca Wirfs-Brock, *ParcPlace-Digital*
Barbara Yates, *Bytesmiths*

SIGS Publications Group, Inc.

Richard P. Friedman, Founder, President, and CEO
Hal Avery, Group Publisher

Editorial/Production

Elizabeth A. Upp, Managing Editor
Elisa Varian, Production Manager
Andrea Cammarata, Art Director
Sue Mycka, Desktop Designer
Margaret Conti, Advertising Production Coordinator
Shannon Smith, Editorial Production Assistant

Circulation

Elayne Glick, Circulation Manager
Lawrence E. Hoffer, Marketing Manager

Advertising/Marketing

Gary Portie, Advertising Manager, East Coast/Canada/Europe
Michael W. Peck, Advertising Representative
Kristine Viksnins, West Coast Exhibit Sales
Sarah Olszewski, East Coast Exhibit Sales
212.242.7447 (v), 212.242.7574 (f)
Diane Fuller & Associates, Sales Representative, West Coast
408.255.2991 (v), 408.255.2992 (f)
Wendy Dinbokowitz, Promotions Manager for Magazines

Administration

Margherita R. Monck, General Manager
David Chatterpaul, Senior Accounting Manager
Bibi Budhram, Accounts Payable



Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, THE SMALLTALK REPORT, THE X JOURNAL, REPORT ON OBJECT ANALYSIS & DESIGN, CROSS-PLATFORM STRATEGIES, OBJECT CURRENTS, OBJECT EXPERT (UK), and OBJECT SPEKTRUM (GERMANY)

Features

A hierarchy that acts like a class

4

Bobby Woolf

An experience report is presented on a simple hierarchy that embodies a number of powerful O-O technologies. How these techniques are successful applications of common design patterns and pattern languages is examined.

Taking out the garbage

11

Derek Williams

Some tools and techniques for detecting, diagnosing, treating, and, most important, avoiding memory size problems caused by "dangling instances."

Columns



Getting Real

15

The three-tier architecture and server Smalltalk *Jay Almarode*

Partitioning Smalltalk applications between clients and the server (the three-tier architecture) overcomes performance bottlenecks and allows the implementation of shared business objects in a server Smalltalk environment.



Managing Objects

17

A case for open development environments *Jan Steinman and Barbara Yates*

The Smalltalk market is maturing, and Smalltalk vendors envy the relative ease of supporting shrinkwrap compilers and libraries. A long tradition of access to Smalltalk source code may be disappearing before our eyes.



Smalltalk Idioms

21

Farewell and a wood pile *by Kent Beck*

The author presents a frequently successful problem-solving technique and a system development approach parable.

Departments

Editors' Corner

2

Recruitment

26

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar-Apr, July-Aug, and Nov-Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1996 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Second Class Postage Pending at NY, NY and additional Mailing offices. Canada Post International Publications Mail Product Sales Agreement No. 290386.

Individual Subscription rates 1 year (9 issues): domestic \$89; Mexico and Canada \$114, Foreign \$129; Institutional/Library rates: domestic \$199, Canada & Mexico \$224, Foreign \$239. To submit articles, please send electronic files on disk to the Editors at 885 Meadows Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

POSTMASTER: Send domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sigs.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

A hierarchy that acts like a class

Bobby Woolf

WHILE RECENTLY CONSULTING for a client, I developed a miniframework that incorporated several design patterns. These patterns combined to make a rather complex hierarchy of subtypes act like a single class. Besides providing useful functionality that the client and their customers needed (which never hurts!), the framework successfully demonstrated the following techniques:

- what I call “how to do a case statement in Smalltalk,” the Objects from States pattern¹
- an example of the Reusability through Self-Encapsulation pattern language²
- a variation of the Factory Method pattern³
- an example of what I call the “Null Object” pattern, also known as NoWorker⁴ and Null Representation⁵
- limited amounts of visual behavior in domain objects

The framework itself is an example of a case statement using Self-Encapsulation. In it are a set of Factory Methods and a Null Object. Even though it is a framework of domain objects, it still contains some application model behavior used to display the objects in a view. Not only does the framework show simple examples of these patterns, but it also shows how to combine individual patterns together to solve more complex problems.

THE PROBLEM

Part of this client’s system consisted of a questionnaire the user could display so that he could answer it. Analysis revealed that the domain objects were Questionnaire, which contained a list of Questions, each of which had exactly one Answer. Further requirements gathering discovered that there were three different ways a user could answer a question: most questions needed yes/no answers; some needed one of a list of possible answers; others needed freeform text answers. The view could easily indicate these different ways of answering using check boxes, combo boxes, and input field widgets, respectively.

Most of the domain code had already been implemented for me. There were already Questionnaire, Question, and Answer classes, and the container relationships between the three had been defined. The problem was the difficulty supporting the three different ways of answering and the three different kinds of visual widgets used to input and display answers.

The solution that had been implemented required a lot of fairly ugly code in the Answer class. The class had

three instance variables: yesOrNo, selection, and responseText. Each instance only used one of those variables; the other two were always nil. The displayString method printed-out each variable’s value as long as it wasn’t nil; because only one variable was not nil at any given time, that’s the only one that was printed. What the implementation did not solve was displaying a particular Answer as different widgets. When I started on the project, testing methods like isYesNo, isSelection, and isText were about to be introduced.

SMALLTALK CASE STATEMENT

Obviously Answer was becoming too complex. The solution I devised was to expand the Answer class into a hierarchy of classes:

```
Answer ()
  BooleanAnswer (yesOrNo)
  EnumerationAnswer (responseChoices, responseIndex)
  TextAnswer (responseText)
```

This way each question could have the appropriate type of answer: boolean, symbol, or text. None of the Answer objects wasted any instance variables. Each class knew what its instance variables’ types were and how to handle them. This factored the complexity of handling these different possibilities into separate classes so that the decisions each class had to make were actually quite simple.

The reason I call this hierarchy an example of a Smalltalk case statement is that it eliminates the need for testing methods like isYesNo, isSelection, and isText. An example of a case statement method would be something like

```
Answer>>visualWidget
  self isYesNo ifTrue: ["Use a check box."].
  self isSelection ifTrue: ["Use a combo box."].
  self isText ifTrue: ["Use an input field."].
```

This is poor object-oriented (O-O) style. Sometimes developers clamor for Smalltalk to have a case statement, usually because they’re trying to write code like this. Although all of us write code like this sometimes, it is best avoided. Factoring the class into a hierarchy allowed me to eliminate the testing methods and simplify the code like this:

```

Answer>>visualWidget
  ^self subclassResponsibility
BooleanAnswer>>visualWidget
  "Use a check box."

EnumerationAnswer>>visualWidget
  "Use a combo box."

TextAnswer>>visualWidget
  "Use an input field."

```

This is a case statement via polymorphism and inheritance. Just send the message and whatever implementer gets run is the correct case. This forces the “testing” to be encapsulated within the hierarchy where it can easily be reused. If the results of the testing need to be changed, the code is easier to maintain because it is so cleanly encapsulated. Finally, the differences between the peer classes are easy to see; just look at the methods they implement instead of inheriting.

This case statement framework is also easy to extend to add new cases. For an example of this, see the “Null Object” section of this article.

SELF-ENCAPSULATION

A problem this hierarchy introduced is that now each particular Question instance had to know what kind of Answer instance it had. To avoid this problem, I wanted all of the concrete classes to be polymorphically equivalent. This means that they would all have the same common interface so that I could generally treat any instance as an Answer without regard to which concrete subclass it was.

To do this, I defined the common interface in Answer with messages like response, response:, and displayString. In Answer, each of these methods returned the subclass-Responsibility error. Each subclass implemented the messages appropriately in terms of its state. For example, here’s how EnumerationAnswer handled the response aspect:

```

EnumerationAnswer>>response
  | index |
  index := self responseIndex.
  ^index == 0
    ifTrue: [nil]
    ifFalse: [self responseChoices at: index]

EnumerationAnswer>>response: newResponse
  self responseIndex:
    (self responseChoices
     identityIndexOf: newResponse)

```

This is an example of the Reusability through Self-Encapsulation pattern language, albeit an extremely simple example. The language shows how to implement an extensible yet well-encapsulated hierarchy. An abstract superclass defines the interface the subclasses will follow. The interface is implemented in terms of a small number of kernel methods that each subclass must implement appropriately. As the language suggests, Answer defines

the hierarchy’s interface and leaves the implementation details to the subclasses.

This could be considered a variation of the Factor a Superclass pattern.⁶ That pattern starts with a number of peer classes and factors their shared variables and behavior into a common superclass so that the subclasses do not duplicate each other’s variables and behavior. What I did with Answer was the same process in reverse; I started with one class and factored it into many subclasses. I use the superclass to define a common interface rather than implement common behavior, so my efforts are more reminiscent of Self-Encapsulation than Factor a Superclass.

FACTORY METHOD

Another problem the Answer hierarchy introduced is the matter of assuring that the right kind of Answer instance gets assigned to each Question. How does a Question phrased as a yes/no question get a BooleanAnswer? How does one with a list of possible answers specify that it needs not only an EnumerationAnswer but the list of choices as well?

To solve this problem, I introduced the following messages into Question: useYesNoAnswer, usePossibleAnswers:, and useTextAnswer. This way, as each Question was created, the answer details could be specified as well. Here are some examples:

```

question1 := (Question text: 'Do chickens have lips?')
              useYesNoAnswer.
question2 := (Question text: 'Are you lazy?')
              usePossibleAnswers:
                #(#always #sometimes #never).
question3 := (Question text: 'How old are you?')
              useTextAnswer.

```

I wanted to hide the complexity of the Answer hierarchy and maintain the illusion that it was still just one class. This way the one class could actually manage the other classes and their use. This will also encapsulate this management within the class. Because this is not a complicated hierarchy, its management is fairly simple. Answer’s instance creation protocol just has to allow for creating each kind of answer. Here are the methods that do this:

```

Answer class>>yesNoAnswer
  ^BooleanAnswer new

Answer class>>possibleAnswers: answerList
  ^EnumerationAnswer new responseChoices: answerList

Answer class>>textAnswer
  ^TextAnswer new

```

Question in turn just delegates the Answer creation to that class:

```

Question>>useYesNoAnswer
  self answer: Answer yesNoAnswer
Question>>usePossibleAnswers: answerList

```

```
self answer: (Answer possibleAnswers: answerList)
```

```
Question>>useTextAnswer
self answer: Answer textAnswer
```

The three instance creation methods in Answer are examples of the Factory Method pattern, or at least a variation thereof. Gamma et al. say that the “Factory Method lets a class defer instantiation to subclasses.” A classic example in Smalltalk-80 is the way View defines the method `defaultControllerClass`. Each subclass of View subimplements `defaultControllerClass` to return the class for its controller. Thus `defaultControllerClass` is a Factory Method.

The Factory Methods in Answer are `yesNoAnswer`, `possibleAnswers:`, and `textAnswer`. Because they are not standard protocol that is overridden in subclasses, they are not standard Factory Method examples. However, they are a variation on the same theme because they use message sends to hide the existence of the various Answer subclasses, as well as their names and interfaces. As far as a collaborator like Question is concerned, there is only one Answer class (not a hierarchy) and it is able to act in these various ways. This encapsulates the hierarchy and simplifies its interface to the rest of the system.

I cannot claim to have invented this technique. In VisualWorks, `Filename` uses it to determine which of its subclasses to use. Similarly, `CompositePart` uses it to determine which Wrapper class to use.

Alternate solution: Question hierarchy

When analyzing the requirements and designing a solution, I considered developing a Question hierarchy as well as an Answer hierarchy. This would have incorporated Factory Method more directly by using subclassing (as the pattern suggests). The Question class would have defined a method like `defaultAnswer` in terms of `defaultAnswerClass`. Then Question subclasses would override `defaultAnswerClass` to return the appropriate Answer subclass.

For example:

```
Question>>defaultAnswer
^self defaultAnswerClass new

Question>>defaultAnswerClass
^Answer

BooleanQuestion>>defaultAnswerClass
^BooleanAnswer
```

and so forth for `EnumerationQuestion>>defaultAnswerClass` and `TextQuestion>>defaultAnswerClass`. In fact, I did implement `Question>>defaultAnswerClass` in preparation for such a protocol.

This combining of dual hierarchies is an example of the Bridge pattern, where an abstraction is decoupled from its implementation by implementing it in two hierarchies. The two hierarchies can be extended independently, and because they are connected by a standard interface, most any pair of instances from the two hierarchies can work together.³

The problem with Question and Answer hierarchies is that the classes are not decoupled from each other. There is a one-to-one correspondence between the classes in the two hierarchies: `BooleanQuestion/BooleanAnswer`, `EnumerationQuestion/EnumerationAnswer`, and `TextQuestion/TextAnswer`. Anytime a new class was added to one hierarchy, a corresponding class just like it would need to be added to the other hierarchy, e.g., `RangeQuestion` would require `SliderAnswer`. So these hierarchies are not truly decoupled; in fact, they require duplicate effort to extend both hierarchies.

Another problem with the Question hierarchy is that subclasses would not have behaved differently from their superclass. All Questions were essentially the same, even though they expected different types of answers. Hopefully, those that claimed to need a yes/no answer were phrased as a yes/no question, but there was no way to enforce this in `BooleanQuestion`.

In the end, although an Answer hierarchy looked promising, a similar Question hierarchy not only wasn't helpful, but was in fact counterproductive. The Question subclasses would not have introduced any helpful behavior but would have required duplicate effort. Thus a Question hierarchy was not necessary.

Tangent topic: SelectionState class?

Kent Beck recently wrote an excellent column, “Clean code: Pipe dream or state of mind?”⁷ In it, he describes how he factored out a State Object using two classes, `SingleSelectionState` and `GroupSelectionState`, a terrific solution to the problem he was facing. He might have taken the solution one step further by using the Factory Method variation described here. Using it, he would introduce an abstract class, `SelectionState`.

The `SelectionState` class would define the interface for all `SelectionState` instances (`SingleSelectionState`, `GroupSelectionState`, and eventually `DelegationSelectionState`). It would also serve as the hierarchy's interface to the rest of the system (collaborators such as `SelectionTool`). Then methods like `SelectionTool>>setSelectionState:` could be moved into `SelectionState` (probably as `SelectionState>>setSelectionState: aFigure`). `setSelectionState:` is a fairly messy method that must contain a lot of knowledge about the classes in the `SelectionState` hierarchy. Notice that when Kent introduced an additional class, `DelegationSelectionState`, he had to rewrite this method. This messiness indicates that the method should be encapsulated within the hierarchy, which moving it to `SelectionState` would do. Also, if `setSelectionState:` were ever needed by another collaborator that was not a `SelectionTool`, the method would be available for reuse.

NULL OBJECT

Although requirements gathering discovered fairly early on that there were three types of answers—yes/no, list of choices, and freeform text—we discovered later that there was actually a fourth, hidden case to be considered. Some “questions” in the questionnaire were actually not ques-

tions per se, but headings for subsequent questions. Such a question might be “Check each of the following traits that describes you:”

Why not Heading?

The problem this introduces is that such a heading is more of a Heading object than a Question object. Both have text, but headings don't have answers the way questions do. But then how should this be displayed in a view?

The questionnaire was displayed as a table with two main columns, question and answer. Every row was expected to have two aspects that would be displayed in the two columns. Thus Heading needed to have an answer aspect just like Question, and the heading's answer would need to be able to display itself the way an Answer can.

This caused Heading to work just like a Question, so I found no need for a separate Heading class. This solution may be an example of improperly letting the view define the domain; iterating over the design might produce a better one. Yet I feel that the solution described below turned out pretty well and may in fact be the most graceful way to hide the exceptions to some otherwise simple and uniform rules.

Perhaps another reason I rolled Heading into the Question class is that my deadline for completing this subsystem was rapidly approaching. It's funny how when I'm near a deadline, the current design I've already implemented can look much better than an alternative that requires rewriting a lot of code!

A heading's answer

Modeling the heading as a Question object, it had to have an Answer, but none of the three Answer subclasses applied. For this purpose, I developed a fourth Answer subclass called NullAnswer:

```
Answer ()
  NullAnswer ()
```

As a subclass of Answer, NullAnswer preserved the Answer interface, but did so without doing anything. Here are some examples of the methods it defined:

```
NullAnswer>>response
  ^nil
```

```
NullAnswer>>displayString
  ^'n/a'
```

NullAnswer is an example of what I call the Null Object pattern. I haven't seen documentation for this pattern published anywhere, but it is discussed fairly often. The pattern describes an object that shares the same interface as others of its type but that reacts to these methods by doing nothing. The trick is in designing, for each message, what

doing nothing means. Typically it means getters that return nil or empty collections and display methods that show the object as null. Setter methods are usually ignored; they can create a real instance and substitute it for the Null Object, but this is more the behavior of a Proxy³ than a Null Object.

The beauty of a Null Object is that it supports an extensive, customized interface and encapsulates the decisions about how it should “do nothing.” nil is perhaps the most famous Null Object, but it doesn't really count because its interface is neither extensive nor customized. Yet programmers often use nil in a variable that hasn't been assigned yet. This leads to copious amounts of code that constantly check the variable for nil before sending it messages. This code can be simplified by assigning the variable a Null Object of the correct type and then sending the variable messages with impunity. Also, rather than each collaborator deciding what to do when the variable is nil, these decisions are encapsulated within the Null Object for reuse and consistency across all collaborators.

Adding NullAnswer

The infrastructure to support the new NullAnswer class was easy to introduce because the hierarchy was well encapsulated. It consisted of exactly one method:

```
Answer class>>nullAnswer
  ^NullAnswer new
```

Then collaborators, such as Question, just needed to tie into the expanded interface in a convenient way:

```
Question>>useNullAnswer
  self answer: Answer nullAnswer
```

Other collaborators could tie in just as easily.

VISUAL CODE IN DOMAIN OBJECTS

One of the distinguishing factors of the different types of answers is the way they were to be displayed. As described earlier, a question could be displayed as a check box, a combo box, or an input field. Also, a null answer would need to be displayed with a “do nothing” widget.

Much has been written recently about the importance of separating domain and application behavior, including by me.⁸ Basically, domain objects represent core business behavior, while application objects know how to display domain objects in useful ways. This is the basic architecture I follow for all of my development, including the Questionnaire framework.

Questionnaire is the root of the domain framework described earlier. In turn, I also implemented a corresponding QuestionnaireUI class to represent Questionnaires. (Because I was developing in VisualWorks, QuestionnaireUI

*A Null Object supports
an extensive, customized
interface while
encapsulating how to
“do nothing.”*

was a subclass of `ApplicationModel`.) `QuestionnaireUI` was essentially a glorified table sort of widget where each row displayed a question. The table had two main columns, the question text and the question answer.

Why not AnswerUI?

The strength and limitation of this simple design are that there were no application models for the Questions and their Answers. I did not want these application models because the `QuestionnaireUI` itself was just a table, practically a `TableView` or a `DataSetView` (in `VisualWorks`). Just as those classes don't contain separate "RowView" and "CellView" classes, I didn't want `QuestionnaireUI` to contain numerous `QuestionUI` objects. Those would do nothing more than contain `QuestionTextUI` and `AnswerUI` objects, each of which would do little except to contain a single view such as `InputFieldView` or `ComboBoxView`. This seemed to me like an explosion of custom classes and do-practically-nothing objects, a complication I wanted to avoid. Once again, avoiding these classes appears to be the most graceful way to hide complicated exceptions to otherwise simple, uniform rules.

Had there been an `AnswerUI` class, it could have made the decision as to what kind of widget to use to display each kind of Answer (its domain model). Actually, this might have necessitated the need for a separate `AnswerUI` subclass for each Answer subclass. Then the `AnswerUI` hierarchy would have been tightly bound to the Answer hierarchy and duplicate effort would have been required to extend both in tandem. In any event, I did not have any `AnswerUI` class available. All I had was a cell in a table that was supposed to display an answer and an Answer domain object that contained the data for that cell.

How to display an Answer

Because the Questions and Answers did not have their own application model counterparts, there was no obvious place to put the code that decided how to display the different types of Answers. The way I solved this problem was to have the cell ask the domain object what widget should be used to display it. The domain object would return the widget and the table would display that widget in the corresponding cell.

This necessitated introducing the message `visualWidget` into the Answer hierarchy (as shown earlier). Each subclass would return an instance of the widget appropriate for itself. Thus `visualWidget` is another example of the Factory Method pattern, a more accurate example, since subclasses override the superimplementer.

Adding an application layer method like `visualWidget` into a domain layer object like Answer is certainly unusual, but not necessarily wrong. It was, after all, a single method, not a whole suite of behaviors that could easily become indistinguishable and inseparable from the domain behavior. Furthermore, its behavior is likely to be appropriate for any application layer that might be built on this domain, so there is little need to be able to swap one application object in for another.

Finally, I was still able to distinguish the application code from the domain code in this domain class using ENVY (ENVY/Developer, Object Technology International, Inc.). I defined the Answer hierarchy in the Domain application. Then I extended each of the hierarchy's classes in the UI application to add the `visualWidget` method. This not only clarified which Answer methods were for UI behavior, but it also meant that a developer could easily unload all UI code from the image—even that which the domain classes contained—by unloading the UI application.

CONCLUSION

One simple class in one minor part of a system turned out to employ a number of powerful O-O techniques. Answer was conceptually an uncomplicated little class that turned out to have multiple personalities. As the code to support those personalities grew, the need to expand the class into a hierarchy became apparent. But because the complexity of the hierarchy distracted from the simplicity of the class, the need to hide this complexity became apparent as well. I was able to develop this complex hierarchy with a simple, single class-like interface by using and combining the following techniques:

- **Smalltalk Case Statement:** This is what led to the hierarchy. Each case was represented as a separate class.
- **Self-Encapsulation:** This is what led to the abstract superclass. It defined the public interface that all subclasses would support so that various instances could be treated polymorphically.
- **Factory Method:** This hid the concrete subclasses so that they were never referenced from outside the hierarchy. The hierarchy's collaborators interfaced with the abstract superclass, telling it what behavior was expected from a new instance and trusting the superclass to return an appropriate instance.
- **Null Object:** This substituted as the answer for a question that did not need an answer. It supported the abstract superclass's interface and thus could be used just like any other concrete subclass. And it encapsulated the "do nothing" code so that all questions without answers would behave the same.
- **Visual Code in Domain Objects:** This acted as the application object for a domain object whose display was so simple that it did not need a separate application object. ENVY extensions demonstrated that separate objects are not the only way to separate independent layers of code.

I hope this successfully illustrates these techniques and shows how they may be used to solve real-world problems. I feel it is important not only that we document these techniques as reusable design patterns and pattern languages, but also that we show how they can be applied in practice to help develop better-quality software. I hope this experience report will prove useful to you. Please feel free to contact me at woolf@acm.org if you have any (tastefully phrased) questions or comments. ☐

References

1. Beck, K. Death to case statements, part 2, THE SMALLTALK REPORT 3(4), 1994.
2. Auer, K. Reusability through self-encapsulation, Coplien, J.O. and D.C. Schmidt, Eds., PATTERN LANGUAGES OF PROGRAM DESIGN, Addison-Wesley, Reading, MA, 1995.
3. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1995.
4. Hendley, G. The NoWorker pattern, available from the author at ghendley@ksccary.com
5. Carlini, G. Type and implementation, available from the author at giuliano@filenet.com
6. Beck, K. Inheritance: The rest of the story, THE SMALLTALK REPORT 3(1), 1993.
7. Beck, K. Clean code: Pipe dream or state of mind? THE SMALLTALK REPORT 4(8):20–22, 1995.
8. Woolf, B. Making MVC code more reusable, THE SMALLTALK REPORT 4(4):15–18, 1995.

Bobby Woolf is a Member of Technical Staff at Knowledge Systems Corp. in Cary, North Carolina. He is actively engaged in the patterns movement that is seeking to document common software development techniques. Comments are welcome at woolf@acm.org.

Taking out the garbage

Derek Williams

"One person's trash is another person's treasure."

"Memory leak!": it's a scary phrase, yet so many object-oriented programmers use it. In a general sense, it means that the memory size of a running program continually grows so less room is available to create new objects. The cause is usually *dangling instances*: objects that are no longer needed but are still around, using valuable memory space. For C++, dangling instances usually mean that the programmer forgot to explicitly delete objects or failed to follow the protocols for who is responsible for deleting. For Smalltalk, dangling instances are usually objects that are considered trash to the programmer but treasure to the garbage collectors.

Thanks to the garbage collectors in nearly all Smalltalk implementations, Smalltalk programmers aren't burdened with having to explicitly delete objects: they simply create new objects as needed and let the garbage collectors remove obsolete objects and reclaim the space. The garbage collectors deem an object to be obsolete if it is no longer referenced (directly or transitively) from the "root" pointers that keep immortal objects around. But sometimes, programming errors can leave unintended, dangling references to an object, causing it to live much longer than it should.

In this article I introduce some common tools and techniques for detecting, diagnosing, and treating dangling instances problems. It is slanted toward the VisualWorks environment and its memory management architecture, but many of the concepts apply to other Smalltalk dialects.

DETECTION: DO YOU HAVE MEMORY PROBLEM?

You might have a memory problem if:

1. Your cursor frequently changes to a garbage collection or compaction cursor.
2. Your development image file grows progressively larger each time you save it, especially if you feel you've done nothing that would cause it to be larger.
3. The memory footprint of your image grows to be much larger than you think it should be. You might notice that less space is available to other programs or you might detect it with your finger on the *dynamically allocated footprint* pulse of Smalltalk.
4. You notice heavy thrashing: lots of disk I/O due to the

operating system constantly swapping in and out the memory allocated for use by Smalltalk.

5. You see a "low space notifier" or "out of memory" error message.

These types of errors usually mean that memory problems have gotten out of hand. We'll look at some steps you can take to avoid getting to this point, but first let's see how each of these can occur. To understand these effects, it helps to know a little about how your Smalltalk virtual machine (a.k.a. object engine) manages memory.

The Smalltalk object engine divides the memory it uses to store objects into a number of separate regions or *spaces*. It does this to get the optimum benefit from different garbage collection schemes. For example, newly created objects that are less than 1 KB in size are stored into the Eden *subspace* in NewSpace. NewSpace is managed by the scavenger garbage collector, which uses a two-space copying algorithm. The scavenger runs as a background process, alternately copying surviving objects between Eden and the two SurvivorSpaces. The two-space algorithm works especially well for NewSpace, since most new Smalltalk objects live very short lives and can be easily discarded simply by not copying them forward.

Objects that survive NewSpace are *tenured* into OldSpace, which is managed by an incremental mark-and-sweep garbage collector. OldSpace is unique in that it dynamically grows as needed to accommodate the "mature" objects in the system. All other spaces are fixed in size when the memory policy is installed (typically at image startup).

You can control the balance between collecting garbage and thus reclaiming space and growing the size of OldSpace by changing memory policy parameters. When a request is made to allocate a new object and there simply isn't room for it, we say a *low space condition* has occurred. It is then up to the memory policy to decide what to do: to try to reclaim space by aggressively collecting garbage and compacting objects or to simply ask for more memory from the operating system. You can set a cap at which reclamation will be favored over growth and even limit the total memory size of the image.

Much of the partitioning of objects into spaces is done "under the covers" and not directly visible to your Smalltalk code. For example, you cannot find out which space a given object resides in or find all the objects in a particular space. But the parameters that control the

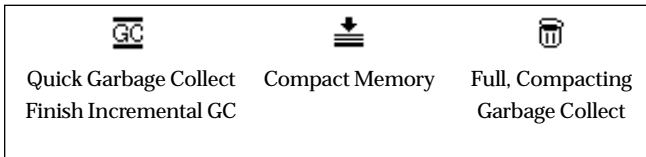


Figure 1. Cursors indicating you are running out of space.

memory management behaviors are available to your Smalltalk code and you can modify them. For example, you can set the space sizes based on your application needs, check various garbage collection statistics, check on the current size of OldSpace, explicitly invoke a garbage collector, etc. There's a lot more that can be said about how the garbage collectors work and how to tailor and tune the memory policy for your needs—that's not the purpose of this article. But you can start learning by reading the class comments and documentation methods in the classes `ObjectMemory` and `MemoryPolicy` or reading the *Memory Management* chapter in the *VISUALWORKS USER'S GUIDE*.¹ You should also read Kent Beck's article on garbage collection in the February 1995 issue of this publication. It explains how the garbage collection algorithms work in detail and covers the Visual Smalltalk environment.

Although we're not going to delve further into the object engine's memory management work, we can use our basic knowledge of how it operates to help detect dangling instance problems. Let's review those five "warning signs" and look at how each could occur.

Frequent cursor changes

The `MemoryPolicy` (through services in `ObjectMemory`) displays special cursors to show you when incremental garbage collection and compaction activities are occurring. Since these activities are typically in response to low space conditions, they provide a visual clue to the state of memory. Frequent collection or compaction cursors are often early indicators that you are running out of space. Figure 1 shows what these cursors look like.

Large image file size

Most Smalltalk applications are coded and unit tested using development images that are frequently saved to disk and then packaged into runtime images for further testing and deployment. Usually the runtime images are delivered to users who load, use, and exit them as needed, but never save them again.

Some memory leaks can go undetected in runtime images because the user starts at square one each time he or she reloads the image. Since each image save writes all the objects to disk, dangling instances have a way of stacking up in a development environment. During development, it's a good idea to occasionally look at the size of your image file. If it grows larger than you would expect, you have an early indicator of a potential memory problem.

Large memory footprint

You don't have to wait until you save your image to a file to determine its size. Sending `ObjectMemory`

`dynamicallyAllocatedFootprint` will answer the total number of bytes of memory your image is currently using. You can send this message as often as you like (such as before and after testing an application scenario) to gather measurements.

There are other services on the `ObjectMemory` class to give you a view of the current sizes and state of memory. For example, since only `OldSpace` will grow in size, you may only be interested in `oldBytes`, rather than the total size.

Thrashing by the operating system

Recall that the image will grow in size until it reaches the limit you set or until the operating system refuses to give more memory to Smalltalk. You should set a cap so that Smalltalk is a good citizen and leaves plenty of room for other programs to run. If you don't and the image grows too large, switching back and forth between Smalltalk and other programs can lead to heavy swapping.

When running under MS Windows, you should be aware that Windows will often politely let Smalltalk have so much memory that it doesn't keep enough space for a good working set of its own. When this happens, even a basic operation like opening a new window can cause swapping.

Because thrashing is never a good thing, and because the object engine never gives space back to the operating system until you exit, you should choose your cap carefully. If you would like to start favoring reclamation over growth at *x* bytes and you never want your image to be larger than *y* bytes, you can set a fixed value with something like:

```
ObjectMemory installMemoryPolicy:
    (MemoryPolicy new
     setDefaults;
     growthRegimeUpperBound: x;
     memoryUpperBound: y;
     yourself).
```

Or you can get information about available or installed memory from the operating system and use this as a basis for setting a cap or controlling your own custom memory policy. Also, the Runtime Packager tool has a window you can use to set memory sizes when you build a runtime image.

Low space notifier

If things are really bad, i.e., you've run out of room for new objects, garbage collections do not reclaim enough space, and the image cannot grow any more, you may see a *low space notifier*.

The `MemoryPolicy` has no direct way of communicating with the user to report that space is running dangerously low. So, it invokes the low space notifier via the user interrupt signal when it needs to say "Emergency: No SpaceLeft" or "Space warning."

The user interrupt signal is the same mechanism used to invoke the emergency evaluator when Ctrl+Shift+C is pressed. If you've tried to disable the emergency evaluator for a runtime image or change the way it displays, you should make sure that you're not masking the low space

notifier. I've seen cases where low space conditions were simply reported as "user interrupts" and the end user had no idea what was happening.

DIAGNOSIS: FINDING THE CAUSE

So, based on the warning signs described above, you think you have a memory problem—now what? We want to find exactly which objects are not being cleaned up and why.

A quick scan of instance counts can help you find the dangling instances and narrow the set of classes to examine for potential problems. A code snippet like the one in Listing 1 can help.

You're now inspecting a dictionary that shows you counts for all classes having greater than 200 instances floating around. Choosing 200 as a cutoff is arbitrary—substitute whatever works for the situation or create your own list of classes to check. If you want to narrow the search, you can replace `Object` with another parent class. Once you have the inspector, you'll probably want to look further, e.g., you may want to sort by class name:

```
self associations asSortedCollection:
  [ :a :b | a key name < b key name ]
```

or by number of instances:

```
self associations asSortedCollection:
  [ :a :b | a value > b value ].
```

Once you've targeted a class that appears to have more instances than it should, send `allInstances` or `allInstancesWeakly`: and inspect the result.

This is a rather brute-force approach to tracking down runaway instances, but it's often all you need. You can wait until runaway instances start to get out of hand, interrupt your code if necessary, and run this snippet. For example, if you see 1,000 instances of your `Scooter` class and you were expecting only two or three, you have a good place to start.

If you can run through an application scenario to consistently create the problem or if you don't know whether or not you have a problem, then check at regular intervals. Periodic measurements taken with the code snippet in Listing 1 and displaying the value of `ObjectMemory dynamicallyAllocatedFootprint` can tell a lot.

It's a good idea to include some lightweight memory diagnostics like the above even in a runtime image you deliver to customers. For example, you might add a window somewhat off the beaten path to display footprint sizes or instance counts on demand. These snippets add very little to the size of a runtime image, and you may just find that your customers can create memory problems you never expected (e.g., by leaving an image running steadily for several weeks at a time).

The `AllocationProfiler` in `Advanced Tools` can show you which new objects a particular block of code allocates by

Listing 1.

```
| dict |
"Faster than sending instanceCount to all classes"
ObjectMemory garbageCollect.
dict := IdentityDictionary new.
Object allSubclasses
  do: [ :ea | ea isMeta
      ifFalse: [ dict at: ea put: 0 ] ].

ObjectMemory allObjectsDo:
  [ :ea | | count | (count := dict at: ea class ifAbsent: []) notNil
      ifTrue: [ dict at: ea class put: count + 1 ] ].
(dict reject: [ :ea | ea value < 200 ]) inspect.
```

tracking calls to methods that create new objects. Since it shows you only memory allocation and not reclamation, be prepared to look through the complete picture (remember, most new objects die quickly). But it is an easy tool to use and gives detailed information. To use it, simply send `AllocationProfiler profile:` and pass it a block to measure. You can learn more about the `AllocationProfiler` by reading the `ADVANCED TOOLS USER'S GUIDE`.²

Finally, I've implemented some of the above techniques in the `Memory Diagnostics` tool, which you can get from the University of Illinois Smalltalk archives (<http://st-www.cs.uiuc.edu/>).

TREATMENT: FINDING AND CLEANING UP REFERENCES

The measurements you took above should tell you at least two things: (1) what the dangling instances are and (2) what application scenario creates them. But often you need to look further: you want to know exactly where the dangling references are coming from.

You can determine this by inspecting one of the dangling instances and looking at the reference path to it. There are several ways to follow reference paths:

1. Manually follow the path of references by sending `allOwners` or `allOwnersWeakly`: and inspecting the result. This will show you the immediate references to your dangling object. Sending `allOwners` or `allOwnersWeakly`: to each of these references will show you the next level. You can continue this process until you start to see objects or methods that point to potential problems.
2. Use the `ReferencePathCollector` in `Advanced Tools`. If you're inspecting one of your dangling instances, you can send `ReferencePathCollector allReferencePathsTo: self` and inspect the result. Read the comments for class `ReferencePathCollector` for more information.
3. Use the `PointerFinder` tool written by Hans-Martin Mosner. You can get it from the author's web page at <http://donald.heeg.de/pub/hmm-goodies/>.

The reference path will often provide its own clues to exactly which portion of code caused the dangling instance and why it is not being cleaned up. Since it helps to know what to look for, here are some common causes.

1. Unbroken dependencies. In many cases, dangling

instances are due to one object being referenced as a dependent of another object that is still in use. The dependent object may no longer be needed, but it isn't collected because it is still referenced by the "parent" object. Usually this is caused by a failure to send `removeDependent:` or one of the related methods. Keeping track of all the places where dependencies are set and broken can sometimes be difficult, given all the different layers and frameworks that use them (dependency transformers, adapters, value models, etc.) and all the message variants.

The unbroken dependency problem is much easier to create if you add dependents to an object that does not track them in an instance variable (e.g., it does not override `Object>>myDependents`). In this case, the dependency connection is kept in the global `DependentsFields` dictionary. With "local" dependents, a failure to break dependencies will be cleaned up when the parent is no longer referenced. But when the dependencies are tracked in the `DependentsFields` dictionary, you have a new pair of references that will keep parent and dependent around. So, inspecting `DependentsFields` is often a good way to look for problems.

2. Overlooked object references. Dependency connections certainly aren't the only common ways that objects reference each other. Indeed, object connections are at the very heart of object design—those associations and aggregations we like so much. You may need to clean up some of your own object references through a release, finalization, or similar protocol. For example, you can use the release event for `ApplicationModel` classes to "nil out" or otherwise clean up references to objects referenced by instance variables.

It's easy to forget about object caches held onto by class variables and class instance variables. While such caches are nice for boosting performance, don't forget about them. You may want to implement and send class-side "uninitialize" methods to clear out caches when necessary. If you're using ENVY, you may find yourself doing this as part of *removing* methods to unload an application.

3. Failure to copy. So many collection operations answer copies that we sometimes take it for granted and assume we always have either a shallow copy of a collection or a deep copy of the collection and its contents. This assumption can be dangerous and can not only indirectly be a source of dangling references, but also lead to other errors such as one client of a collec-

tion modifying its contents and affecting others. This is, by the way, why a method answering a literal string that is coded in it is generally a bad idea.

How you manage references and copies really depends on what you are trying to do, so it's hard to give general rules. But if you suspect a problem caused by a reference to a shared object rather than a copy, you can use the identity comparison (`==`) or compare object identifiers to see if the references really are to the same object. To see the object identifier, send `asOop` to the object and print or inspect the result.

Finally, now that you've found the dangling instances and cleaned up the cause, what do you do with all those "zombies" floating around?

It's best to start with a clean image and load your code into it. But if you're fond of your current image and want to keep it, you'll need to do your own clean up. When cleaning up an image, track back to the root cause and correct it. For example, you may have to remove dependency connections from an inspector on `DependentsFields`.

I often hear the suggestion to use `become: String new` to "morph" a dangling object so that it loses its instance variable links. Using `become:` should always be a last result and done with great care. And you should keep in mind how your Smalltalk implements it—whether it swaps pointers or copies state.

CONCLUSION

Now that you have the fear of runaway memory problems, take comfort: it's usually a rare occurrence. The garbage collectors do an amazing job of managing memory efficiently and the class libraries are tolerant of potential errors. Only rarely do I have to pull out this bag of tricks to help someone diagnose a memory problem. But by using some of these techniques, you'll have the diagnostics to easily watch for problems and, when you find one, the tools to track it down and fix it. ☞

References

1. ParcPlace Systems. *VISUALWORKS USER'S GUIDE*, Sunnyvale, CA, 1994.
2. ParcPlace Systems. *ADVANCED TOOLS USER'S GUIDE*, Sunnyvale, CA, 1994.

Derek Williams has been developing vertical client/server applications for 11 years and using Smalltalk for the past 4 years. He can be reached at derek_wi@hboc.com.

continued on page 32



Jay Almarode

The three-tier architecture and server Smalltalk

IN PREVIOUS COLUMNS, I have described the features that make up multi-user Smalltalk. These features include support for transactions, concurrency control and locking, versioning and instance migration, and security. These are but a few of the features required for a Smalltalk system to function as a server. In addition to these features, a server Smalltalk system must also provide persistence of objects, fault tolerance, and scalability. This column is the first of two that describe how multi-user Smalltalk fits in the emerging 3-tier architecture, and how partitioning Smalltalk applications between clients and the server overcomes performance bottlenecks and allows the implementation of shared business objects in a server Smalltalk environment.

There are three kinds of objects that exist in a typical application: presentation, application, and business objects. Presentation objects are the widgets, forms, and windows that present information to the end user. Application objects are the objects responsible for the sequencing of tasks and the management of how business objects are used by the end user to achieve a specific task. Business objects are general-purpose objects that model the processes and basic concepts of the business. Business objects are a hot topic these days, as companies undergo business process reengineering to better model the basic functions of the company. (See Rymer¹ for a detailed discussion of business objects.) There is even a Business Object Management Special Interest Group that was founded by the OMG.

When Smalltalk is used as the implementation language on the client machine only, the objects that implement the application and business logic, as well as the objects that implement the presentation logic, must all reside in the Smalltalk image on the client machine. Typically, when the application starts up, it connects to either a relational or object database and transfers the object state needed to run the application to the client machine. If the database is relational, the tabular data in the database must be mapped to objects in the image.

The application may take advantage of the query capability of the database to selectively choose which objects are manifested in the client. However, to execute any business logic, the objects must be present in the client Smalltalk image. This is how Smalltalk is used in a pure client/server architecture.

The pure client/server architecture works well on a small scale, but has a number of drawbacks that hinder its ability to implement enterprise-wide, shared business objects. In this architecture, the server does not have the ability to execute complex business logic. The database may provide some query capability or stored procedures, but does not provide an object model or a computationally complete language like Smalltalk. Consequently, to execute any complex business or application logic, much data must be transferred to the client Smalltalk to be turned into objects that can execute behavior. As the number of client workstations increases, the network becomes overloaded. As applications execute more complex business logic, requiring more objects to be transferred to the client, the client machines need more memory and processing power. Increasing network bandwidth and CPU/memory capability for thousands of client workstations can become an expensive proposition.

There are business drawbacks to the pure client/server architecture as well. Transferring sensitive data to the client machine to execute application behavior can pose a security risk. Client machines are inherently insecure, and none of the client Smalltalk systems currently implement security features in the virtual machine. When the business logic is duplicated across thousands of clients, maintenance is expensive, and this discourages frequent updates to the application. The logic and algorithms implemented within business objects are a strategic asset to the company and should be a shared resource under centralized control. For example, consider the algorithm by which a portfolio management application distributes a customer's funds among stocks with different risks. The better that the application can evaluate market conditions and risk, the better the rate of return. The application's risk assessment algorithm needs to be shared by multiple clients, may need to be updated frequently based upon new strategies, and needs to be protected by theft from competitors.

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@slc.com.

To overcome the drawbacks of the client/server architecture, the 3-tier model has emerged. The 3-tier architecture is an evolution of the client/server architecture that defines a middle tier, called an application server, between multiple client workstations and the data server (a relational or legacy database). For a more complete description of the 3-tier architecture, see Lozinski.² The middle tier is where shared business objects are implemented in multi-user Smalltalk. This architecture reduces the amount of data transmitted to the client, because business logic can be executed in Smalltalk on the middle tier rather than transmitted to the client for execution. With this architecture, only a single, high-bandwidth connection from each legacy relational database to the application server is needed, and the relational-to-object mapping is performed in one place, and only when needed. The Smalltalk objects on the middle tier can be thought of as an objectified view of the legacy data, ready for each client when needed. This allows for easier integration of legacy data, live data feeds, or other external data sources into Smalltalk applications because each client only sees objects in the application server. Clients do not have to know where the server objects came from, and are insulated from changes to the source of these objects.

Having the shared Smalltalk business objects in the middle tier also provides a central point of control for updating business logic, defining security policy, and providing fault tolerance of important objects. In the previous example, the risk assessment algorithm of the portfolio management application should be implemented as functionality provided by a business object. This business object is available to clients through a message interface, but the implementation of the business object is located on the server. The methods that implement the algorithm can be updated in one place, and new implementations can be made available to clients immediately (after adequate testing, of course). The server Smalltalk objects define security policy so that only certain clients, say, those who have paid to use this service, are allowed to execute the risk assessment methods. Finally, the server Smalltalk provides a central repository of objects that can be backed up to tape for archival purposes or recover from hardware failure.

Because of the different roles played by client applications and the application server in the 3-tier model, the requirements for server Smalltalk are quite different than those of client Smalltalk systems. Client Smalltalks operate in a single-user environment. They provide an extensive GUI and graphics class library, and are integrated with the windowing environment of the client workstation. The virtual machines of client Smalltalks are tuned for virtual memory access. A Smalltalk on the server, on the other hand, operates in a multi-user environment. It must pro-

vide a model of transactions and concurrency control, and provide a class library designed with multi-user access in mind. The virtual machine of server Smalltalk is tuned for disk access, and must be able to handle very large objects and a very large number of objects. This Smalltalk is tailored to operation on server-class machines to take advantage of shared memory, asynchronous IO, and raw partitions on disk. Server Smalltalk is built with transaction throughput and client communication as chief considerations. Trying to build a server using a client Smalltalk system will not provide the performance or functionality required for large-scale applications.

By using multi-user Smalltalk as the application server in a 3-tier architecture, developers can implement shared

Trying to build a server using a client Smalltalk system will not provide the performance or functionality required for large-scale applications.

business objects with the same language used to build client applications. This enables developers to move behavior easily from the client to the server, an activity called *application partitioning*. See Wadhwa³ for a description of application partitioning issues. With a common object model on both the client and the application server, objects do not need to be transformed from one form to another. Because the same

code can execute in either the client or the application server, the developer can initially build the application entirely on the client, then move portions of it to the server as needed. This might be done to share objects, optimize performance, enforce security, or backup important data. In addition, these partitioning decisions are easily changed as new hardware or software is added to the system. Having a common object model and language between the client and the server makes the repartitioning of an application much simpler, since there is little if any code to rewrite.

When partitioning an application, how does a developer determine where certain objects should reside, i.e., in the client or in the server? Here are some general rules-of-thumb to help in this process. The following kinds of objects belong on the server: business objects, sensitive objects requiring security, large collections of objects requiring optimized query capability, objects requiring shared access, objects requiring fault tolerance, and "gateway" objects (i.e., objects that provide a view of raw data on the data server). The following kinds of objects belong on the client: window or GUI objects, application-specific objects, and "view" objects (i.e., objects that provide a view of a server object). My next column will discuss the implementation techniques for application partitioning. ■

References

1. Rymer, J.R. Business objects, DISTRIBUTED COMPUTING MONITOR, 10(1), 1995.
2. Lozinski, C. THREE-TIER CLIENT-SERVER ARCHITECTURES, Berkeley Productivity Group, 1995.
3. Wadhwa, V. Partitioning apps: What are the issues? UNIX REVIEW, May 1995.



Jan Steinman



Barbara Yates

A case for open development environments

WARNING: *This column contains inflammatory material. It is likely to raise the blood pressure of Smalltalk vendors. If you are an easily inflamed person, and you don't care much about what your customers want or need, press "n" now (and start reading Chapter 11!)*

AS SMALLTALK FINDS wider use and new vendors appear, the trend is clear: Smalltalk developers need to be protected from themselves. Those pesky developers are so demanding—they want to change everything, and then have their changes supported, too! (Exit cynic mode—we realize open environment technical support is no easy task.)

One way out of this mess is well-specified interfaces. If Smalltalk vendors had the technical facilities to draw a line in the sand and say “pass this, and you're on your own,” users and vendors could work together to determine the appropriate cost for different levels of support. But the easy way out is to simply remove access to the source code, without even specifying a system programming interface (SPI).

Everybody is talking about “application programmer interfaces” (APIs) these days, but they are forgetting that not everybody is an “application programmer.” The beauty of Smalltalk is that it works both as a systems language and an applications language. With persistent rumors that the new regime at ParcPlace-Digital is considering “protecting developers from themselves,” we've decided to publish some of our favorite Smalltalk system programming examples. Most of these examples will only work with VisualWorks today. Tomorrow, they may not work with VisualWorks either.

COMPILER MACROS

New users of Smalltalk often react with a sense of wonder when they discover that all control constructs are actually implemented in the language. One of those new users was a project leader at one of our clients.

Jan Steinman and Barbara Yates are cofounders of Bytesmiths, a technical services company that has been helping companies adopt Smalltalk since 1987. Between them, they have over 20 years Smalltalk experience. They can be reached at Barbara.Bytesmiths@acm.org or Jan.Bytesmiths@acm.org.

“You know what I really miss about C,” he said, “is the ‘question-colon’ operator.” Upon closer questioning, we discovered that what he *really* wanted was a quick and simple way of dealing with uninitialized variables.

“Okay, let's implement it!” we replied. “Hmmm...it needs to be simple...don't want a bunch of parentheses everywhere...sounds like a binary message to me.”

Object

? block

“If I am not nil, answer myself, otherwise answer the value of <block>.”

UndefinedObject

? block

“If I am not nil, answer myself, otherwise answer the value of <block>.”

^block value

This allows you to easily protect against unwanted nils, therefore making your system more robust. For example, you might have a method that prompts the user for a String, but you want a reasonable default:

```
Dialog request: message ? [Type something, will ya?  
We're paying for this stuff!]
```

This avoids having to use a conditional assignment to a temporary variable, and is also quite easy to read. Also, it doesn't involve any “systems” programming, yet. On the other hand, a message send is involved, which costs a bit more time than `ifTrue:iffalse:` does.

“But why should it cost more?” our client asked. It didn't take more than a few minutes of rummaging around the compiler to come up with the answer: it needn't cost more. Add the method in Listing 1 to *MessageNode*, keeping in mind the warnings we gave about base modifications in the July issue.¹ (We put this and similar extensions in a separate ENVY application called *CompilationBytesmiths*). Evaluate

```
(MessageNode classPool at: #MacroSelectors)  
at: #? put: #transformIfNil
```

(Using ENVY, we put this expression, and other similar ones below, in *CompilationBytesmiths* class>>loaded,

Listing 1.

```
MessageNode
transformIfNil
    "If the receiver is nil, evaluate the argument. MacroSelectors associate this action with the selector #?."

    ^((arguments first isBlockWithNumArgs: 0) and: [receiver hasEffect not]) ifTrue:
        [receiver := self class new
         receiver: receiver
         selector: #==
         arguments: (Array with: (LiteralNode new value: nil)).
         self makeIfTrue: arguments first ifFalse: (BlockNode new body: receiver receiver)]
```

Listing 2.

```
MessageNode
transformIfInDevelopment
    "If the system is not in development, remove this message. If the system is in development, insert the argument block's statements. MacroSelectors associate this action with the selector #ifInDevelopment:."
    "self halt. self ifInDevelopment: [Transcript cr; show: 'Yup, I'm in development.']. 27 = 27"

    ^((self respondsTo: #isInDevelopment) and: [self isInDevelopment])
        ifTrue: [receiver := arguments first body]
        ifFalse: [receiver := SequenceNode new statements: #()]
```

Listing 3.

```
MessageNode
transformRuntimeNoOp
    "If the system is not in development remove this message. If the system is in development, generate the message. MacroSelectors associate this action with the selectors #debug, #debug:, and #halt."

    ^((self respondsTo: #isInDevelopment) and: [self isInDevelopment])
        ifFalse: [receiver := SequenceNode new statements: #()]
```

and also added a **removing** method to get rid of these new "macro selectors" when *CompilationBytesmiths* is removed).

Now when **? appears** in your code with a simple receiver, the compiler "in-lines" it into a test for nil and a conditional branch, the same way it deals with **ifTrue:** and other "fake" messages—no message sends involved. Some simple timings show it to be about 30% faster without the message sends.

This is kind of cute, but the few microseconds it saves is hardly going to make or break a project. However, this basic mechanism can be exploited to strip your code of debug statements and assertions.

Assertions are like bran cereal—everybody agrees it's good for you, but nobody really likes the taste. Smalltalk assertions typically steal cycles from you even beyond the development phase where they're needed. What assertion writers *really* want is the C preprocessor, so that when you hit "**#ifdef DEBUG**" on your final compile, the assertion code simply goes away. Well, we've got access to the VisualWorks compiler, so let's do it!

First, establish a predicate for all manner of development-only code. We define an ENVY application called *TestingBytesmiths* that olds our test management framework; its presence is an ideal development predicate:

Object

isInDevelopment

"Is this a development image? Since this consumes runtime, avoid using this in performance critical code."

^Smalltalk includesKey: #TestingBytesmiths

As the comment indicates, a dictionary look-up is a rather heavy price to pay to find out you don't want to print a Transcript message in a production environment! To further encapsulate this, we also have a conditional action:

Object

ifInDevelopment: block

"If this image is in a 'development' state (whatever that means), do <block>, otherwise do nothing. In either case, answer self. Since this consumes runtime in either case, avoid using this in performance critical code."

self isInDevelopment ifTrue: [block value]

In fact, the greater encapsulation of the conditional action is much preferred, as you'll see shortly—the predicate method **isInDevelopment** should be considered private.

Listing 4.

```

ifFalse: "No temp declarations yet; have to insert whole line"
["Added/modified by Bytesmiths, on 7 October 1995: figure out how many tabs to insert."
 tabs := 1.
 [(editor text at: endTemps - tabs) == Character tab] whileTrue: [tabs := tabs + 1].
 replacement := ' | ', name, ' | ',
 (tabs = 1
  ifTrue: [' ' ]
  ifFalse: [(String new: tabs withAll: Character tab) at: 1 put: Character cr; yourself])
  "*****end addition/modification*****".
editor selectAt: endTemps.

```

Listing 5.

CompiledMethod**setSpecificationFromSource: source**

"Set my comment user field to the comment contained in <source>, my source code."

```

| comments comment args charSet |
comments := Compiler preferredParserClass new
  parseMethodComment: source setPattern: [:x | ].
comments size > 0 ifFalse: [^self].

comment := TextStream on: (String new: 100).
args := (Compiler preferredParserClass new
  parseArgsAndTemps: source
  notifying: nil) readStream.
self selector numArgs = 0
ifTrue: [comment emphasis: #bold; nextPutAll: selector]
ifFalse:
  [self selector keywords do: [:kw |
    comment
      emphasis: #bold; nextPutAll: kw; space;
      emphasis: #italic; nextPutAll: args next; space]].
comment emphasis: nil.
args := args contents copyFrom: 1 to: self selector numArgs.
charSet :=
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ' asSet.
comments do: [:cmt | | cmtStream |
  cmtStream := cmt readStream.
  comment cr; tab.
  [cmtStream atEnd] whileFalse: [ | pair |
    pair := cmtStream nextWordAndNonWordDefinedBy: charSet.
    (args includes: pair first)

```

od, and step through the *halt*, you will see the following decompiled code in a "development" image:

```

self halt.
Transcript cr; show: 'Yup, I'm in
  development.'.
27 = 27

```

and if you temporarily redefine *Object>>isInDevelopment* to answer false, the decompiled code will look like

```

self halt.
27 = 27

```

all without a single "#ifdef"! We also in-line other conditional development-time messages using the code in Listing 3. This works because there is no "ifTrue:" part, so it returns nil when not in development, which tells the sender to generate the original message send, instead of generating an in-line bytecode sequence. We can hear some of the Smalltalk vendors who hide their compiler mumbling something in the background like "we can provide 'hooks' to do things like that." Great—we're happy they can perfectly anticipate all the potential uses one might make of the compiler! But what about their compiler bugs? (No, they don't have compiler bugs!)

COMPILER BUG FIXES

Remember the ? example? It pays back the effort it took to understand it when used as a pattern for stripping out development-specific code (see Listing 2).

Let the compiler know you've defined a new macro. If you're using ENVY, remember to put this in your **loaded** method, and to remove your macro selector in your **removing** method.

```

(MessageNode classPool at: #MacroSelectors)
at: #ifInDevelopment put:
  #transformIfInDevelopment

```

Now if you do the comment expression in the above meth-

Some of these are controversial, and some may well be a minority opinion, but our point is that without compiler source code, we would not even have a choice about dealing with, shall we say, "undesired compiler behavior."

Most Smalltalk dialects have block temporary variables, and using block temporary variables whenever possible can have an important performance benefit. For simple cases, we've measured a 100% speed penalty when using method temporaries instead of block temporaries.

Yet the "helpful" VisualWorks compiler always places undeclared temporaries in the method context. This often causes what could be a "clean" block to be a "copying"

INFO@SIGS

SIGS Publications, Inc., 71 West 23rd Street, 3rd Floor,
New York, NY 10010; 212.242.7447; Fax: 212.242.7574

ARTICLE SUBMISSION

To submit articles for publication, please contact:
John Pugh & Paul White, Editors, 885 Meadowlands Dr.
#509, Ottawa, Ontario, K2C 3N2 Canada; email:
streport@objectpeople.on.ca

PRODUCT REVIEWS AND ANNOUNCEMENTS

To submit product reviews or product announcements,
please contact the Editors at the address above.

CUSTOMER SERVICE

For customer service in the US, please contact PO Box
5050, Brentwood, TN 37024-5050; 800.361.1279; Fax:
615.370.4845; in the UK, please contact Subscriptions
Department, Tower Publishing Services, Tower House,
Sovereign Park, Market Harborough, Leicestershire, LE16
9EF, UK; +44.(0)1858.435302; Fax: +44.(0)1858.434958

SIGS BOOKS

For information on any SIGS book, contact: Don Jackson,
Director of Books, SIGS Books, Inc., 71 West 23rd Street,
New York, NY 10010; 212.242.7447; Fax: 212.242.7574;
email: donald_jackson@sigs.com

SIGS CONFERENCES

For information on all SIGS Conferences, please contact:
SIGS Conferences, 71 West 23rd Street, 3rd Floor, New
York, NY 10010; 212.242.7515; Fax: 212.242.7578; email:
info@sigs.com

BACK ISSUES

To order back issues, please contact: Back Issue Order
Department, SIGS Publications, 71 West 23rd Street, 3rd
Floor, New York, NY 10010; 212.242.7447; Fax:
212.242.7574

REPRINTS

For information on ordering reprints, please contact:
Reprint Management Services, 505 East Airport Road,
Box 5363, Lancaster, PA 17601; 717.560.2001; Fax:
717.560.2063

ADVERTISING

For advertising information, please contact: Advertising
Department, SIGS Publications, 212.242.7447; Fax:
212.242.7574

SIGS HOME PAGE

To access the SIGS Home Page on the
World Wide Web: <http://www.sigs.com>.

block instead. But we have the source—let's fix it! In
InteractiveCompilerErrorHandler>>declareTemp:from: change

```
endTemps := codeStream homeStream topNode body
              sourcePosition first.

to
endTemps := codeStream topNode body sourcePosition
              first.
```

That's right, simply remove "homeStream."

Now if you let the compiler declare your temps for you, it will put them where they usually belong, in the innermost scope. (If the variable is used after the block, it will complain that you've redeclared it, thus clueing you that you really need it in the outer scope, and perhaps clueing you that you should consider redesigning the method so that it won't have a full block!)

The only problem we've discovered with this "bugfix" is that the formatting is a little weird—try further modifying the above method as shown in Listing 4 (first and last lines are from the original method), which makes things pretty again. We desperately needed this for a block editor we were building for a hypertext system—temps automatically declared outside the block were simply unacceptable in this case.

A less controversial change for anyone who has needed the compiler in a "headless" environment is its insistence on interacting with someone when syntax errors are detected. We mentioned in our September column² that we found it necessary to implement *silentEvaluate:*, which always raises an exception when evaluation fails for any reason, rather than bringing up a syntax error dialog. This "bugfix" would not have been possible without the compiler source code.

COMMENT PULLING

Access to the Smalltalk parser simplifies many tasks, particularly for tool builders. For example, our SmallDoc system (partially described in our June³ and September² 1995 columns) pulls method comments out of the source code and pastes them as styled *Text* into the little-used ENVY comment field (Listing 5), where they are easily accessed for a variety of documentation purposes. This method is conditionally sent from *ClassDescription>>insert:withSource:classified:ifNewAddTo:* so that every "accept" updates the ENVY comment field.

METRICS

Interest in measurement is rapidly increasing, yet no standard solution exists. The Smalltalk vendors will probably give developers some sort of metrics capability someday, but will it be right, and will those early adopters who have implemented their own metrics be willing to give them up? Will emerging third-party products be rewarded for their risk by being stranded without sufficient SPIs?

We've implemented some code quality metrics for our clients that rely on "deep" access to compiler and parser classes. At the OOPSLA '95 Smalltalk Testing Workshop

continued on page 32

MANAGING OBJECTS *continued from page 20*

we hosted, John Brant presented a code quality tool he is working on at University of Illinois, Urbana-Champaign that goes far beyond what we have done. He expects to make it widely available when completed, but if the system-level classes he exploits become “protected,” such a thing might not be possible. Emerging coverage tools that do bytecode manipulation might also be threatened.

CONCLUSION

As Smalltalk enters mainstream management information system shops, Smalltalk vendors claim there is a need to “protect developers from themselves.” They propose to do this by removing source code and by supplying “hooks” to anticipated “APIs.” This flies in the face of conventional wisdom about reuse. Reuse often involves modifications and extensions; it usually is discovered, harvested, reengineered, and sought out, but it is rarely anticipated.

We feel this argument is a thinly disguised way of reducing support costs, which is better addressed by clearly

defining SPIs in such a way that both system and application programmers know when they’ve overstepped their limits (and their support contracts!)

The power of Smalltalk comes from many different aspects and trying to be more like Visual Basic, PowerBuilder, or Hot Java by reducing access to “dangerous” (i.e., “hard to support”) system features in the name of making things “safe” for application programmers is likely to have a detrimental effect.

If you want to do some of the things in this column, but your Smalltalk dialect doesn’t have the proper hooks, let your vendor know you need the full source to discover what you need to reuse. If your Smalltalk vendor currently gives you all the source you need, call them and thank them, then tell them you expect the situation will not change if they don’t want you running off to Hot Java! ☞

References

1. Steinman, J and B. Yates. A case for open development environments, *THE SMALLTALK REPORT* 4(9):26–27, 31, 1995.
2. Steinman, J. and B. Yates. Managing project documents, *THE SMALLTALK REPORT* 5(1):23–30, 1995.
3. Steinman, J. and B. Yates. Managing project documents, *THE SMALLTALK REPORT* 4(8):25–28, 1995.



Kent Beck

Farewell and a wood pile

IT'S THE OBJECTS, STUPID

SOMETIMES IT TAKES me awhile to see the obvious. Sometimes even longer than that. Three or four times in the last month I've been confronted by problems I had a hard time solving. In each case, the answer became clear when I asked myself the simple question, "How can I make an object to solve this problem for me?" You think I'd have figured it out by now: got a problem? make an object for it.

Here's an example: I had to write an editor for a tree structure. There were several ways of viewing and editing the tree. On the left was a hierarchical list. On the top right was a text editor on the currently selected node of the tree. On the bottom right was a list of text editors on the sub-nodes of the currently selected node (see Fig. 1).

Figure 2 shows the domain objects that live behind this view. How is the editor going to work? Let's say I have an editor on the value 5 (Fig. 3). How are we going to write the code to parse and install a new function? The first part is simple enough:

```
FunctionEditor>>parse: aString
| new |
new := FunctionParse parse: aString.
```

But now we're stuck. If we just say:

```
function := new
```

then the "right" instance variable of the *BinaryFunction* (which the editor knows nothing about) won't be updated.

"Make an object for it," that's the ticket. The object is an *EditingWrapper*. When you go to edit a function, you first wrap every node in the function tree, as shown in Figure 4. Now the editor looks like Figure 5. And we can write the parsing method like this:

```
FunctionEditor>>parse: aString
| new |
new := FunctionParse parse: aString.
function function: new
```

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

If we parsed the string "*@years*", the resulting picture would look like Figure 6. When the *BinaryFunction* unwraps its children, the right function will be in place.

As I said, several times in the last month I've faced baffling problems that became easy when I asked myself the question, "How could I make an object to solve this problem for me?" Sometimes it was a method that just didn't want to be simplified, so I created an object just for that method. Sometimes it was a question of adding features to an object for a particular purpose without cluttering the object (as in the editing example). I recommend that the next time you run into a problem that just doesn't seem like it has a simple solution, try making an object for it. It won't always work, but when it does it's sweet.

THE PARABLE OF THE WOOD PILE

The following is really about software. Really.

I live in the redwood forest. Fall in the forest has its own set of smells, distinct and different from the smells of every other season. Crushed dry ferns have a sharp, dusty

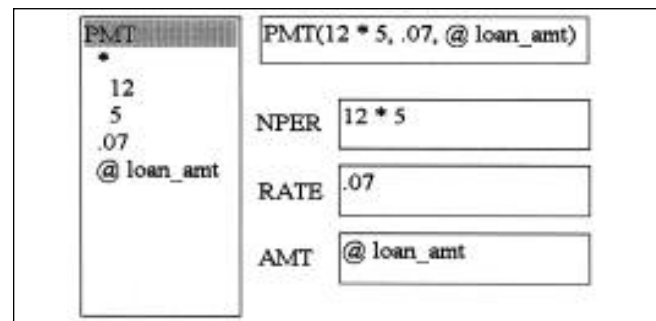


Figure 1.

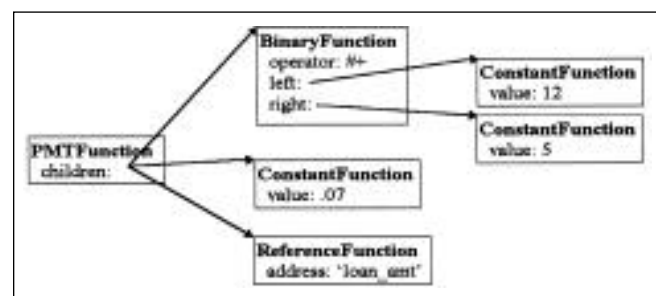


Figure 2.

smell. Rotting bay nuts are like psychedelic bay leaves. When we get our wood delivered, the smells of freshly split oak and madrone add to the mix.

My house is down by the creek, maybe 25 feet below the level of the driveway. There is a sheer cliff off to one side and stone steps directly in front of the house. When we get our customary two cords of wood delivered (for you city folk, that's a pretty damn big pile of wood, takes most of a 2-ton truck to carry it), the easiest way to get it down near the house is to throw it over the cliff, one stick at a time, then go down later and stack it.

Wood chucking time has become something of a ritual for me. The smells of the fall forest, the filtered fall light through the surrounding redwoods, the ache of my generally-desk-bound body, the knowledge that I'm keeping my family warm for the rest of the winter, all combine for a satisfying couple of days.

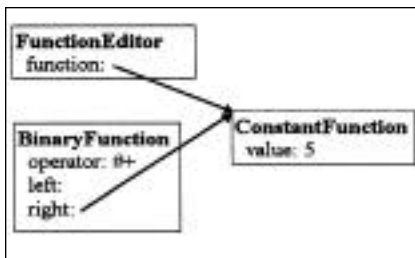


Figure 3.

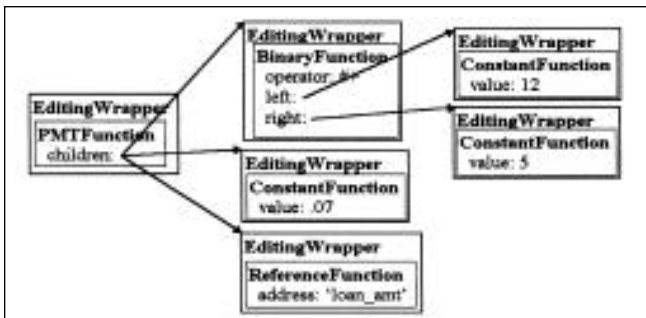


Figure 4.

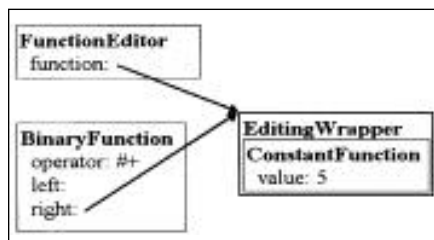


Figure 5.

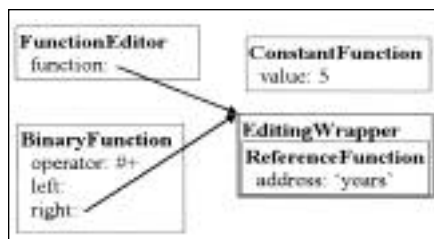


Figure 6.

My driveway is long and narrow, so when the truck delivers the wood it makes a long pile, maybe 25 feet long and eight or nine feet wide. The end of the pile is right at the top of the cliff, so the first hour or so is easy—turn, pick up a stick, turn, throw. Once I get settled into a rhythm, I probably throw a stick every five seconds.

This year we had a dinner party to attend, and I didn't want to have to walk over pile of firewood all dressed up, so I wanted to at least get a path cleared quickly. Once I got the sticks close to the top of the cliff thrown, I noticed that my progress slowed down. Instead of "turn, grab, turn, throw" I was doing "walk, grab, walk, throw," where I was having to walk a few steps to get to the front of the pile. It may not seem like much, but it slowed down my throwing rate by half. The more progress I made, the further I had to walk, the slower I went, the further my goal of walking to the car without scuffing my shiny shoes receded.

I'm an engineer at heart, and repetitive manual labor leaves me plenty of time to think, so I wasn't about to let this state of affairs continue without at least trying to bring my productivity back up. I discovered I could throw light sticks down with one hand. On every trip to the front of the wood pile I began picking up two sticks, a heavy one in my left hand and a light one in my right. I'd throw the light one one handed first, then heave the heavy one with both. This let me amortize my walking over two sticks. The pace picked up.

Pretty soon, though, I noticed I was still going slow. The front of the pile kept receding as I worked, so my time spent walking kept increasing. What I really needed was a way to get back to working like I had worked at first, just turning and throwing with no walking at all.

You've probably guessed the solution. I went to the pile and tossed sticks the 10 or 15 feet to the top of the cliff. I tossed 30–40 sticks, walked over, threw them down the hill, then walked back. This way my walking was amortized over so many sticks it didn't even count. I had to handle each stick twice, so my productivity was half of what it was at the beginning, but I could sustain the pace through the rest of the pile. No matter how far back the front of the pile got, it was always easy to quickly toss a little stack to the top of the cliff.

My wife and I made it to our party—shoes, suit, and dress unscathed.

When an experienced team starts a project in Smalltalk, the first few months go smoothly. The first cut at the domain model slides right in and away you go. Pretty soon, though, the team starts to bog down. Decisions they made without complete information begin to take their toll. The easy progress of the early days is soon but a fond memory. New functionality, rather than sliding in, has to be shoved in with a pile driver. Quality and predictability go out the window, because the team doesn't know if the next feature will fit with what's there, in which case all will be well, or it won't fit, in which case who knows how long it will take to shoehorn it in.

I have seen two unproductive reactions to this situa-

tion and one reasonable one. The first are the teams that keep walking back and forth to the wood pile, no matter how far it recedes. I call this “Smalltalk is more rope.” These teams ignore the increasing risk and decreasing productivity, but Smalltalk is forgiving enough that they can keep their application sort of running while they add new features. Throw in enough *nil* checks and *isKindOfs* and you can make almost anything work. The result is disaster deferred. Eventually the team is asked to do something that just can’t be shoved into the system.

The shell-shocked veterans of “more rope” failures often turn the other way. Ignoring the sticks right there in front of them, they try to toss the whole pile close before they start throwing down the hill. They insist on creating the frameworks first. The application is divided into strict layers and developers are only allowed to work on their own layer. The layers don’t precisely fit, because they are developed in isolation, but developers have no choice but to carry on as best they can. The result is again disaster deferred. The system gets big, because layers provide services no one needs and because there is no view of the whole system that would allow large-scale simplifications.

The sustainable solution is to find a balance between moving the pile and tossing the logs. Toss some, move some, toss some, move some, starting with tossing. (Jeff McKenna had a great article about this years ago, and Ward Cunningham has a pattern language called Checks about the same idea: <http://c2.com/ppr>). Take advantage of the quick wins to give you concrete information from which to generalize. Make it run, make it right, make it run, make it right.

FAREWELL

This is my last column, at least this go around, for THE SMALLTALK REPORT. It’s been quite a ride, programming in Smalltalk and trying to write about it. When I started, when THE SMALLTALK REPORT started, we were the wild-eyed purveyors of what many people saw as a crazy language. Since then, Smalltalk has become the language of

choice for many kinds of applications. Recently the Smalltalk market has been thrown into turmoil by the merger of ParcPlace and Digitalk and their subsequent disappointing financial performance.

From that standpoint, it seems like a strange time to quit. I’d like to go out on a high note, with noble Smalltalk standing proudly head and shoulders above the crowd. However, when I saw that I wasn’t putting the thought or care into these columns that they, that you, deserve, I knew the time had come.

I’ll still be involved in the Smalltalk world, in fact, more than ever. You won’t get rid of me that easily! I’ll be unveiling a one-day Smalltalk patterns course at Smalltalk Solutions in March. I’m working on a book, THE SMALLTALK BEST PRACTICE PATTERNS: VOLUME 1, CODING, due out in the first quarter next year. I’m scrambling to keep up with my products. I’m working on some fascinating contract programs. To top it off, consulting has picked up since OOPSLA. The only way you’ll be rid of me is if I drop dead of exhaustion.

Thanks

I’d like to thank all the people who helped me during the last few years. In particular:

- Rick Friedman, for giving Smalltalkers a forum for our voices when we were far out in the wilderness.
- John Pugh and Paul White, for all their work making the Smalltalk Report work well.
- Elizabeth Upp and the production team at SIGS, for dealing with late submissions, raw ASCII, and requests for odd graphics.
- Liz St. Pierre, for hassling me in the gentlest possible way consistent with results.
- Ward Cunningham, for help refining many of my best column ideas.
- You, THE SMALLTALK REPORT readers, for support, encouragement, email, and ideas. Without you I could have written all the columns I liked, but no one would have read them.

So long. I hope I see you Smalltalkin’ down the road. ☞

Editors' Corner



John Pugh



Paul White

WE'VE DONE A fair bit of "globetrotting" over the past month or so, and two observations we've been claiming (in chorus with others) for a long time have been very visible. One is the claim that the adoption of objects is further ahead in North America than in the rest of the world. This is not to say that object technology is not in use in quite a number of places. We know of a number of organizations "abroad" that have been building sophisticated, mission-critical systems using Smalltalk for a long time. And we know that our travels as a training organization have brought us to roughly 20 countries in just the last year alone. But individual examples aside, there just doesn't appear to be the widespread use of objects, and, in particular, Smalltalk, outside of the US, and to a lesser extent Canada.

This begs the obvious question of why not, for which there is probably no right answer and certainly not a simple one. Some comments we've heard lately is that the sales force for the Smalltalk vendors has been relatively small and has not had the reach to penetrate large organizations. This is certainly changing with IBM now utilizing its full worldwide sales and marketing force. But also, ParcPlace-Digitalk has been partnering with a variety of distributors throughout the world over the past few years, which has allowed them to make local connections with high-profile companies. Perhaps a second reason for the relatively smaller market size outside North America is the very nature of Smalltalk's class library. As it is written in a style that is very "English-like," and to a great extent American-English, perhaps learning the language is that much more difficult in countries where this style of language is not used. Certainly the grammatical style used with the language and library could be described as colloquial and would be foreign to most developers in, say, Asia or Eastern Europe. I doubt there is an easy way for this issue to be addressed. A third comment, made to us by a few of our clients, is that the style of development used in North America differs from that used elsewhere. Without generalizing too much, we in North America seem more eager to adopt this new iterative, incremental lifecycle than our counterparts in other parts of the world, where a more formal lifecycle, which is documented religiously, is much more the

We in North America seem more eager to adopt this new iterative, incremental lifecycle than our counterparts in other parts of the world.

norm. We know that changing to this more rapid and free-wheeling development approach has often been very difficult to accept in US corporations—it must be even more so in many places elsewhere.

The second observation that has been confirmed is simply that we have a long, long way to go before those of us responsible for delivering systems rapidly and effectively will ever catch up to the demands of business. For example, the improvements made by many of the airlines with respect to their reservation systems is enormous, but they still don't meet the demands of the current airline business. With rapidly changing schedules and new partnerships and alignments between carriers appearing each month, it is difficult to imagine how systems will ever keep up. An even better example

was seen on a recent trip to a major theme park in the US. Just a quick glance at the variety in their holiday packages would bring a tear to the eye of someone imagining what their CIO must go through. They listed roughly a dozen "new" packages, each with different pricing, scheduling, and restrictions listed, some of which were

intertwined as "packages of packages." From a consumer point of view it was wonderful, but watching the poor lady behind the counter spend 90 minutes trying to figure out how to register us for two of these packages was heartbreaking. She had the two-inch manual, which contained the step-by-step instructions, which had ink written all over it with recent amendments and corrections to it. The saddest part of course was the 25 minutes it took her to recover from entering an invalid date in just one field! An "undo" button would have been very useful for her.

As a final note, we wish to formally offer Kent Beck our most sincere thanks for the contributions he has made within these pages over the past five years. As one of the very first people we contacted after being asked to act as editors of this publication, it is sad that we must finally bid him adieu. He has been our most prolific contributor, and I know many of you have benefited from the ideas and musings in his columns. We know he's not disappearing, just taking an extended sabbatical from writing here, and it is with our deepest gratitude we say thanks.

Happy New Year everyone!

Editors

John Pugh and Paul White
Carleton University & The Object People

SIGS Publications Advisory Board

Tom Atwood, *Object Design*
François Bancelhon, *O₂ Technology*
Grady Booch, *Rational*
George Bosworth, *ParcPlace-Digitalk*
Jesse Michael Chonoles, *Lockheed Martin ACC*
Stuart Frost, *SELECT Software*
Adele Goldberg, *ParcPlace-Digitalk*
Thomas Keffer, *Rogue Wave Software*
R. Jordan Kriendler, *IBM Consulting Group*
Thomas Love, *Consultant*
Bertrand Meyer, *ISE*
Meilir Page-Jones, *Wayland Systems*
Cliff Reeves, *IBM*
Bjarne Stroustrup, *AT&T Bell Labs*
Dave Thomas, *Object Technology International*

The Smalltalk Report

Editorial Board

Jim Anderson, *ParcPlace-Digitalk*
Adele Goldberg, *ParcPlace-Digitalk*
Reed Phillips
Mike Taylor, *ParcPlace-Digitalk*
Dave Thomas, *Object Technology International*

Columnists

Jay Almarode, *GemStone Systems Inc.*
Kent Beck, *First Class Software*
Juanita Ewing, *ParcPlace-Digitalk*
Bob Hinkle, *Consultant*
Tim Howard, *FH Protocol, Inc.*
Ralph E. Johnson, *University of Illinois*
Alan Knight, *The Object People*
Mark Lorenz, *Hatteras Software, Inc.*
Jan Steinman, *Bytesmiths*
Rebecca Wirfs-Brock, *ParcPlace-Digitalk*
Barbara Yates, *Bytesmiths*

SIGS Publications Group, Inc.

Richard P. Friedman, Founder, President, and CEO
Hal Avery, Group Publisher
John McCormick, Editorial Director

Editorial/Production

Elizabeth A. Upp, Managing Editor
Elisa Varian, Production Manager
Andrea Cammarata, Art Director
Kathleen M. Major, Sr. Production Editor
Sue Mycka, Desktop Designer
Margaret Conti, Advertising Production Coordinator
Shannon Smith, Editorial Production Assistant

Circulation

Elayne Glick, Circulation Director
Lawrence E. Hoffer, Marketing Manager

Advertising/Marketing

Gary Portie, Advertising Manager, East Coast/Canada/Europe
Elisa Marcus, Advertising Manager, Central US
Michael W. Peck, Advertising Representative
Kristine Viksnins, West Coast Exhibit Sales
Sarah Olszewski, East Coast Exhibit Sales
212.242.7447 (v), 212.242.7574 (f)
Diane Fuller & Associates, Sales Representative, West Coast
408.255.2991 (v), 408.255.2992 (f)
Wendy Dinbokowitz, Promotions Manager for Magazines

Administration

Margherita R. Monck, General Manager
David Chatterpaul, Senior Accounting Manager
Bibi Budhrum, Accounts Payable



Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, THE SMALLTALK REPORT, THE X JOURNAL, REPORT ON OBJECT ANALYSIS & DESIGN, OBJECT CURRENTS, OBJECT EXPERT (UK), and OBJECTSPEKTRUM (GERMANY)

Features

The selection channel technique

4

Bobby Woolf

The author documents a technique often used in VisualWorks that simplifies adapting several value models to a single domain model. The technique itself demonstrates the ease and flexibility value models can provide.

Delivering and sharing components using Smalltalk link libraries

9

Makrand Utpat

Developing a system architecture consisting of components and sub-systems offers a number of advantages. The construction of Smalltalk link libraries (SLLs) both facilitates component-oriented software development and reduces the need to trim "fat" executables.

Columns



The Best of comp.lang.smalltalk

13

Principles of OO design: or, everything I needed to know in life, I learned from Dilbert
Alan Knight

There have been many attempts to define principles of OO design or coding, but these often lack concrete information or are just plain boring! Try these interesting (but completely serious) principles of OO-ness instead.



Managing Objects

15

"Special" team members

Jan Steinman and Barbara Yates

Creative people can be a mixed blessing; their insight often comes wrapped in a demanding personality. Through its enhanced productivity, Smalltalk

amplifies both the advantages and disadvantages of team members. Here are some coping strategies for some of these "special" people.



Getting Real

18

Mechanisms for application partitioning

Jay Almarode

In the three-tier architecture, you need to balance the processing load between clients and server, and find a way to share business objects. Application partitioning is the key to handling both these issues.

Departments

Editors' Corner

2

Recruitment

20

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar-Apr, July-Aug, and Nov-Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1996 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Bulk rate U.S. postage paid Lancaster, PA, permit 161. Canada Post International Publications Mail Product Sales Agreement No. 290386.

Individual Subscription rates 1 year (9 issues): domestic \$89; Mexico and Canada \$114, Foreign \$129; Institutional/Library rates: domestic \$199, Canada & Mexico \$224, Foreign \$239. To submit articles, please send electronic files on disk to the Editors at 885 Meadowlands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

POSTMASTER: Send domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sigs.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

The selection channel technique

Bobby Woolf

In “Making MVC more reusable,”¹ I talked about how VisualWorks has improved upon classic model-view-controller (MVC). That article discussed new objects like value subviews and value models as well as the way “model” has been split into application model and domain model. Its third figure shows the objects involved when the model is split into application and domain models. When I drew it, I threw in another object that is often involved, the one that selects which domain model the application model will use. I labeled this object “SelectionChannel.” In this article, I’ll explain what a selection channel is.

Selection channel is an extension of what ParcPlace is calling the “Slam Dunk” architecture.² Slam Dunk uses subject channels to greatly simplify the code needed to connect an application model to its domain model. The idea is that once you set up an application model this way, you can insert (i.e., slam dunk) a domain model in there and everything in the application model and its window will update automatically. Selection channel takes this idea one step further to specify where the various domain models are coming from and who is inserting them into the application model.

To understand selection channel, let’s first take a look at where graphical user interfaces (GUIs) come from. The easiest way to produce a GUI window in VisualWorks is to use the Painter to produce a view with a bunch of widgets on it. Most of the widgets are value subviews, each displaying some value it contains: an Input Field shows some text, a Check Box shows whether a setting is on or off, etc. Where do these values come from?

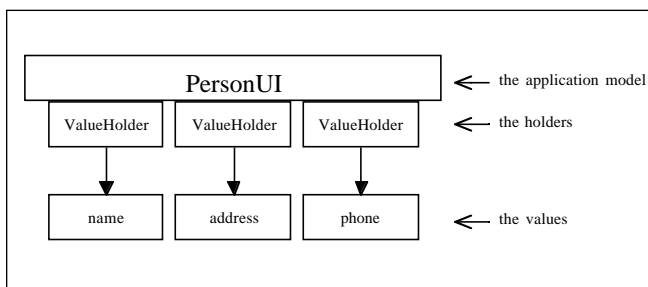


Figure 1. An application model with no domain model.

ADAPTING A DOMAIN MODEL

VisualWorks uses a revision of the MVC paradigm that breaks the model into two parts: the domain model and the application model. The *domain model* contains the information to be displayed, whereas the *application model* organizes the information so that it can be displayed. In the MVC, the window is the view and the application model is its model. This macro MVC system contains a number of MVC micro systems, one for each value subview, where the subview is the view and its model is a value model. A *value model* (an instance of ValueModel) is a simplified model that has exactly one aspect. Thus the value model contains a single value, which in this context is the information a single widget displays.

A window generated by the Painter uses an application model but, by default, does not use a domain model. The Painter generates an application model (a subclass of ApplicationModel) that is able to open the window when told to do so. Since the window contains widgets that require value models, the application model contains those

value models. By default, the application model initializes those value models to be holders (instances of ValueHolder) on default values like empty string, false, zero, and nil (Fig. 1). All this default initialization is enough to make the view work: the window will open, the widgets display their values, and the widgets can be used to change their values. This, however, is not very useful functionality.

For an application model to really be useful, it should organize information that is stored in a separate object—a domain model. The question then is how the application model should get the information from the domain model. Often the first solution people discover is for the application model to copy the values out of the domain model and into the application model’s value holders. This works for displaying the domain model, but if the user edits the values in the window, the process of putting the values back into the domain model is difficult and redundant. It would be easier to associate each value model with its corresponding value in the domain model.

The type of value model that knows what domain value

SELECTION CHANNEL TECHNIQUE

it contains is an adaptor value model. An *adaptor value model* (an instance of `PluggableAdaptor`, `AspectAdaptor`, etc.) extracts its value out of an object that contains the value. The adaptor calls the value's container its *subject* (accessible via `ProtocolAdaptor>>subject`). Each adaptor knows how to pull its value out of its subject to show the value to the user. If the user edits the value, the adaptor knows how to put the new value back into its subject. An application model adapts its domain model by using adaptors. The application model initializes its value models as adaptors instead of holders, where each adaptor's subject is the application model's domain model. This way the application model maps each widget in the window to its corresponding value in the domain model (Fig. 2).

SUBJECT CHANNEL

When an application model is using several adaptors to adapt its domain model, all the adaptors must have the same subject. This way all the widgets in the window are showing different parts of the same domain model. This presents some difficulty when setting up the adaptors on a domain model: the code must ensure that all the adaptors are connected to the same domain model. Furthermore, to change to a different domain model, all the adaptors must be disconnected from the old model and connected to the new one. This leads to verbose, repetitive code that is prone to flaws.

The code for connecting and disconnecting a set of adaptors to a single subject can be virtually eliminated using a subject channel. A *subject channel* (accessible via `ProtocolAdaptor>>subjectChannel`) establishes a single point of access to a subject. The subject channel is itself a value model, typically a holder, whose value will be used as a subject. When multiple adaptors share the same subject channel, they are all guaranteed to share the subject. Even if the subject is replaced with a new one, the subject channel will automatically cause all the adaptors to disconnect from their old subject and connect to their new one. This way the application model does not need any code for connecting or disconnecting its adaptors to their subject; the subject channel feature does this automatically (Fig. 3).

Note that in VisualWorks 2.0, `PluggableAdaptor` does not support subject channel because it is not a subclass of `ProtocolAdaptor`.

The application model, instead of connecting its adaptors to their subject, has to connect them to their subject channel. First it must set up its domain model for use as a subject channel. Then it must connect the adaptors to their subject channel.

There are two ways an application model can set up the subject channel. One alternative is for the application model to obtain a value model that contains the domain model. The other alternative is for the application model to obtain the domain model and wrap a value model around it. Either way, once the application model has its domain model in a value model, it stores the value model. This value model becomes the application model's *domain model channel*, which means that the value

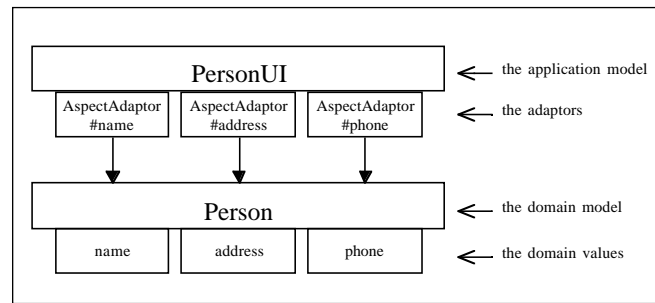


Figure 2. An application model adapting a domain model using adaptors.

model is the application model's single point of access to its domain model. The application model will go through its domain model channel to get its domain model. Once the domain model channel is established, the channel object (the value model) should never be replaced with another.

The application model uses its domain model channel to connect all its adaptors to the same domain model. As the code in the application model creates its adaptors, rather than setting each adaptor's subject to be the domain model, it sets each one's subject channel to be the domain model channel. This way the subject channels and the domain model channel are all the same object. Each adaptor sees this domain model channel as its subject channel and will connect itself to its subject via the subject channel. This way, if the domain model is replaced with a new one, all the adaptors will attach themselves to the new one.

SELECTION CHANNEL

Subject channel does not specify where the subject comes from, just that it will become available by being inserted into the subject channel. However, somebody must be setting the subject channel to contain a new subject. *Selection channel* considers not only that the subject is being replaced with a new one, but what object is making the replacement and where it's getting the new subject from. Whereas a subject channel just specifies how the subject can be accessed, selection channel also specifies where new subjects come from (Fig. 4).

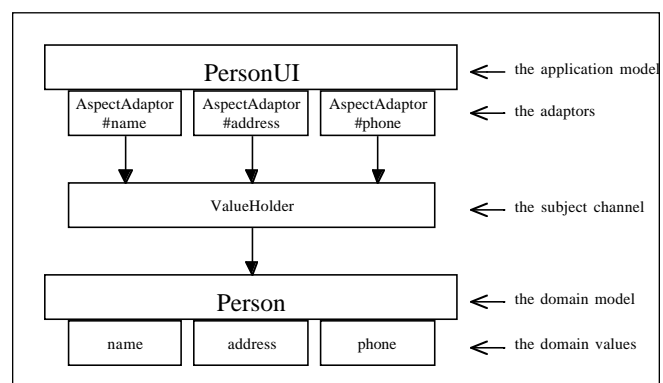


Figure 3. An application model adapting a domain model via a subject channel.

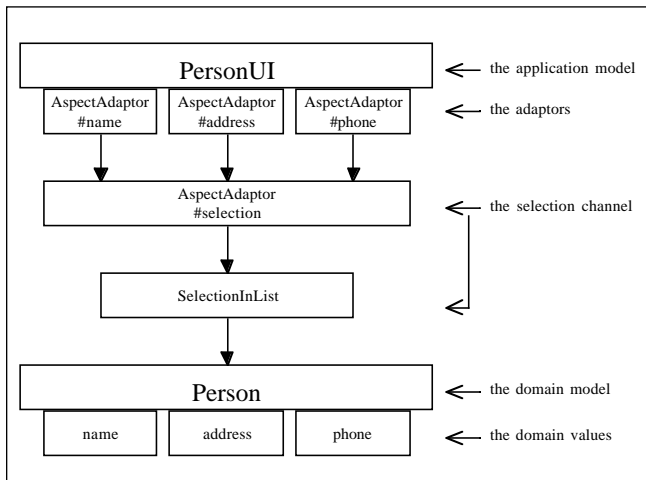


Figure 4. An application model adapting a domain model via a selection channel.

Considering how new subjects are obtained is important for setting up the subject channel correctly. For two objects to share the same value using a value model, they must share the same value model.³ Thus the object obtaining the new selection and the objects using the selection must share the same value model. The selecting object, the one choosing a new subject from a list of possibilities, must store its selection in a value model. This selection value model will be shared by the other objects using the selection.


VisualWorks already includes an object that is perfectly suited for use as a selection channel, the selection in list (an instance of `SelectionInList`). A selection in list itself is not a selection channel, but it contains one. In VisualWorks 1.0 you had to create your own selection channel on a selection in list using a pluggable adaptor. VisualWorks 2.0 has an enhanced aspect adaptor and a new method, `SelectionInList>>selectionHolder`, which returns an adaptor that is perfect for use as a selection channel. Selection channel does not require a selection in list, but that is the object most commonly used for this task.

An application model that will use a selection channel must obtain it from the selection object so that they will be sharing the same one. When an application model is establishing its domain model channel, it can either obtain that channel or obtain the domain model and create the channel itself. However, if the domain model channel is a selection channel, they must be the same object, so the application model must not create a new channel. If it does, the application model will not be notified when the selecting object changes the selection.

CONCLUSION

In a nutshell, here's what the article said:

- An application model organizes a group of values so that they can be displayed by the widgets in a window.
- The group of values should be stored in a domain model.
- An application model adapts its domain model using adaptor value models.
- An application model should use a domain model channel to set its adaptors' subject channels; this is a simple way to force them all to adapt the same subject/domain model.
- This domain model channel should be a selection channel from a selecting object such as a selection in list; this way the window will automatically display the item selected in the list.

Hopefully this explains what that object labeled `SubjectChannel` is all about. 

References

1. Woolf, B. Making MVC more reusable, *THE SMALLTALK REPORT* 4(4):15-18, 1995.
2. Robicheaux, M. Visual Slam Dunk tutorial, *THE PARCPLACE INTERNATIONAL USERS CONFERENCE*, 1994.
3. Woolf, B. Understanding and using the ValueModel framework in VisualWorks Smalltalk, *PATTERN LANGUAGES OF PROGRAM DESIGN*, Coplien, J.O. and D.C. Schmidt, Eds., Addison-Wesley, Reading, MA, 1995.

Bobby Woolf is a Member of Technical Staff at Knowledge Systems Corp. in Cary, North Carolina. Comments and questions are welcome at woolf@acm.org.

Delivering and sharing components using Smalltalk link libraries

Makarand Utpat

RECENT WORK ON frameworks¹⁻⁴ depicts the advantages of components and frameworks in general. Developing a system architecture consisting of components and subsystems is an excellent idea from the modular design and software maintenance points of view. In this article, I will show how construction of Smalltalk link libraries (SLLs) aids in maintaining a developed application and delivering component-oriented software. I will describe a way to ship a Smalltalk application by creating SLLs. Each SLL can represent a subsystem or a specific component in your application. The process of Smalltalk library construction makes a developer realize the framework benefits at the time of delivery.

The code examples described are based on Digitalk Smalltalk/V Version 3.0.1 for OS/2 PM (Presentation Manager), but the same concept can be tailored to the Smalltalk/V Windows environment. This example also demonstrates how Smalltalk takes advantage of host operating system features for the application delivery. Example code can be obtained on the World Wide Web at <http://www.objectpeople.on.ca/software>.

SMALLTALK APPLICATION PACKAGING AND MAINTENANCE

It is a well-known fact that once the learning curve barrier is overcome, a Smalltalk developer becomes proficient in writing Smalltalk programs and the development task becomes fairly easy. In large projects, a considerable amount of time may be spent producing and maintaining a good executable image. Full-time resources may need to be allocated to perform this task. As the executable image size begins growing, a Smalltalk developer may hear comments about having a "fat" executable. Everybody expects a nicely trimmed executable for application delivery. There is absolutely nothing wrong with this expectation; it is easier to manage a smaller executable during application shipment. However, some momentum gained in developing Smalltalk applications can be lost in the process of constructing and delivering a Smalltalk executable.

Once development is complete, the organization begins a testing phase. As a result of code enhancements,

code optimization, bug fixes, etc., code tuning begins as per feedback and suggestions of users. Often, these changes are specific to particular subsystem classes and/or methods, and classes and/or methods in other subsystems are not affected. Because the Smalltalk image (which uses `v.exe`, `change.log`, and `recover.log` files) is not partitioned, there is no way to focus the efforts on particular subsystem classes or methods to carry out code tuning. Here, SLLs come to the rescue. The organization can create a Smalltalk image that consists of SLLs for each of its subsystems and use them as needed, or distribute them appropriately to clients to satisfy client requirements. These SLLs are used either in a development or runtime environment as per one's needs. SLLs are very handy for accelerating the overall application packaging task and maintaining the Smalltalk application.

A REAL-LIFE EXAMPLE

Assume that an organization has developed a software framework to satisfy its business needs. Consider a simple situation where this framework uses a dependency mechanism (i.e., the user interface is dependent on a business model) and is comprised of two subsystems including a *business model subsystem* (to contain business state and business logic) and a *user interface subsystem* (to display business model data), etc. In reality, framework implementations may contain other subsystems such as a *rule model subsystem* (to handle business rules), an *application model subsystem* (to handle behavior related to the application model), a *communication model subsystem* (to handle message send/receive), a *database subsystem*, etc. Classes in these subsystems collaborate with each other appropriately per their responsibilities to provide the behavior specified by the framework.

Also assume that there is a client base of this organization that wants to use the framework-level classes (i.e., high-level abstract superclasses) from these subsystems to ensure they comply with the overall organization framework. In addition, they want to utilize both framework-level and concrete-level classes in an existing business model subsystem, and customize their user-interface model differently than the one used by the organization.

Until recently, few options were available to handle this situation. If the organization and its clients were using just a Smalltalk (without Team/V) environment, then the only option was for the organization to give its clients the whole Smalltalk image, containing the base Smalltalk image, user-interface classes (abstract superclasses), and business model classes (framework- and concrete-level classes). If the organization and its clients were using Smalltalk with Team/V, then clients could access business model subsystem classes using the individual “package migration” technique⁵ or by creating specific tools (visual or textual) on top of Team/V to handle “migrating a list of packages.” These approaches are on the verge of obsolete for a variety of reasons including limitations encountered during application packaging and maintenance phases; the Parcplace-Digitalk merger, which is resulting in a new architecture; and an overall industry shift toward componentware.

DYNAMIC LINK LIBRARIES AND SLLs

OS/2 provides a powerful way of bundling your application program into a coherent unit called a *dynamic link library* (DLL). As the name suggests, a DLL provides services that are accessed and linked dynamically by different application programs. It provides a way for an application program to dynamically reference and access functions and resources outside its own executable environment.⁶ Such resources might be icons, bitmaps, or pointers, etc., whereas functions could be the PM application programming interface (API) calls for handling window management, graphics, device driver routines, or accessing OS/2 executable programs, etc. Smalltalk/V PM with the use of SLLs takes this concept one level higher, allowing one to access the services provided by individual subsystems. By tapping the power of OS/2’s DLL mechanism underneath it, Smalltalk/V PM allows one to create SLLs of classes, methods, and metaclasses that can be shared by different teams within an organization. Object libraries were a common means of building DLL files in Digitalk Smalltalk previous to Version 3.0.1. SLLs are those object libraries with a new face.

DELIVERING A SMALLTALK APPLICATION

In Smalltalk/V PM, the Smalltalk image consists of the executable environment (v.exe file, change.log file, recover.log file), one or more DLL files, and one or more SLLs. DLLs are divided into *base class DLLs* and *development class DLLs*. Base class DLLs contain classes such as collections, streams, windows, etc., whereas development class DLLs contain the Smalltalk compiler, debugger, etc.⁶ A Smalltalk programmer uses the development classes to create a Smalltalk application program. As development progresses, the code created by the programmer (once saved) is added into the v.exe file, which begins to grow. Typically a programmer uses the v.exe file to deliver the application. This approach works well whether the design contains a relatively small or large number of classes. Then, by making appropriate changes in the

#startUpApplication method in the NotificationManager class, the programmer uses v.exe to start the application. This standard approach always works.

Another way to deliver a Smalltalk application is filing out all the classes from one programmer’s image and installing them on another programmer’s image.

Both approaches are cumbersome from the application maintenance point of view. A better way is to build Smalltalk libraries of classes to provide flexibility in delivering Smalltalk applications.

DIFFERENT WAYS TO CONSTRUCT SLLs

Before transferring the executable to users, programmers test their applications to confirm that they meet requirements. It might be a good idea to start building SLLs at the initial application development phase, to ease the future maintenance task. Such SLLs can be loaded in a development or runtime environment. Here, the task of Smalltalk library construction can be accomplished in a variety of ways. One could categorize it based on different aspects such as the existing services, different subsystem frameworks, or available standalone classes (explained next). Referring back to our example, assume that a simple business model for the organization consists of classes such as Person, Address, HomeAddress, BusinessAddress, Phone, HomePhone, BusinessPhone, etc., while the user-interface model consists of classes such as UserInterfaceModel, UserInterfaceModelForOrganization, and Person InformationWindowDialog (to display Person information). Combined, these different classes could provide a “view” service that enables one to view information about persons.

Construction based on the service behavior

The organization could treat the aforementioned business model and user-interface classes as one entity, and construct a single Smalltalk library representing a Person information “view” service, i.e., construct one Smalltalk library that contains business model subsystem classes (abstract and concrete) and user-interface subsystem classes (abstract and concrete) to provide this specific service. If the organization needs a “change” service, consisting of additional user-interface classes such as ChangePersonInformationDialog (to change the Person information) and other business objects created to handle the changes, the organization could then construct another Smalltalk library representing a “Person information change service.” In general, this approach facilitates development efforts within the organization.

Smalltalk library for view service:

BusinessModel Subsystem (abstract and concrete classes) and UserInterfaceModel Subsystem Smalltalk Library (abstract and concrete classes related to viewing the person)

Smalltalk library for change service:

BusinessModel Subsystem (abstract and concrete

classes) and UserInterfaceModel Subsystem Smalltalk library (abstract and concrete classes related to changing the person information)

Construction based on framework implementation

In construction based on framework implementation, the organization treats the aforementioned business model classes and user-interface classes as two different entities and constructs two separate SLLs. Thus, by constructing two SLLs for business model subsystem classes (abstract and concrete) and user-interface subsystem classes (abstract classes), the organization creates a server environment that provides services such as knowing the state of the business objects and providing user-interface protocols to display the current state of the business objects. The organization and its client base use this server environment to customize user-interfaces appropriately. In general, this approach is best used when the organization has to satisfy client base requirements.

Server SLLs:

BusinessModel Subsystem Smalltalk library	UserInterfaceModel Subsystem Smalltalk library
---	--

Client SLLs:

UserInterfaceModel Subsystem Smalltalk library (concrete classes for organization)

UserInterfaceModel Subsystem Smalltalk library (concrete classes for clients)

Construction based on standalone classes

One can construct a Smalltalk library of standalone classes that don't belong to a particular subsystem but are required by different subsystems, such as classes for managing application configuration, setting application environment, other helper classes, etc. This Smalltalk library can then be treated as a standalone component in the application.

CREATING A BUSINESS MODEL SMALLTALK LIBRARY

Assume that the organization follows the second approach to constructing a Smalltalk library, which results in their having two DLLs—one for business model classes and one for user-interface classes.

The typical steps to create a business model Smalltalk library are described below:

1. Open the Library Builder dialog. Select the package containing business model subsystem classes. Select the following menu option: Module —> Build Library.
2. The Library Builder dialog offers two options. Customize the classes that you would like to add into the library by clicking on the Customize option or just let create an SLL for the classes contained within the selected pack-

age/cluster. Also, one can optionally include the source code for classes not in SLL one is creating.

BINDING SLLs TO A SMALLTALK IMAGE

The Digitalk help manual describes three ways to package Smalltalk application using SLLs; these approaches let the developer bind SLLs statically or dynamically.

The first approach is to bind SLLs during startup by including their names in an autobind ascii file, e.g., app.bnd (during development image, it looks for vdev.bnd file). This approach permits one to save the image without binding it with the SLL, thus avoiding having to bind the SLL to a specific version of v.exe.

The second approach is to bind the SLL dynamically. The developer can then bind and unbind SLLs on demand, which results in low memory overhead.

Finally, one can bind SLLs in a hybrid way using a combination of the aforementioned two approaches: binding some SLLs to the image and binding others dynamically.

ADVANTAGES OF USING SLLs

1. Data sharing—Multiple applications access and share subsystem SLLs. A server environment is created by storing different SLLs on the network. All teams within the organization would then have ready access to it so that consistent access is maintained. Thus, SLLs make data sharing a transparent process across multiple applications.
2. Pluggability—Modular components are created to enhance reusability. Thus, the maintenance task becomes flexible. This aids easy replacement and shipping of appropriate subsystem DLLs.
3. Application of the producer/consumer concept—This key concept of componentware (producing and consuming the components) is easily adopted and applied. Referring to the example at the beginning of this article, the organization becomes a producer of SLLs and the client base becomes a consumer of SLLs.
4. Decreased image size—This conserves hard disk space and reduces v.exe size, resulting in less overhead.
5. Flexibility of use for the organization and its clients—Once the business model Smalltalk library is created, it is ready for distribution to the client base, as well as use by the organization itself. The client base, which is uninterested in the user interface Smalltalk library (containing classes such as UserInterfaceForOrganization, its subclasses, composite panes used in dialogs, etc.), can load the business model Smalltalk library in their development/runtime environment and begin using it alone.
6. Creation of standalone class libraries—Components consisting of standalone classes are constructed and distributed appropriately.
7. Realization of the framework benefits—During the

The task of Smalltalk library construction can be accomplished in a variety of ways.

design phase, classes (both framework- and concrete-level) are logically grouped to form a subsystem based on the intended behavior that a subsystem is supposed to perform. The correct decision to group a certain class in a particular subsystem aids the task of building and maintaining SLLs.

8. Construction simplicity—Previous Digitalk Smalltalk versions used the concept of object libraries, and Smalltalk developers spent a lot of time constructing them. On the other hand, SLLs are constructed simply and quickly.
9. Platform portability between OS/2 and Win32 operating systems—SLL uses a unique and system-independent format.⁷ Hence, it is easier to create applications that are portable between these two platforms.
10. Use in runtime and development environments—Works excellently in both runtime and development Smalltalk environments.
11. Scripting—During a release phase, a particular Smalltalk library may have to be reconstructed many times. By taking advantage of scripting facilities,⁷ the task of constructing SLLs is simplified by creating scripts to reconstruct SLLs.

DISADVANTAGES OF USING SLLS

1. Difficult to use in a Team/V development environment—If you load a Smalltalk library in a Team/V development environment, it loads all the classes in the “unpackaged” package and not where they belong. Because most of the source code related to Team/V classes is hidden, it is difficult to ascertain how to replicate the “Load/Migrate” action (which loads all the classes in a particular package/cluster in one’s Smalltalk image) during Smalltalk library loading so that classes included in the Smalltalk library will fall into packages where they belong.
2. Inability to extend existing Smalltalk library environ-

ment—To build a Smalltalk library, Digitalk uses a few classes (SmalltalkLibrary, SmalltalkLibraryBind, TeamVLibraryInformation, TeamVInterface, etc.) whose implementation is hidden to the developers. Because they cannot access the code associated behind the methods, they cannot add any enhancements to base Smalltalk library classes.

CONCLUSION

This article reviewed different approaches to constructing SLLs and described advantages to facilitate the Smalltalk application delivery task. It provided shared transparent access to different teams by conserving hard disk space.

I found that the ability to create SLLs provides a pluggable approach that facilitates application maintenance tasks. The article also showed how with this approach, the momentum gained in developing Smalltalk applications is retained while delivering Smalltalk applications.

Acknowledgment

The author thanks Anne Marie Frederick at Prudential Insurance Corporation in New Jersey for encouraging him to write this article. ■

References

1. Johnson, R. and B. Foote. Designing reusable classes. JOURNAL OF OBJECT ORIENTED PROGRAMMING 1(2):22–35, 1988.
2. Taligent. BUILDING OBJECT-ORIENTED FRAMEWORKS, 1993.
3. Harris, J. Object Insider: Breaking out of the object ghetto, OBJECT MAGAZINE 4(8):12–14, 1995.
4. Johnson, R. How to develop frameworks, TUTORIAL NOTES OF OOPSLA’93, 1993.
5. Digitalk Inc. DIGITALK PROGRAMMING REFERENCE FOR SMALLTALK/V FOR OS/2, 1993.
6. Petzold, C. PROGRAMMING THE OS/2 PRESENTATION MANAGER, Microsoft Press, 1989.
7. Digitalk Inc. DIGITALK ONLINE HELP MANUAL FOR SMALLTALK/V FOR OS/2, 1995.

Makarand Utpat is a Senior Consultant at Envision in St. Louis, MO.

*Classes are logically
grouped to form a subsystem
based on the intended
behavior that a subsystem
is supposed to perform.*



Alan Knight

Principles of OO design: or, everything I needed to know in life, I learned from Dilbert*

EVERYONE KNOWS THAT objects and object-oriented (OO) design are the hottest things since sliced bread (and, of course, slices of bread are objects). The problem is that it's hard to agree on what exactly they are. There have been many attempts to define principles of OO design or coding, with varying degrees of success. In my opinion, most of them suffer from two flaws. First, they don't tell me enough about how to code. Reading a definition of "polymorphism" doesn't tell me how to exploit it in my programs. Second, and more important, is that they're dull. Even if the definition of polymorphism did tell me how to code, it's hard to stay awake long enough to finish reading it.

Therefore, I modestly present some of my own principles of OO-ness, which I hope address both of these flaws. Furthermore, I believe that these principles relate well to the corporate environments that have seen so much Smalltalk use recently.

NEVER DO ANY WORK THAT YOU CAN GET SOMEONE ELSE TO DO FOR YOU

This is always good advice, but it's particularly applicable in OO. In fact, I consider it the fundamental principle of OO. As an object, my responsibilities are very clearly defined, and so are those of my co-workers. If something is (or ought to be) one of their responsibilities, then I shouldn't be trying to do that work myself.

Let's look at a concrete example:

```
total := 0
aPlant billings do: [:each |
  (each status == #paid and: [each date > startDate])
  ifTrue: [total := total + each amount]].
```

versus

```
total := aPlant totalBillingsPaidSince: startDate.
```

Alan Knight has had great success avoiding responsibility with The Object People, 885 Meadowlands Dr. East, Ottawa, Ontario, K2C 3N2. He can be reached at 613.225.8812 or by email as knight@acm.org.

In the first case, we're asking the plant for all of its billings, figuring out for ourselves which ones qualify, and computing the total. That's a lot of work, and almost none of it is our job. Far better to use the second option, where we simply ask for something to be done and get a result back. In real-world terms, the first example is like the following conversation:

"Excuse me, Smithers. I need to know the total bills that have been paid so far this quarter. No, don't trouble yourself. If you'll just lend me the key to your filing cabinet I can go through all the records myself. I'm not familiar with your new filing system, but I'm sure I can figure it out. I'll try not to make too much of a mess."

Smithers actually understands his filing system, so he can probably do the work faster than we can, and he's much less likely to mess everything up. In attempting to do his job for him, we're just making things worse. Things will get worse when he switches over to that new filing system next week. We'd be far better off acting like a stereotypical tyrant boss.

"SMITHERS! I need the total bills that have been paid since the beginning of the quarter. No, I'm not interested in the petty details of your filing system. I want the total, and I'll expect it on my desk within the next half millisecond."

Let's look at a simpler example, which is all too common.

```
somebody clients add: Client new.
```

versus

```
somebody addClient: Client new.
```

There's always a temptation to choose the first option, because it saves writing a couple of methods that do nothing but add and delete on the other class. But you know it's wrong. You're trying to do somebody's work for

* Dilbert is a trademark of United Feature Syndicate.

them, and ultimately it's only going to cause problems. Writing those extra methods keeps the responsibility where it belongs and will make the code cleaner in the long run.

This principle is close to the more conventional idea of "encapsulation", but I like to think it makes the idea somewhat clearer. I often see people who are happily manipulating the internal state of another object, but think it's OK because they're doing it all through messages. Encapsulation is not just about accessing state, it's about responsibilities. Responsibility is about who gets stuck doing the real work.

AVOID RESPONSIBILITY

If responsibilities are about getting stuck with work, it's important to avoid them. This has some important corollaries:

- If you must accept a responsibility, keep it as vague as possible.
- For any responsibility you accept, try to pass the real work off to somebody else.

Our first principle tells us to take advantage of other objects when writing code. We also have to avoid being taken advantage of. Any time I (as an object) am tempted to accept a responsibility, I should ask myself, "Is this really my job?" and "Can't I get someone else to do this?"

If I do accept a responsibility, it's important to keep it as vague as possible. If I'm lucky, this vagueness will help me avoid doing the work later. Even if I must do the work, it may allow me to take some shortcuts without anybody else noticing.

For example, I've seen CRCs with responsibilities like:

Maintain a collection of the whosits to be framified

This is much too specific. My job isn't to maintain a collection, it's to be able to report, when necessary, which whosits need framification. That may be implemented by maintaining a collection, or by asking one or more other objects for their collection(s), it may be hard-coded, or computed dynamically as `Whosit allInstances select: .` Regardless of which option I choose, there shouldn't be any impact on my responsibilities.

My preference for phrasing a responsibility of this kind is:

Know which ...

but I'm flexible as long as the phrasing is suitably vague. I'd probably be even happier with

"Be able to report which ..."

Carried to the extreme, it seems this could lead to the situation where everyone passes information around and nothing ever gets done. Exactly. Object bureaucracy at it's finest.

Seriously, a good OO system can actually approach this state. Each object will do a seemingly insignificant amount of work but somehow they add up to something much larger. You can trace through the system, seeking the place where a certain calculation happens, only to realize that the calculation is finished and you just didn't notice it happening.

POSTPONE DECISIONS

The great virtue of software is *flexibility*. One way we achieve flexibility is through late binding. We most often discuss late binding between a method name and the method it

invokes, but it's also important in other contexts. When faced with a decision, we can gain flexibility by postponing it. The remaining code just needs to be made flexible enough to deal with any of the possible outcomes.

The ideal is when we can avoid making the decision at all, leaving it up to someone else (the end user, other objects). For example, consider the question of how to implement dictionaries. The standard thing to do is use a hash table. That works well for medium-sized collections, but it's a waste of space and effort for very small collections. For very large collections, it may also be wasteful, particularly if the number of elements exceeds the resolution of our hash function. We must make a decision here, so we'd like to postpone it or pass it off to someone else.

Some implementations of the collection classes do precisely this. The collections transfer much of their behavior to an implementation collection that actually does the work. Depending on the size, the nature of that collection can change. In VisualAge 2.0, small dictionaries were stored as arrays because the overhead of hashing was more than the cost of a linear search. Larger dictionaries could be represented as either normal or bucketed hash tables. This seems to have disappeared in 3.0, so I suppose the overhead of this mechanism became more than the cost of using a single representation. Visual Smalltalk also has dictionaries that are capable of switching between normal and bucketed hash tables.

Be careful in applying this principle because it's possible to take it too far. Decisions aren't just sources of problems, they give us the power to solve problems. Because we cannot solve all the problems of the world at once, we make the decision to limit ourselves, and we make assumptions about the problems we'll be given. The problem arises when our decisions were poor, or our assumptions don't hold any more. The trick is to make

continued on page 28

THE BEST OF COMPIANG.SMALLTALK

continued from page 14

enough decisions to be able to work, but few enough that our code doesn't become brittle. That's one of the things that makes software difficult.

Passing off decisions to another object is often referred to as using policy or strategy objects. This is discussed in DESIGN PATTERNS¹ as the Strategy pattern.

Other related ideas are "Open Implementations," which can allow important decisions to be postponed so far that even the end user of the module can control them. I can't do justice to this topic here, but there's a web page available at <http://www.xerox.com/PARC/spl/eca/oi.html>

Because web pages change so rapidly, I'll also mention that I found it using the search terms *open implementation* and *Gregor Kiczales* (the project leader).

POSTSCRIPT

Although there is a significant element of humor in these principles, I do take them quite seriously and urge you to do the same. They illustrate some very important aspects of OO design and coding. I've even come up with enough of them to fill another column, so the next issue will continue this theme. ☞

Reference

1. Gamma, E. et al. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1994.

MANAGING OBJECTS

continued from page 17

person can cause damage more quickly on a Smalltalk project than they can on a traditional project, and corporate cultural checks that normally help such people, such as peer reviews, management one-on-one meetings, and performance reviews, are tuned to the slower beat of the traditional project.

Beginning a Smalltalk project offers the opportunity for a "behavioral context switch," in which old patterns can be broken. By catching behavioral difficulties early, you can keep them from becoming established patterns. Once behavioral patterns are established, their impact on productivity must be carefully monitored and humanely dealt with. ☞

References

1. Kroeger, O. and J.M. Thuesen. TYPE TALK AT WORK, HOW THE 16 PERSONALITY TYPES DETERMINE YOUR SUCCESS ON THE JOB, Tilden Press, New York, 1992. [*This book concentrates on applying Jungian personality type theory in the workplace, and is much more approachable than defining works on the topic.*]
2. Bramson, R.M. COPING WITH DIFFICULT PEOPLE, Anchor Press/Doubleday, Garden City, NY, 1981.
3. Brooks, Jr., F.P. THE MYTHICAL MAN-MONTH (20th anniversary ed.), Addison-Wesley, Reading, MA, 1995. [*A wonderful classic.*]



Jan Steinman



Barbara Yates

“Special” team members

LET’S FACE IT—WE’RE ALL “DIFFICULT” at some time or other. This can range from mild irritability over a bad hair day, to active sabotage between competing groups. These problems are best dealt with before they develop into a pattern of behavior, but the pace of Smalltalk development often results in people settling into behavioral patterns before anyone notices.

We divide special team members into three categories:

1. “Pluses” offer net productivity but can be much more productive if their unique strengths can be exploited while reducing the impact of their weaknesses.
2. “Zeros” are a wash and can be tolerated while your organization finds a place where they can become pluses.
3. “Minuses” detract from the productivity of others and can seriously impact a project if not dealt with in some way.

Keep in mind that we are writing about established behavior *patterns* here. Obviously, new assignments, emotional problems, family crises, etc., make the best of us “zeros” or even “minuses” from time to time, and a compassionate organization will help, or at least tolerate, these non-chronic productivity losses.

Here are some of the more prevalent behavior patterns we’ve found in Smalltalk projects and suggestions for dealing with them.

“THE LONER”

This person is an enigma to management. the Loner is often a Meyers—Briggs¹ “INTP” type, who may be perceived as “not a team player,” and might even be fired if he wasn’t so damned creative. Like Bramson’s² “Analyst,” the Loner will often miss deadlines, not because he isn’t working, but simply because he is *still* working!

The big danger on a Smalltalk project is that the Loner

may disappear after having been given an assignment and come up for air several months later with a beautifully crafted, complete solution to the wrong problem. Because Smalltalk is so productive, it may be tempting to redefine the project around the Loner’s wonderful solution of the wrong problem, because he may well be far ahead of the rest of the team, who’ve been busy collaborating all these months!

At worst, a single Loner is a “zero,” but two or more on a team may quickly destroy a project if not guided by a skilled architect. Once you’ve discovered a Loner on your team, there are several techniques you can use to harvest his creativity without yielding control of the project:

- Schedule regular peer review, especially at the design level, before the Loner is able to write reams of code.
- If the Loner is also a Know-It-All (discussed later), call these peer reviews “educational reviews” to avoid wounding his fragile ego.
- Assign the Loner a “shadow,” “buddy,” or “stunt-double”—someone who keeps up-to-date on what the Loner is doing, in case it is necessary to fill in in an emergency and provide the communication that the Loner is unable to provide.
- Limit Loners to well-specified, well-defined tasks. This is a last resort, because junior people will not break their Loner habit and senior people will get bored and possibly become Slackers (discussed later).

“THE LOANER”

While discussing the Loner, we realized we have, on several occasions, experienced its pun! For various reasons, the project is running late and senior management decides they had better round up nine people so they can ship this baby in a month.

Almost 20 years ago, Frederick Brooks, Jr.³ noticed that adding resources to a late project makes it even later. Loaners often consume more time than they add, and are, therefore, “minuses,” because they need to be integrated with the team’s procedures and conventions, and also absorb all the history that has gone into getting to this point of crisis. This is not so much a reflection on the

Jan Steinman and Barbara Yates are cofounders of Bytesmiths, a technical services company that has been helping companies adopt Smalltalk since 1987. Between them, they have over 20 years Smalltalk experience. They can be reached at Barbara.Bytesmiths@acm.org or Jan.Bytesmiths@acm.org, or via <http://www.bytesmiths.com>.

person as it is on the process that put them in this unfortunate situation.

This problem is amplified by Smalltalk because Smalltalk is not a language, it is an environment. Not only must this person learn your project before they can be useful, they often must learn exotic (to them) concepts, such as Dictionaries—concepts for which your team has developed a shared vision.

Even more insidious is the possibility that you might not be getting quality material to begin with. Think of it—if asked to loan one of your team members, would you give your best person, or perhaps someone whom you haven't quite been able to find a place for yet?

If your manager insists on doing you this “favor,” keep the following in mind:

- Make sure they are doing you a favor—don't accept someone else's problem when you're already in schedule trouble!
- Keep loan lengths on the long side to better amortize the “interest” cost of borrowing a person—don't take a Loaner for less than six months, unless the person has enough history with your project to hit the ground running.
- Your Loaner must either be Smalltalk-knowledgeable, or must be able to contribute without ever touching Smalltalk. Growing Smalltalk talent is too much of an investment to return when the loan is due!
- Make sure Loaners document their work so others can pick up where they leave off.
- Assign them a “stunt double,” who will work with them on a day-to-day basis. (Be aware that too much “time suckage” from the double may turn a Loaner into a “minus.”)

“THE COWBOY”

The Cowboy typically learned Smalltalk in relative isolation and is used to being “king of the image”. Cowboys delight in tricky code, sometimes doing it for sheer intellectual pleasure without the slightest rationale.

The Cowboy's nemesis is ENVY/Developer, because he doesn't like people looking at his tricky code, he can't imagine others actually *working* on his tricky code, and absolutely hates the constraints imposed by a code management system—if changing the implementation of basicNew suits his purpose, he cannot tolerate the thought of getting the permission of the Library Supervisor!

Cowboys can be wonderful “pluses” if carefully managed; they can also be “minuses” if they consistently destabilize your environment or if their escapades consume an entire “stunt-double” resource. To deal with the Cowboy, try the following:

- Use and enforce your code management system's security features. This includes passwords for all accounts

and no shared accounts, especially privileged accounts such as ENVY's Library Supervisor.

- Never, ever let the Cowboy use a privileged account to work on the base image!
- Find tasks for Cowboys that exploit their curious nature—some tasks demand tricky code!
- Establish a culture where the only tricky code tolerated is well documented, complete with the rationale for being tricky.
- The Cowboy is often a Loner, and some of those coping strategies, such as extensive peer-review and “stunt-double” coverage, work well for him also.

*If asked to loan one of
your team members, would
you give your best person, or
perhaps someone whom you
haven't quite been able
to find a place for yet?*

“THE SLACKER” OR “ROBINSON CRUSOE”

The Slacker often knows the best web sites and is fluent on the latest Usenet newsgroup gossip. He may often quickly collapse a window as you approach his desk and you may notice his long print jobs that are totally unrelated to work. When others are at his desk, they often seem to be doing the typing or mousing.

We sometimes call this pattern “Robinson Crusoe” because it seems that Slackers always expect to have their work done by Friday, even though they haven't started it by Thursday. (And if on Friday they are inconveniently stranded on some desert isle, Slackers are perfectly content to arrange for other team members to pick up the slack!)

The Slacker never meets a deadline and never works a full week, but neither does he ever report that he is behind schedule and, of course, there is always “The Good Excuse.”

Slackers come in two varieties: Dumb and Lazy, and Bored. It is difficult to distinguish between them but the difference is vital:

- A Dumb and Lazy Slacker is in over his head but won't admit it and doesn't really care. He may become a minor “plus” if given a simpler task.
- A Bored Slacker is in well *under* his head and may become a major “plus” if properly inspired.

If you do not raise The Slacker to at least a “zero,” your project will suffer much more than the mere loss of effective head count. Sometimes you can do this by the following:

- Give Slackers additional training or mentoring. Put a hard limit on this “time suckage,” and let Slackers know it or their mentors will merely end up doing all the work.
- Give Slackers many intermediate deliverables, which may help determine whether they are Dumb and Lazy or merely Bored.
- Micromanage the Slackers with daily progress checks, but recognize that this activity alone may take enough of your time to keep them a “minus.”

Often, despite your best efforts, a Dumb and Lazy Slacker

cannot be raised to a "plus." This cannot be tolerated and the person must go. If removing a Dumb and Lazy Slacker from your project is not possible, you need to minimize his impact on your team.

- Isolate the Slacker; forbid him to seek help, and forbid others to help him with his work.
- Perhaps you can turn your Slacker into someone else's Loaner? (Nah, we wouldn't suggest that!)

"THE KNOW-IT-ALL"

This person often actually knows a lot, but the Know-It-All's insecurity causes them to "know" more than they actually do. (In the immortal words of Bo Diddley, "It ain't what you don't know; it's what you know that just ain't so!") We've found this often results from taking someone who has been the "big cheese" on a traditional project and immersing them in Smalltalk, which is strange, different, and frightening to someone who has become used to being an acknowledged expert.

This is the only pattern that Bramson also uses, and he divides them into two categories: the "Bulldozers" and "Balloons," the primary difference being that "Bulldozers" know what they're talking about whereas "Balloons" do not. Of the two, "Bulldozers" are merely obnoxious—although they may demoralize others with their strong assertions, they are still strong "pluses," even in context of the entire team. We're more concerned with "Balloons," who can lead an entire project astray if they have the ear of someone important!

Don't let the insecurity of the Know-It-Alls blind you to what they can be contributing. To deal with the Know-It-All try the following:

- Be quick to acknowledge and reward the greatness of Know-It-Alls when you know they are right—give them strokes freely when they deserve it and they will be less likely to seek strokes for false knowledge.
- Ease Know-It-Alls out of their comfort zone—carve off a bit of the project, such as designing C primitives or RDBMS access, which will allow them to use their expertise while slowly coming to grips with Smalltalk.
- A Know-It-All can be responsible about his or her lack of knowledge when not threatened and may do well if assigned a junior "buddy" to mentor. The mentoring can surreptitiously become two-way, especially if the junior person is farther along the Smalltalk learning curve, but monitor them carefully to make sure The Know-It-All is not filling an impressionable mind with puffery.

CONCLUSION

There are very few truly useless people in this world, but there are many people who are viewed in light of their weaknesses, rather than being put to work using their strengths.

As Smalltalk amplifies this problem, an out-of-place

continued on page 28

INFO@SIGS

SIGS Publications, Inc., 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

ARTICLE SUBMISSION

To submit articles for publication, please contact:
John Pugh & Paul White, Editors, 885 Meadowlands Dr. #509,
Ottawa, Ontario, K2C 3N2 Canada;
email: streport@objectpeople.on.ca

PRODUCT REVIEWS AND ANNOUNCEMENTS

To submit product reviews or product announcements, please contact the Editors at the address above.

CUSTOMER SERVICE

For customer service in the US, please contact PO Box 5050, Brentwood, TN 37024-5050; 800.361.1279;
Fax: 615.370.4845; in the UK, please contact Subscriptions Department, Tower Publishing Services, Tower House, Sovereign Park, Market Harborough, Leicestershire, LE16 9EF, UK; +44.(0)1858.435302; Fax: +44.(0)1858.434958

SIGS BOOKS

For information on any SIGS book, contact: Don Jackson, Director of Books, SIGS Books, Inc., 71 West 23rd Street, New York, NY 10010; 212.242.7447; Fax: 212.242.7574; email: donald_jackson@sigs.com

SIGS CONFERENCES

For information on all SIGS Conferences, please contact: SIGS Conferences, 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7515; Fax: 212.242.7578; email: info@sigs.com

BACK ISSUES

To order back issues, please contact: Back Issue Order Department, SIGS Publications, 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

REPRINTS

For information on ordering reprints, please contact: Reprint Management Services, 505 East Airport Road, Box 5363, Lancaster, PA 17601; 717.560.2001; Fax: 717.560.2063

ADVERTISING

For ad information for any SIGS publication, please contact:
East Coast/Europe: Gary Portie
Central US: Elisa Marcus
Recruitment: Michael Peck
Exhibit Sales, West Coast: Kristin Viksnins
Exhibit Sales, East Coast: Sarah Olszewski
212.242.7447; Fax: 212.242.7574
email: sales@sigs.com
West Coast: Diane Fuller
408.255.2991; Fax: 408.255.2992
email: dhfsigs@hooked.net

SIGS HOME PAGE

Access the SIGS Home Page at: <http://www.sigs.com>.

THE BEST OF COMPIANG.SMALLTALK

continued from page 14

enough decisions to be able to work, but few enough that our code doesn't become brittle. That's one of the things that makes software difficult.

Passing off decisions to another object is often referred to as using policy or strategy objects. This is discussed in DESIGN PATTERNS¹ as the Strategy pattern.

Other related ideas are "Open Implementations," which can allow important decisions to be postponed so far that even the end user of the module can control them. I can't do justice to this topic here, but there's a web page available at <http://www.xerox.com/PARC/spl/eca/oi.html>

Because web pages change so rapidly, I'll also mention that I found it using the search terms *open implementation* and *Gregor Kiczales* (the project leader).

POSTSCRIPT

Although there is a significant element of humor in these principles, I do take them quite seriously and urge you to do the same. They illustrate some very important aspects of OO design and coding. I've even come up with enough of them to fill another column, so the next issue will continue this theme. ☐

Reference

1. Gamma, E. et al. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1994.

MANAGING OBJECTS

continued from page 17

person can cause damage more quickly on a Smalltalk project than they can on a traditional project, and corporate cultural checks that normally help such people, such as peer reviews, management one-on-one meetings, and performance reviews, are tuned to the slower beat of the traditional project.

Beginning a Smalltalk project offers the opportunity for a "behavioral context switch," in which old patterns can be broken. By catching behavioral difficulties early, you can keep them from becoming established patterns. Once behavioral patterns are established, their impact on productivity must be carefully monitored and humanely dealt with. ☐

References

1. Kroeger, O. and J.M. Thuesen. TYPE TALK AT WORK, HOW THE 16 PERSONALITY TYPES DETERMINE YOUR SUCCESS ON THE JOB, Tilden Press, New York, 1992. [*This book concentrates on applying Jungian personality type theory in the workplace, and is much more approachable than defining works on the topic.*]
2. Bramson, R.M. COPING WITH DIFFICULT PEOPLE, Anchor Press/Doubleday, Garden City, NY, 1981.
3. Brooks, Jr., F.P. THE MYTHICAL MAN-MONTH (20th anniversary ed.), Addison-Wesley, Reading, MA, 1995. [*A wonderful classic.*]



Jay Almarode

Mechanisms for application partitioning

MY LAST COLUMN described the differences between client Smalltalk systems and server Smalltalk, and how server Smalltalk fits into the three-tier architecture that is emerging to meet the performance and business requirements of enterprise-wide applications. The key to balancing the processing load between clients and server, and sharing business objects in such architectures, is the ability to partition applications.

Application partitioning is the activity in which code written for the client can be moved to the server (or vice versa). When both the client and server can execute the same Smalltalk code, this movement of objects and code is much simpler and allows the application to be dynamically tuned in the face of changing hardware and software. Applications can be developed initially only on the client, and then portions can be moved to the server to share objects, enforce security policy, and gain fault tolerance of critical data as needed.

When the clients and server speak a different language, this partitioning should occur earlier in the design, because partitioning decisions are more difficult and costly to change later. Unfortunately, performance tuning often occurs late in the software process so, in many cases, the decision to repartition an application must balance the cost of reimplementing large sections of code against the expected performance gain. Making such changes is discouraged because the cost of repartitioning is higher. Also, developers must be proficient in two different languages, one for client development, and one for the server.

When Smalltalk is the language on both the client and server, what mechanisms are available to partition the application and to distribute objects and behavior between the client and server? In GemStone Smalltalk, there are several mechanisms available so that client applications can reference and manipulate objects located on the server. One mechanism is *forwarders*. A forwarder is a client object that covers for a server object. A forwarder does not contain any state of the server object, but maintains enough information to communicate with the server object when needed. When a message is sent to a forwarder, the execution of its behavior actually takes

place on the server. The forwarder knows the identity of the server object and how to communicate with it.

Forwarders are implemented in such a way that no special code is required to check for the presence of a forwarder before sending it a message. Forwarders utilize Smalltalk's message-sending mechanism to automatically forward messages by special handling of the `doesNotUnderstand:` error. The Forwarder class does not inherit from class Object, so forwarders understand very few messages on the client side. Most messages to a forwarder are silently trapped by the execution thread on the client, forwarded to the server for execution, and the result returned to the client for continuation of its execution thread. If the message to a forwarder contains arguments, those arguments are transformed automatically into server objects if needed. This is implemented in such a way so that application code does not have to be written any differently, whether the receiver of a message is a forwarder or some other client object.

There are several ways a programmer can get a forwarder to a server object. One way is to send the message `beForwarder` to a replicate (discussed later). For example, a newly created client object could be copied to the server by sending it the message `putInGS` (thus, making it a replicate), then send the message `beForwarder`. At that point, the state of the object is only stored in the server Smalltalk, and any messages sent to the client object cause execution in the server. In some cases, a developer may design the application so that all instances of a particular client class are intended to be forwarders. Whenever an instance of such a class is fetched from the server, it should be instantiated as a forwarder. This is specified by implementing the method `instancesAreForwarders` in the client class to return true. Two other ways to specify that certain objects are to be manifested as forwarders in the client Smalltalk is by a *replication specification* (described later) or by a *connector*. A connector is a mechanism connecting certain client objects with certain server objects at the time the client logs into the server. There are many different kinds of connectors: some that connect classes and some that connect class variables; there are those that connect class instance variables; those that connect objects by name; and those that connect objects by identity. Each type of connectors allows a developer to specify that the client object is to be manifested as a forwarder.

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@slc.com.

Another mechanism to manipulate server objects on the client is with *replicates*. A replicate is a copy of a server object that resides on the client. Some or all of the state of the server object is copied in the replicate, and when a message is sent to a replicate, the execution of its behavior takes place on the client. Using replicates requires that a mapping be defined between classes on the client and classes on the server. At its simplest, this mapping can specify that a server class maps to a client class with the same name. This is the default mapping. You can also specify more complex mappings as objects are translated between client and server. This is done by reimplementing the `instVarMap` method in the class of the replicate. This method should return nested arrays, where each sub-array contains an instance variable name and a specification of how it should be mapped. This allows a developer to handle reordering, renaming, or omission of instance variables when an object moves from one domain to the other.

A key consideration when using replicates is the amount of synchronization that occurs between the client replicate and its corresponding server object. There are messages available to the application developer to explicitly manage keeping the two objects in sync. However, it is much easier to let the interface layer that manages replicates be responsible for keeping the state of client and server objects in sync. In this way, a replicate always accurately reflects the state of the server object (based on the current transaction's point of view), wherever the object is used in the application. This level of synchronization, called *full transparency*, is configurable by class. To enable full synchronization for a class, send it the message `makeGSTransparent`.

When replicates are used in full transparency mode, then modifications to replicates are managed automatically. When the application modifies a replicate, it is automatically marked dirty and changes are flushed to the server at the appropriate time. For example, modifications to replicates are flushed to the server before any server behavior is executed or when the transaction is committed. When other users modify and commit changes to server objects, those changes are not seen in the replicate until the current transaction is committed or aborted. At this time, the replicate is eligible to have its state updated from the server, a behavior called *faulting*. Ordinarily, the replicate will not be faulted until it is next accessed. However, this default behavior can be overridden by implementing a method called `faultPolicy` for the class of the replicate. This method should return `#immediate` if the replicate should be faulted immediately when the next transaction begins. It is also possible to cause additional application code to be executed before or after the replicate is faulted by implementing a `preFault` and `postFault` method.

An important consideration when programming with replicates is how to control the replication of composite

objects (objects with nested subobjects). Some client applications may only need a portion of the state of the server object, so why send more to the client than is needed? When a replicate is being instantiated from a server object, an application wants to control which instance variables are retrieved and in what form the objects referenced by those instance variables are created (as forwarders or replicates). In addition, if the instance variable is assigned a replicate, the application may also want to specify how many levels deep to replicate. To exercise this control, a developer implements the `replicationSpec` method for the class of the replicate. This method returns nested arrays where each subarray contains the name of an instance variable and a specification of how it is to be instantiated. The developer has the option to specify whether the instance variable is to be instantiated as a

replicate, a forwarder, or a stub (discussed later). If the instance variable is to be created as a replicate, the developer can specify a minimum or maximum number of levels to replicate as well. The following example shows the implementation of the `replicationSpec` method for class `Employee`, where the name instance variable is replicated, the address instance variable is replicated to at

least level 2, and the department instance variable is created as a forwarder.

```
classmethod: Employee
```

```
replicationSpec
```

```
"Return nested arrays specifying how Employees are to be replicated."
```

```
^ super replicationSpec ,
  #( (name replicate)
    (address min 2)
    (department forwarder) )
```

In cases where not all of a composite object is copied into the client, some placeholder object must take the place of each object that remained on the server. This object, called a "stub," maintains information concerning its corresponding object on the server. When a stub is sent a message, it has the ability to create a replicate, replace the stub with the new replicate, and then resend the message to the replicate. This happens transparently to the end user, so application code does not have to test for the presence of a stub object. It is also possible to turn a replicate into a stub object. This is desirable if you want to free the space taken up by the replicate and its subobjects. You can do this by sending the message `stubYourself` to a replicate.

The mechanisms just described can be utilized in several ways to partition and fine-tune an application for maximum performance in a client/server environment. Developers can exercise greater control over where execution of object behavior takes place and how much data is transferred to the client. ■

Modifications to replicates are flushed to the server before any server behavior is executed.

Editors' Corner



John Pugh



Paul White

Has Smalltalk passed its prime? Will it remain relevant as we move to distributed and Web-based applications? Clearly, if Smalltalk is to prosper in its second quarter-century, it must evolve to remain a language of choice for developers of corporate applications. The good news is that the Smalltalk vendors are among the first out of the starting gate with tools for supporting highly distributed applications and building live Web applications. Here is a short preview of what's available in the new releases from the Smalltalk vendors and what's in store for the future. In upcoming issues of THE SMALLTALK REPORT, we will endeavour to explore both of these areas in much more detail.

To date, Smalltalk has most often been used in two-tier client/server applications involving server-based relational database management systems. As the trend toward three-tier architectures and truly distributed applications accelerates, Smalltalk must provide much enhanced support for distributed computing. There are many issues to be faced when developing truly distributed applications. In a Smalltalk development context, for example, there are the issues of inspecting, debugging, and garbage collection when objects are distributed across Smalltalk images. In addition, new tools are required to assist developers in determining how an application should be distributed and how such an application should be tuned for performance.

Both IBM (Distributed Smalltalk) and ParcPlace-Digitalk (ServerWorks) plan to offer support for distribution between Smalltalk objects residing in different images and differing operating environments, as well as access to object request brokers (such as DSOM in the case of the current release of IBM Distributed Smalltalk or HP Distributed Smalltalk's CORBA-compliant ORB) to provide access to non-Smalltalk objects. IBM shipped its Distributed Smalltalk along with the 3.0 release of VisualAge for Smalltalk in November, while ParcPlace-Digitalk will ship the first release of ServerWorks sometime this year. IBM (for obvious reasons) and ParcPlace were slow off the


mark in support for OLE; Digitalk and VMark with their Object Studio product have led the way. Now all the vendors are promising enhanced support in the near future.

The excitement generated by Java and similar technologies has reached fever pitch in the last few months as organizations seek to realize the potential of client/server software on the Internet. We can't count the number of times we have been asked: "What will be the impact of Java on the use of Smalltalk?" The implication in the question is that Java is somehow a threat to Smalltalk's very existence! We (and the

Smalltalk vendors, we are sure) view the WWW as a marvelous opportunity for Smalltalk to demonstrate the power of the marriage between object technology and the deployment vehicle the Web provides. For people in organizations with existing Smalltalk applications who are accustomed to the development environment and class libraries underlying all the Smalltalk dialects, the move to Web-based

applications should be a fairly smooth transition. Indeed, it will likely be quite some time before Java-related technologies provide application development environments with features equivalent to those of Smalltalk. Moreover, there are plenty of opportunities for synergy between Smalltalk-based Web technology and Java.

Both IBM and ParcPlace-Digitalk have demonstrated web technology at recent shows. VisualWave from ParcPlace-Digitalk, based on the VisualWorks Smalltalk environment, is the first product to ship and promises developers the opportunity to go beyond the existing "brochure duplication" and "form-filling" characteristics of most Web sites to provide highly interactive applications giving the feel of a reactive two-way conversation.

The beauty of Smalltalk is that by its very nature it is highly extensible. While there are many challenges to overcome, Smalltalk is well-positioned to provide elegant transparent solutions to the challenges of the future and, what's more, to get there ahead of the crowd. 

*It will likely be
quite some time before
Java-related technologies
provide application
development environments
with features equivalent
to those of Smalltalk.*

Editors

John Pugh and Paul White
Carleton University & The Object People

SIGS Publications Advisory Board

Tom Atwood, *Object Design*
François Bancelhon, *O, Technology*
Grady Booch, *Rational*
George Bosworth, *ParcPlace-Digital*
Jesse Michael Chonoles, *Lockheed Martin ACC*
Stuart Frost, *SELECT Software*
Adele Goldberg, *ParcPlace-Digital*
Thomas Keffer, *Rogue Wave Software*
R. Jordan Kriendler, *IBM Consulting Group*
Thomas Love, *Consultant*
Bertrand Meyer, *ISE*
Meilir Page-Jones, *Wayland Systems*
Cliff Reeves, *IBM*
Bjarne Stroustrup, *AT&T Bell Labs*
Dave Thomas, *Object Technology International*

The Smalltalk Report

Editorial Board

Jim Anderson, *ParcPlace-Digital*
Adele Goldberg, *ParcPlace-Digital*
Reed Phillips
Mike Taylor, *ParcPlace-Digital*
Dave Thomas, *Object Technology International*

Columnists

Jay Almarode, *GemStone Systems Inc.*
Kent Beck, *First Class Software*
Juanita Ewing, *ParcPlace-Digital*
Bob Hinkle, *Consultant*
Tim Howard, *FH Protocol, Inc.*
Ralph E. Johnson, *University of Illinois*
Alan Knight, *The Object People*
Mark Lorenz, *Hatteras Software, Inc.*
Jan Steinman, *Bytesmiths*
Rebecca Wirfs-Brock, *ParcPlace-Digital*
Barbara Yates, *Bytesmiths*

SIGS Publications Group, Inc.

Richard P. Friedman, Founder, President, and CEO
Hal Avery, Group Publisher
John McCormick, Editorial Director

Editorial/Production

Elizabeth A. Upp, Managing Editor
Elisa Varian, Production Manager
Andrea Cammarata, Art Director
Kathleen M. Major, Sr. Production Editor
Sue Mycka, Desktop Designer
Margaret Conti, Advertising Production Coordinator
Shannon Smith, Editorial Production Assistant

Circulation

Elayne Glick, Circulation Director
Lawrence E. Hoffer, Marketing Manager
Byron Scarlett, Assistant Circulation Manager

Advertising/Marketing

Gary Portie, Advertising Manager, East Coast/Canada/Europe
Elisa Marcus, Advertising Manager, Central US
Michael W. Peck, Advertising Representative
Kristine Viksnins, West Coast Exhibit Sales
Sarah Olszewski, East Coast Exhibit Sales

212.242.7447 (v), 212.242.7574 (f)
Diane Fuller & Associates, Sales Representative, West Coast
408.255.2991 (v), 408.255.2992 (f)

Wendy Dinbokowitz, Promotions Manager for Magazines

Administration

Margherita R. Monck, General Manager
David Chatterpaul, Senior Accounting Manager
Bibi Budhram, Accounts Payable



PUBLISHERS OF JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, THE SMALLTALK REPORT, THE X JOURNAL, REPORT ON OBJECT ANALYSIS & DESIGN, JAVA REPORT, OBJECT CURRENTS (ONLINE), OBJECT EXPERT (UK), and OBJECTSPEKTRUM (GERMANY)

Features

A performance challenge

4

Keith Piraino

Don't assume Smalltalk isn't fast enough for your whole application. The performance gap can be narrowed to the point that it need not be a factor in your technology selection—as this first-person account demonstrates.

A framework for multiple language support

12

William Hollings

Software applications today are sometimes required to support users who communicate in a variety of languages. The author discusses the multiple language support framework his team is developing in Visual Smalltalk.

Tactical patterns for the real world—Instantiation patterns

21

Darrow Kirkpatrick

Creating competitive shrink-wrapped software for Windows led to the development of these tactical patterns for working with domain models. This series begins with a family of instantiation patterns.

Sequential key allocation strategies in Smalltalk

24

Dayle Woolston & Chris Kesler

The right approach for generating sequential keys depends on factors such as system administrative policy, application features, and overall performance expectations. A number of strategies are presented and discussed.

Columns



Managing Objects

16

The demo trap

Jan Steinman & Barbara Yates

Smalltalk's legendary productivity is a two-edged sword—corporate cultures that are used to thinking “GUI is hard” may assume the project is done upon seeing the first prototype, or may keep you “churning GUI.”



Getting Real

18

Tuning multi-user Smalltalk

Jay Almarode

Understanding the reading and writing characteristics of your application and clustering objects that are frequently used together can help streamline access in multi-user Smalltalk.



The Best of comp.lang.smalltalk

20

More principles of OO design

Alan Knight

Managers don't do real work, delayed optimization is best, and other “rules of life” that are equally applicable in the realm of OO programming and design.

Departments

Editors' Corner

2

Recruitment

29

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar–Apr, July–Aug, and Nov–Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1996 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Bulk rate U.S. postage paid Lancaster, PA, permit 161. Canada Post International Publications Mail Product Sales Agreement No. 290386.

Individual Subscription rates 1 year (9 issues): domestic \$89; Mexico and Canada \$114, Foreign \$129; Institutional/Library rates: domestic \$199, Canada & Mexico \$224, Foreign \$239. To submit articles, please send electronic files on disk to the Editors at 885 Meadows Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

POSTMASTER: Send domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sig.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

A performance challenge

Keith Piraino

WHEN MY COMPANY began looking into Smalltalk we had quite a few skeptics. One challenged me to match the performance of the most time-consuming portion of an existing C-based application. At first, I resisted—I knew that Smalltalk was not as fast as C but felt that its' many other strengths greatly outweighed this weakness. I did my best to steer this effort toward a more complex problem that would better demonstrate the strong modeling capabilities of Smalltalk.

It became clear, however, that performance concerns would torpedo Smalltalk adoption unless these fears were met head on. I accepted the performance challenge and this article describes my experience in trying to meet it.

The problem was reading data from binary files (proprietary structure) and aggregating it into a multi-dimensional symmetrical matrix.* I used VisualWorks 2.0 running under Microsoft Windows 3.11.

DICTIONARY

I decided to use a Dictionary to represent my matrix. The keys are the cell coordinates and the values are the cells. I defined the matrix as follows:

```
Object subclass: #SymmetricalMatrix
  instance variables: 'dimensionSizes initialCellValue
                     cellDictionary'
```

dimensionSizes is a collection with the number of elements in each dimension. initialCellValue contains the value to initialize each cell to. cellDictionary is the actual Dictionary of cells. I added a class method to answer a new instance and an instance method to create the actual matrix:

```
SymmetricalMatrix class>>withSizes: aCollection
  initializeCellsTo: aValue
    ^(self new)
      dimensionSizes: aCollection copy;
      initialCellValue: aValue;
      build; yourself

SymmetricalMatrix>>build
  self cellDictionary: Dictionary new
```

For example, "SymmetricalMatrix withSizes: #(2 3 4) initialize

CellsTo: 0" would answer a 2x3x4 matrix where the default value of all 24 cells is zero.

To access the cells we implement #at:, #at:put:, and #at:incrementBy. The first two selectors should look familiar. The last is a convenience method to increment the value of a cell. In all three cases the at: parameter is a collection representing the coordinates of the cell. In our 2x3x4 example "at: #(1 1 1)" will answer the first cell and "at: #(2 3 4)" will answer the last cell. Here is the implementation of the #at: method:

```
at: aCollection
  ^self cellDictionary at: aCollection
  ifAbsent: (self initialCellValue)
```

Dictionary's #at:ifAbsent: method is used to answer the cell at the coordinates or return our initialCellValue if the cell doesn't exist yet. From this the #at:put: and #at:incrementBy methods are trivial. The only catch is to make sure we copy aCollection in #at:put: because if the caller modifies it our dictionary key will change. Here is the code:

```
at: aCollection put: aValue
  self cellDictionary at: (aCollection copy) put: aValue

at: aCollection incrementBy: aValue
  self at: aCollection put: (self at: aCollection) + aValue
```

That's it. Other than accessor methods the code shown is a complete implementation of the matrix. It works, but how fast is it?

The test consisted of creating a three-dimensional matrix and calling #at:incrementBy: over a million times. The coordinates to use were retrieved from binary files in a proprietary format.†

The cells were initialized to zero and always incremented by one. If the C program took X amount of time my first cut in Smalltalk took over 4X (i.e., 4 times slower). A profile of the code showed that most of the time was spent in Dictionary code—#at: and #at:put:. Leveraging the Dictionary class gave me a solution very quickly but there was no room for improvement so I turned to another approach.

LINEAR

The C application uses a linear array to implement the matrix. A 2x3x4 matrix, for example, is represented by

*The current C application can handle up to 6 dimensions. Symmetrical meaning that the number of elements in a dimension is constant across all combinations of the other dimensions. If, for example, Region and Brand were two of the dimensions in a matrix, the number of brands can't change between regions.

†Because I'm not presenting the code to "feed" the matrix, all times given are after that section of code was already optimized. In reality, both the matrix and supporting code were optimized simultaneously.

A PERFORMANCE CHALLENGE

an Array with a size of 24. Given the coordinates of the cell we need to calculate the index in the linear array. One key optimization here is to pre-compute dimension ranges. The range is the number of cells we pass when the dimension index changes. It's calculated by taking the product of the inner dimensions. In a 2x3x4 matrix the range of the first dimension is 12 (3 * 4), the range of the second is 4 and the range of the last is 1. For example, the index of #(1 1 1) is 1 and the index of #(2 1 1) is 13. The difference between them is the range for the first dimension: 12. The code following assumes that cellArray and dimension Ranges are instance variables containing the Array of cells and the Array of dimension ranges, respectively. Initializing these variables is left as an exercise for the reader.

```
at: aCollection incrementBy: aValue
| index |
index := self cellIndexFor: aCollection.
self cellArray at: index
  put: (self cellArray at: index) + aValue

cellIndexFor: aCollection
| index |
index := 1.
aCollection with: self dimensionRanges do:
  [:eachCoordinate :eachRange |
    index := index + ((eachCoordinate - 1) * eachRange)].
^index
```

At 2.5X, the linear matrix was an improvement over the dictionary approach but it was still pretty far off. As I would expect, the profiler showed almost all the time being spent in #cellIndexFor:. Apparently, this calculation was a lot slower in Smalltalk than in C. Other than caching dimension ranges, I don't know of any other optimizations to this algorithm. I still wanted to meet the challenge so I moved on yet again.

TREE

To avoid the cost of calculating an index, I decided to implement a matrix as a tree structure where intermediate nodes are a kind of index but the actual cells are all in leaf nodes. Figure 1 shows an Object Explorer picture of a matrix created via the following code fragment:

```
matrix := TreeMatrix withSizes: #(2 3 4) initializeCellsTo: 0.
matrix at: #(1 1 2) put: 5.
matrix at: #(2 3 4) incrementBy: 1.
```

The code to create the initialized tree structure is left as an exercise for the reader. To access a particular cell, we must traverse the tree to get to the correct leaf node. The #leafNodeContaining: method answers the leaf node containing the cell defined by the given coordinates. From there it's a simple #at:put: to the Array. Here's the code:

```
at: aCollection incrementBy: aValue
| leaf |
leaf := self leafNodeContaining: aCollection.
leaf at: aCollection last
```

```
put: (leaf at: aCollection last) + aValue.
```

```
leafNodeContaining: aCollection
^(aCollection copyFrom: 1 to: (self numDimensions - 1))
  inject: (self rootNode) into: [:node :each | node at: each]

numDimensions
^self dimensionSizes size
```

The TreeMatrix reduced the time to about 2.1X: only slightly better than the linear approach. Having run out of ideas, I did what I should have done earlier—looked closely at the profile results. Most of the time was spent in #leafNodeContaining:, but not performing the #at: that traverses the tree. There was some overhead in #copyFrom:to: but most of the time was spent in the #do: loop[†]called by #inject:into:. The overhead of looping was far greater than what I was actually doing in the loop!

I went back to LinearMatrix and, sure enough, most of the time was spent in #with:do:. The #with:do: method creates a Stream to traverse the second collection. That occupied some time, but again, most of the time was spent in the #do: loop. I was dead wrong when I assumed that calculating the cell index, the addition and multiplication necessary was the problem. With this revelation I could

have returned to optimizing LinearMatrix but I decided to stick with TreeMatrix because it was slightly faster even discounting the #do: impact.

LOOP UNROLLING

If the problem is looping, I figured why not eliminate the loop? The #leafNodeContaining: method can easily be optimized for a particular number of dimensions. One way to do this would be to create a different subclass for each number of dimensions and override #leafNodeContaining: in each. A more manageable approach is to use blocks. When the matrix is instantiated, a block optimized for that number of dimensions is assigned to the instance variable leafAccessorBlock. Given the cell coordinates and the root node this block goes directly to the leaf node via a series of #at: messages. Listed next are the one-, two-, and three-dimensional blocks:

```
[:coord :root | root]
[:coord :root | root at: (coord at: 1)]
[:coord :root | (root at: (coord at: 1)) at: (coord at: 2)]
```

Modifying #at:incrementBy: to use the leafAccessorBlock dramatically reduced the time to about 1.1X. A fresh profile showed that doing "aCollection last" twice to access the index of the cell in the leaf node was now taking a noticeable amount of time. I was able to take advantage of the fact that the position of this last coordinate is known at the time the matrix is instantiated. After adding an instance variable to cache the number of dimensions in numDimensions, at:incrementBy: was changed to:

```
at: aCollection incrementBy: aValue
```

[†] I use the term "loop" somewhat loosely to also include enumerating the elements of a collection.

A PERFORMANCE CHALLENGE

```
| leaf lastCoordinate |
leaf := leafAccessorBlock value: aCollection
      value: self rootNode.
lastCoordinate := aCollection at: self numDimensions.
leaf at: lastCoordinate
      put: (leaf at: lastCoordinate) + aValue
```

This brought the time down to 0.9X. Actually, a little faster than the C code! Success!!

I'm not suggesting that Smalltalk is faster than C in a head-to-head comparison. There are mitigating factors[§] that prevent me from making that claim even in this case. The real point is that the performance gap can be narrowed to the point where it will only rarely be the deciding factor in technology selection.

To generalize, this idea of “unrolling loops via blocks” is useful in the following circumstances:

- Looping code takes more time than contents of loop
- Number of iterations varies but the maximum is relatively low
- Contents of loop relatively simple

It's fine for the problem at hand but the last two constraints limit the usefulness of this technique. If the problem involved 100 iterations with even 5 lines of code in the loop, this approach becomes completely unwieldy. At this point I released my findings and declared success. Although I wasn't completely satisfied with my solution, I didn't have any more time to spend on it.

A BETTER LOOP?

Several months later I returned to this problem. I reviewed back issues of the *Smalltalk Report* looking for any performance information regarding looping. I found the answer I was seeking in an article on performance by Alan Knight.¹

It turns out that certain kinds of blocks are inlined and others aren't. These optimizations are vendor specific but can have a major impact. In *LinearMatrix* I used `#with:do:` for looping and in *TreeMatrix* I used `#inject:into:`. Neither is inlined and both call `#do:` which is also not inlined. In *VisualWorks* both `#whileTrue:` and `#to:do:` are inlined. To take advantage of this, I rewrote `#leafNodeContaining:` as shown:

```
leafNodeContaining: aCollection
| node |
[node:= self rootNode
 1 to: self numDimensions - 1 do:
  [:i | node := node at: (aCollection at: i)].
 ^node
```

I tested this and received results similar to unrolling the loop: 0.8X. The only cautionary note here is to avoid add-

ing parentheses to improve readability. If you change this code to read “(1 to: self numDimensions -1) do:” it will run much slower. Why? Because instead of the compiler recognizing `#to:do:` and inlining, it instantiates an *Interval* via `#to:` and then sends the *Interval* the `#do:` message. I was actually doing this in some of the code that feeds data to the matrix. When I removed the parentheses the time was further reduced to 0.5X (i.e., twice as fast as the C code!!)

Optimizing the loop via `#to:do:` is better than unrolling it for two reasons: (a) it is more generally applicable; (b) perhaps more important, the code is closer to the original intent and, hence, much more readable. In fact, although slightly less concise, the `#to:do:` version may be more easily understandable than my original version that used `#copyFrom:to:` and `#inject:into:`.

Letting go of code and ideas you've invested in isn't always easy. More than once I've seen developers refuse to do it. Sometimes they think they've gone too far to back off. This is understandable but usually misguided in light of long-term maintenance costs and the malleability possible with newer tools. A less defensible cause of this refusal is emotional attachment. People get excited about their first idea (which is good) but sometimes become blind to newer and better ones (which is bad). If you write something you think is really cool but you've since found a simpler more maintainable approach—file out the cool code and play with it in your spare time. Don't leave it in a production application.

Good systems come, in part, from a willingness to throw away some of your code. Vendors can provide the tools to rework code and enlightened management can provide the time but it simply doesn't work without people with the right attitudes.

MISCELLANEOUS

In the course of optimizing your code be careful not to break it. An approach that works for me is to try out optimizations in subclasses. Once you finish optimizing you can decide whether or not to consolidate. Next is the hierarchy of *Matrix* classes I ended up creating:

```
AbstractSymmetricalMatrix
DictionaryMatrix
LinearMatrix
  LinearMatrixFastLoop
TreeMatrix
  TreeMatrixUnrollLoop
  TreeMatrixFastLoop
```

For the particular test I used, *TreeMatrix* gave the best performance. When I tried a test with more dimensions, however, *LinearMatrix* performed better than *TreeMatrix*. The best approach might be to instantiate the optimum *Matrix* using the *Bridge* pattern² in the same way that the *VisualAge* *Collection* classes are implemented.³

In the code that feeds data to the matrix I found blocks useful as a way to avoid reevaluating conditionals in tight

[§] My Smalltalk version has only a subset of the C functionality. This means that the C program must pass through a lot more conditionals and code even for my simple test. Compounding this is the fact that Smalltalk is running in 32 bits (via win32s) and the C code is a 16 bit app.

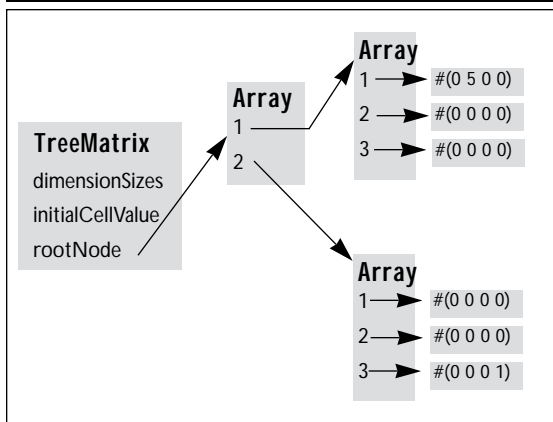



Figure 1. A 2x3x4 TreeMatrix.

loops. If there are more than two possibilities, this can help eliminate case-like statements.

One area where I didn't need to worry about performance was reading from the binary files. ExternalRead Stream seems to be plenty fast. My only problem was that methods like #nextLong assume a Little-Endian byte order and my data is Big-Endian. Correcting this wasn't a big deal but it's ironic since VisualWorks itself seems to store Integers in the image in Big-Endian order.

CONCLUSION

Smalltalk's expressive nature, extensive class library, and interactive environment greatly improve the most important performance measure—*developer productivity*. You can quickly create a solution to the problem. The combination of hardware advances (faster, more memory) and Smalltalk vendor advances (dynamic compilation, faster library code) mean that you may never have to worry about performance. Don't assume Smalltalk isn't fast enough for your whole application.

If performance is an issue, use a profiler to locate the small percentage of code that needs work. It's easy to do and works a lot better than intuition. Always keep in mind the maintenance cost of tweaking code for performance. You might want to consider trying another algorithm/design approach instead. This will often give you greater performance leverage and more understandable code. It's also a lot easier to do in Smalltalk than in other environments. Again, don't resist throwing code away—it's an important part of the process. 

References

1. Knight, A. More performance tips, THE SMALLTALK REPORT 4(2): 19-20, 1994.
2. Gamma, E. et al. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1995.
3. Lalonde, W. and Pugh, J. Communicating reusable designs via patterns, JOOP 7(8): 69-71, 1995.

Keith Piraino can be reached at keith_piraino@npd.com.

A framework for multiple language support

William Hollings

FOR MANY SOFTWARE applications, communicating with the user in a single language (usually English) is sufficient. However, some applications, such as those supporting customers in an urban banking environment, must communicate with users from diverse language backgrounds. These applications must be designed to support dynamically changing languages at the user interface. For example, Figure 1 illustrates a single test window that has been opened in both English and French as determined by the user's language preference selection.

I refer to this concept of dynamic language selection as *multiple language support (MLS)*, to differentiate it from the more common *national language support (NLS)* which assumes the use of only a single language. This article describes parts of the multiple language framework that our team at the Toronto-Dominion Bank is developing using Digitalk's Visual Smalltalk.

THE LANGUAGEMANAGER CLASS

Central to the MLS framework is a subclass of `NationalLanguageSupport` called `LanguageManager`. A singleton instance of `LanguageManager` is plugged into the existing global variable `NationalLanguage` at application start up time.

In addition to the responsibilities it inherits, `LanguageManager` adds the capability to manage *language files*. Each language file contains the information required to translate and format all on-screen text within the application into a particular language. We can incrementally add new languages simply by distributing new language files without redeveloping the application.

The `LanguageManager` singleton manages these language files (with help from `ObjectFiler`) via two public methods:

- `LanguageManager>>getSupportedLanguages`. Answers a collection of language names that are currently supported by the application. During application startup, the `LanguageManager` singleton scans the directory for all language files (e.g., *.lng) to build this collection of language names.
- `LanguageManager>>setLanguage: aLanguageName`. Sets the current language to the one identified by `aLanguageName`. This method loads the contents of the appropriate language file from disk. This message is typically received from a user preferences selection tool.

Each language file contains a dictionary to translate wid-

get labels, menu labels, and other strings. It also contains various data formatting information such as the decimal separator character and date formats. Once loaded from file, this information is maintained in instance variables within the `LanguageManager` singleton. In particular, the instance variable `stringDictionary` is populated with the dictionary of translated strings.

The keys for `stringDictionary` are language-neutral string abbreviations. In the case of widgets and menu items, the label text that was assigned to the widget or menu item at GUI, design time is used as the key (e.g., 'CloseButn'). In the case of other displayable strings, such as those displayed in message boxes, the key is a language-neutral abbreviation such as 'ErrMsgComm073'.

Typically, the language dictionary for each language is created and maintained in a spreadsheet. It is read into the development image and saved into the language object file using additional `LanguageManager` methods.

Some points regarding performance are in order here. If the `stringDictionary` gets too big, the dictionary lookup times may become unacceptably slow. Also, the use of `Strings` as dictionary keys is less efficient than using `Symbols` as keys to an `IdentityDictionary`.

One solution to the first problem is to factor the dictionary into smaller dictionaries. Our production application uses three dictionaries within the `LanguageManager` singleton, one each for widget labels, menu labels, and general strings. Other methods of factoring are possible, however for simplicity, the framework described here has only one language dictionary.

With respect to the second issue, the choice of `Strings` as keys in `stringDictionary` was motivated by the fact that the GUI environment and tools assume the use of `Strings` for the names and labels of widgets and menus. A design using `Symbols` would require hacking the GUI environment and tools. Such a design would also have to avoid using the `String>>asSymbol` method, which performs its own (larger and longer) string-keyed dictionary look up. The `String` keys have proved to be fast enough in our application, though this remains an area for potential performance improvement.

TRANSLATING STRINGS

The primary collaborator with `LanguageManager` is the `String` class itself. `Strings` respond to the `asMLSString` mes-

sage and simply delegate the translation work to the LanguageManager singleton as follows:

```
String>>asMLSString
"Answers my translation in the current language."
^NationalLanguage translateString: self
```

which is handled by LanguageManager as:

```
LanguageManager>>translateString: keyString
"Answer the translation of the string keyString."
^self stringDictionary at: keyString ifAbsent: [
    keyString ]
```

Notice that the original key string is returned if a translation string could not be found. This allows the application to work even if some or all translations are missing from the language file. This comes in handy during development when the GUI and language files are in a state of flux.

TRANSLATING WIDGETS

The translation of widgets and menu labels takes place during the opening of a window. We added the following method to the TopPane class:

```
TopPane>>translateWindow
"Tell myself, all my widgets, and my menu bar to
translate themselves."
self translate.
```

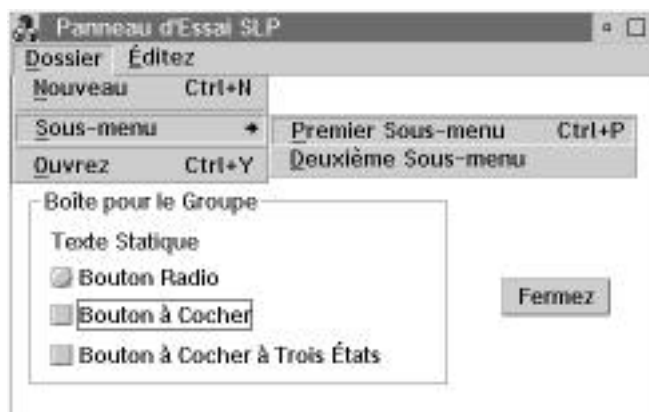
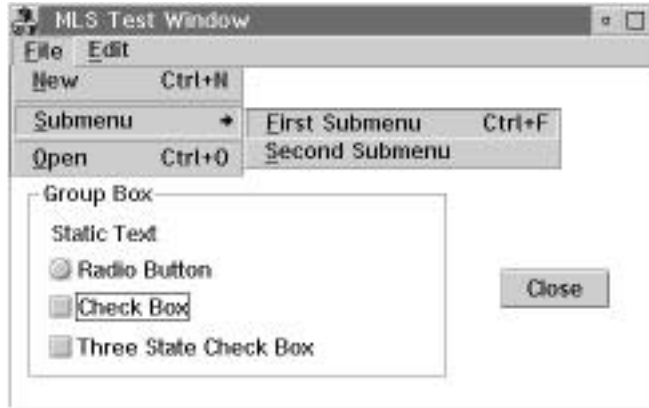


Figure 1. Illustration of a single test window opened in both English and French. Notice that even the menu accelerator keys can sometimes be different.

```
self allChildrenDo: [ :each | each translate ].
self menuWindow translate.
```

which sends the translate message first to itself, then to all of the widgets contained on the window, and then to the window's menu bar.

The TopPane>>translateWindow message is sent after the widgets and menus have been created as objects but before they have been made visible. The question of which object sends this message depends on which GUI builder is being used. Under WindowBuilder, a good place to send this message is in the preInitWindow method of the ViewManager subclasses. Under PARTS, a subclass of PARTSWindowPart can be created to override the open method so that the window translates itself before opening:

```
MLSPARTSWindowPart>>open
"Translate myself before opening."
self translateWindow.
^super open
```

In our MLS framework, all objects respond to the translate message. The default method (defined in Object) does nothing. All widgets and menus with a displayable label override this default method to specifically translate their label text. For widgets and windows this is done in the ControlPane and TopPane classes, respectively, and is the same for both classes:

```
TopPane>>translate
ControlPane>>translate
"Translate my label according to the current language."
self label: (self label asMLSString).
```

In this method, the widget retrieves its existing language-neutral label string and tells it to translate itself. The resulting translated string is then assigned back to the label. To stop nil labels from breaking the system we also added an UndefinedObject>>asMLSString method which simply answers nil.

TRANSLATING MENUS

Translating menu labels is a little more complex because of the need to register menu selection accelerator keys (e.g., - Ctrl+S), which may be different for each language.

Recall that the TopPane>>translateWindow method sends the translate message to the menu bar. In a non-PARTS application, the menu bar is an instance of MenuWindow and it simply passes the message on to its component menus:

```
MenuWindow>>translate
"Tell each of my menus to translate itself."
self menus do: [ :each | each translate ].
```

The Menu>>translate method first translates its own title (e.g., File, Edit, etc.) and then cycles through each of its

INFO@SIGS

SIGS Publications, Inc., 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

ARTICLE SUBMISSION

To submit articles for publication, please contact:
John Pugh & Paul White, Editors, 885 Meadowlands Dr. #509,
Ottawa, Ontario, K2C 3N2 Canada;
email: streport@objectpeople.on.ca

PRODUCT REVIEWS AND ANNOUNCEMENTS

To submit product reviews or product announcements,
please contact the Editors at the address above.

CUSTOMER SERVICE

For customer service in the US, please contact PO Box
5050, Brentwood, TN 37024-5050; 800.361.1279;
Fax: 615.370.4845; in the UK, please contact Subscriptions
Department, Tower Publishing Services, Tower House,
Sovereign Park, Market Harborough, Leicestershire, LE16
9EF, UK; +44.(0)1858.435302; Fax: +44.(0)1858.434958

SIGS BOOKS

For information on any SIGS book, contact: Don Jackson,
Director of Books, SIGS Books, Inc., 71 West 23rd Street,
New York, NY 10010; 212.242.7447; Fax: 212.242.7574;
email: donald_jackson@sigs.com

SIGS CONFERENCES

For information on all SIGS Conferences, please contact:
SIGS Conferences, 71 West 23rd Street, 3rd Floor, New York,
NY 10010; 212.242.7515; Fax: 212.242.7578;
email: info@sigs.com

BACK ISSUES

To order back issues, please contact: Back Issue Order
Department, SIGS Publications, 71 West 23rd Street, 3rd
Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

REPRINTS

For information on ordering reprints, please contact:
Reprint Management Services, 505 East Airport Road, Box
5363, Lancaster, PA 17601; 717.560.2001; Fax: 717.560.2063

ADVERTISING

For ad information for any SIGS publication, please contact:
East Coast/Europe: Gary Portie
Central US: Elisa Marcus
Recruitment: Michael Peck
Exhibit Sales, West Coast: Kristin Viksnins
Exhibit Sales, East Coast: Sarah Olszewski
212.242.7447; Fax: 212.242.7574
email: sales@sigs.com
West Coast: Diane Fuller
408.255.2991; Fax: 408.255.2992
email: dhfsigs@hooked.net

SIGS HOME PAGE

Access the SIGS Home Page at: <http://www.sigs.com>.

menu items telling each menu item to translate itself
before setting the accelerator key for the menu item:

```
Menu>>translate
```

```
"Translate my title, then translate my menu items."
```

```
self title: ( self title asMLString ).
```

```
self translateItems.
```

```
Menu>>translateItems
```

```
"Tell each menuItem to translate itself and then set its  
accelerator key."
```

```
self items do: [ :each |  
    each translate.  
    self setAccelKeyOf
```

From here, the MenuItem>>translate method translates its
own label and then tells any submenu (which would be an
instance of Menu) to translate itself.

```
MenuItem>>translate
```

```
"Translate my label text and if I have a submenu,  
translate it."
```

```
self label: ( self label asMLString ).
```

```
self submenu translate.
```

Note that because the default implementation of the
translate method in Object does nothing, this method will
work correctly even if the submenu is nil (i.e., there is no
submenu).

Finally, the MenuItem>>setAccelKeyOf: method is where
things get a little complicated and algorithmic. This meth-
od extracts the accelerator substring from the menu item
label (e.g., 'Ctrl+S') and then parses the components of this
substring to convert them into a key code and bit flags.
These are in turn inserted into an array of accelerators that
is maintained by the Menu instance. This code is as follows:

```
MenuItem>>setAccelKeyOf: mlItem
```

```
| lbl tabIdx accelString itemIdx bits bitsString key  
keyString |
```

```
( lbl := mlItem label ) isNil ifTrue: [ ^self ].
```

```
(tabIdx := lbl indexOf: Tab) > 0 ifFalse: [ ^self ].
```

```
accelString := ReadStream on: ( lbl copyFrom: (tabIdx  
+ 1) to: lbl size ).
```

```
accelString isEmpty ifTrue: [ ^self ].
```

```
itemIdx := self items indexOf: mlItem ifAbsent: [ ^self ].
```

```
bitsString := accelString upTo: $+.
```

```
keyString := accelString upTo: $+.
```

```
keyString first isDigit
```

```
ifTrue: [
```

```
    key := 0.
```

```
    1 to: keyString size do: [:i |
```

```
        key := key * 10 + ( keyString at: i ) digitValue ].
```

```
    bits := AfVirtualkey ]
```

```
ifFalse: [
```

```
    key := keyString first.
```

```
    bits := AfChar ].
```

```
( bitsString includes: $C ) ifTrue: [ bits := bits |
    AfControl ].
( bitsString includes: $A ) ifTrue: [ bits := bits | AfAlt ].
( bitsString includes: $S ) ifTrue: [ bits := bits | AfShift ].
accel at: itemIdx put: (self accelArray: key accelBits: bits).
```

Although all of the Menu and MenuItem methods described apply to both PARTS and non-PARTS development, the structure of the menu bar is slightly different under PARTS. PARTS keeps the menu titles separate from the actual menus, and the Menu instances are not attached to the PARTSMenuBar (a subclass of MenuWindow) instance until immediately before the window is opened. As a result, PARTSMenuBar requires a different translate method:

```
PARTSMenuBar>>translate
    "Tell each menu title and Menu to translate itself."
    self children do: [ :each | each translate ].
    self partApplication componentDictionary
        do: [ :each |
            each isPARTSMenuItem
                ifTrue: [ each menuObject translateItems ] ].
```

This method first translates the menu titles (accessed via self children). It then locates each instance of Menu in the PARTS application controlling the window and tells each of them to translate their menu items.

Incidentally, don't use PARTSMenuBar>>translate as an example of good programming practice. In a production application, we should add methods to both PARTSApplication and PARTSMenuItem to reduce the coupling in the PARTSMenuBar>>translate method. Currently, this method must know that the PARTSApplication has a componentDictionary that contains instances of PARTSMenuItem, which in turn holds on to instances of Menu. I cheated a bit here to reduce the amount of code required for this article.

TRANSLATING MESSAGE BOX STRINGS

Because any string can be translated, we created a new message box that accepts language-neutral abbreviation strings instead of raw text. These are then translated into the current user language. Therefore, instead of coding the following:

```
MessageBox warning: 'This action will destroy the
    known universe.'
```

we would code:

```
MLSMsgBox warning: 'WarnUniverseByeBye'
```


which would be translated via String>>asMLSSString before the message box was displayed.

Unfortunately, it is not sufficient to simply create MLSMsgBox as a subclass of MessageBox because that class relies on native OS message boxes, which use their own text for the 'Yes', 'No', 'OK' and 'Cancel' buttons. We built MLSMsgBox (and other utility windows such as MLSPrompter) from scratch.

OTHER ISSUES

The format of numbers displayed as on-screen text or in entry fields varies from language to language (e.g., \$1,000.00 is displayed in some languages as 1.000,00\$). To handle this, we have added methods such as Number>>asMLSSString which formats the number with the appropriate "thousands" and decimal separators.

For Help files, we maintain a separate Help file for each supported language. When a new language is selected the LanguageManager singleton renames the Help files so that the one associated with the newly selected language will be used by the Help system.

Finally, I have certainly not exhausted the issues surrounding full MLS support in this brief article. The framework described does not translate the text on any window that is already open. This would involve tagging all menu components with a name and rebuilding the menu accelerator key tables on the fly. I also did not address the formidable challenge of supporting text *input* in multiple languages, which touches on issues as diverse as physical keyboards and database storage. These framework extensions are left as an exercise for the reader. 

William Hollings is a Smalltalk architect and consultant in Toronto. He is currently helping the Toronto-Dominion Bank develop brokerage and banking applications. He can be reached at hollings@inforamp.net.

Tactical patterns for the real world: Instantiation patterns

Darrow Kirkpatrick

BETWEEN DESIGN AND code lies a set of important issues that must be mastered to ship a product out into the real world of paying endusers. This article is the first in a three-part series that deals with efficiently implementing and managing domain models using patterns discovered in the course of creating engineering products for numerical modeling.

Domain models are the fundamental objects that represent the business logic of an application and store the state underlying the user interface. The patterns presented in this series of articles form a family of generic approaches for working with these domain models: setting their attributes, validating them, presenting information about them, and optimizing them for size and speed. Each of these patterns represents a generic solution that we have applied over and over during the creation of our engineering products.

Many of these patterns arose out of the need to create shrink-wrapped software that would be competitive in the resource-constrained Windows environment. I call these patterns *tactical* because they operate at the class or method level to solve implementation, not strategic or architectural, problems.

Here, we consider a family of *instantiation patterns*—patterns that aid in creating or initializing domain objects.

INTERIOR DECORATOR (MIX-IN STATE)

Problem: How do you share a set of useful behaviors that may be selectively needed throughout a family of classes in a broad hierarchy without using multiple inheritance?

Motivation: You are developing an application where many of the objects need common behavior for maintaining a label. However, not every kind of object needs this label behavior. You implement the methods for maintaining labels in the top-level class, referencing the label attribute via an accessor method.

```
Object subclass: #AbstractDomainModel
  instanceVariableNames:"
  classVariableNames:"
  poolDictionaries:"
```

```
label
  "Subclasses must provide access to label state to
  enable label behavior."
  ^self implementedBySubclass
```

```
reportOn: aStream
```

```
"Implement a simple reporting mechanism. Requires
that label behavior be enabled by subclasses."
```

```
aStream
```

```
  nextPutAll: self label;
```

```
  nextPutAll: self results
```

You then add label instance variables to only those classes that need labels.

```
AbstractDomainModel subclass: #ConcreteDomainModel
  instanceVariableNames: 'label'
  classVariableNames:"
  poolDictionaries:"
```

```
label
```

```
"Answer a string, the object's label. This method
provides the state needed to enable inherited label
behavior."
```

```
^label
```

Applicability: This pattern is for use in single-inheritance environments where the only other implementation choice would be a proliferation of redundant subclasses or methods. It requires that the classes needing to share behavior have a common parent class or that you introduce one. Use this pattern when many classes don't need all the possible behaviors and some need none. Each behavior requires some state to support its implementation. Interior Decorator lets you avoid the size penalty of adding unused state throughout the entire hierarchy.

Solution: Create a fat interface with optional state: Place all the methods to support the required behavior high in the hierarchy in abstract classes. Allocate the state that supports the behavior as needed low in the hierarchy in concrete classes.

Implementation: This pattern requires the use of accessor methods to encapsulate the references to instance variables that are only allocated in concrete classes. Typically, you enable a desired behavior by adding instance variables to the class definition and implementing these accessors.

Consequences: This pattern trades off small amounts of state or instance variable redundancy in the leaves of an inheritance hierarchy to share behavior and avoid code redundancy. Note that this pattern works against type

safety: It is possible to send a message to an object requiring a behavior that is present in the class but has not been enabled with the requisite state, resulting in a walkback in a low-level abstract accessor.

Related Patterns: This pattern is like Decorator¹ in that it appears to add selective, small behaviors to a class. However, the behaviors are already present in the parent class and it is the addition of storage for related state that enables them. Interior Decorator does not require replicating the decorated object's public interface in a separate decorator class and does not pay the cost of an extra level of delegation through the decorator.

EPITOME (ATTRIBUTE FACTORY)

Problem: How do you consolidate and share the default values for an object's attributes?

Motivation: You are designing a domain model whose behavior requires certain critical initial values that the user may edit or optionally return to factory defaults. You do not wish to duplicate these initial values throughout the code so you embed them in a single method.

Applicability: Use this pattern when the default values for a domain object's attributes may be referenced in several places: for example, in accessors for lazy initialization, in an initialize method, or in a method that resets to factory defaults. We have found this pattern is most useful for high-level global or project options. In certain low-level objects that must be instantiated and accessed quickly, this pattern may be a performance bottleneck.

Solution: Create a class method that answers a dictionary with one association per attribute, where the key is the symbol for the attribute's accessor and the value is the object that should be the default value of the attribute. Have all references to the attribute's default retrieve the value from this default attribute map.

Implementation: This should be a private method.

```
attributeDefaultMap
  "Answer a dictionary, the default attribute values for
  instances of this class."
  ^IdentityDictionary new
  at: #errorMessage put: String new;
  at: #flags put: 0;
  at: #label put: 'Element';
  yourself
```

Note: If subclasses add many attributes, allocating a larger initial dictionary can offer a substantial performance optimization.

Consequences: Building the Epitome map dynamically and accessing it attribute by attribute can be very expensive. The map could be built when code is loaded and cached in a class variable, otherwise it should be cached

in a temporary variable when used by clients. A benefit of using this pattern, instead of embedding constant default values inside lazy-initializing accessors, is that the initial state of the object is available for review in one method rather than being spread out over many methods. Note that this pattern does not prevent the use of lazy-initialization, it simply moves the values elsewhere.

Related Patterns: An alternative to this pattern is to keep a constant-valued Prototype object available to the class at runtime and use it to seed the initial values of instances. Epitome is essentially a way to build and answer the Memento for that Prototype in code. The benefit of using Epitome is that you do not need to implement mechanisms to manage a Prototype object; the drawback of the pattern is slower performance.

ACTUATOR (INITIALIZING SETTER)

Problem: How do you convert a constant attribute of an object to one that can vary during the lifetime of the object?

Motivation: You have defined a domain model class that is initialized with a diameter. During initialization, the model must size other parameters based on the initialized diameter, like this:

```
initialize
  "Initialize instances to default values."
  diameter := 10.
  self sizeFittings.
```

Later, you decide to modify the domain model so that clients can change its diameter at runtime. To accomplish this you provide a setting accessor method and move the dependent sizing code from the domain model's initialization method to the new accessor:

```
diameter: aFloat
  "Set the diameter of this object to the passed floating
  point value and execute dependent actions."
  diameter := aFloat.
  self sizeFittings.
```

You designate the new accessor for public use by clients when they wish to change the diameter at runtime as well as modifying the domain model's own initialization method to use it:

```
initialize
  "Initialize instances to default values."
  self diameter: 10.
```

Applicability: This pattern is applicable when an object has been designed with some attribute or collaborator that is set at initialization and originally does not change during the life of the object. Special initialization code must run when the identity of that attribute or collaborator is known. Use the pattern when you wish to change the original design to allow clients to dynamically configure the attribute or collaborator during the lifetime of

the object. Actuator is also a candidate for use whenever special initialization actions must be taken once the identities of an object's attributes or collaborators are known.

Solution: Create a setting accessor method for the attribute. Move dependent initialization code into the accessor immediately after the value is set. Ensure that the object itself, when created, uses this accessor for initializing the attribute and that clients use it for changing the attribute's value during the lifetime of the object.


Implementation: Move code from initialization and other methods into a new accessor method. (If the object was initially designed for the given attribute to be constant, some research may be required to find all the initialization code that is dependent on the attribute.) Note that in some cases (e.g., when event handlers have been established on a collaborator), it may also be necessary to write code in the accessor to perform finalization actions before the collaborator can be replaced.

Consequences: Application of this pattern may be beneficial even when attributes aren't expected to change at runtime because it associates dependent initialization logic more closely with the attribute it applies to. Actuator can reduce the size of complex initialize methods by moving their logic into separate accessors.

Related Patterns: Application of this pattern is similar to Template Method in that it turns an initialize method with much attribute-specific logic into a skeleton that delegates to a series of lower-level accessor methods. However, unlike Template Method, those lower-level methods are concrete and not usually intended for overriding.

Actuator is also related to Observer in that dependent code runs in response to some change in state. However, Observer is intended for loose coupling between two or more objects at runtime, whereas Actuator is for setting up at development time, quick responses to changes within a single object.

COMING UP

The next article of my three-part series considers two families of patterns: validation patterns for checking and protecting domain objects and informational patterns for managing status and validation messages. The third and final article will review a family of optimization patterns. 

Reference

1. Gamma, E. et al. *DESIGN PATTERNS*, Addison-Wesley, Reading, MA, 1994.

Darrow Kirkpatrick is VP of Research and Development at Haestad Methods, Inc., which specializes in numerical modeling for hydrology/hydraulics, and has pioneered using Smalltalk for shrink-wrapped Windows applications. Darrow enjoys hunting for patterns while leading a team of talented software engineers who have become experts at coaxing Smalltalk to perform in the real world. He can be contacted at 203.755.1666 (voice) or by email at 75166.525@compuserve.com.

THE BEST OF COMPI.LANG.SMALLTALK

continued from page 20

you'll be happier in the long run if you can just hold off a little longer.

There are several reasons for this. First, time spent on optimization isn't being spent on those "meaningless" chores that are often more important to the success of the project. If testing and documentation are inadequate, most people won't notice or care how fast a particular list box updates. They'll have given up on the program before they ever got to that window.

That's not the worst of it. Premature optimization is usually in direct violation of the principle of postponing decisions. Optimization often involves thoughts like "if we restrict those to be integers in the range from 3 to 87, then we can make this a ByteArray and replace these dictionaries lookups with array accesses". The problem is that we've probably made our code less clear and we've greatly reduced its flexibility. It may have felt really good at the time but the other people involved in the project may not be entirely satisfied.

Of course this rule doesn't apply to all optimizations. Most programs will need some optimization sometime and this is particularly true in Smalltalk. As a very high-level language, Smalltalk makes it very easy to write very inefficient programs very quickly. A little bit of well-placed optimization can make the code enormously faster without harming the program.

There's also a large class of optimizations that I call "stupidity removal" that can be profitably done at just about any time. These include things like using the right kind of collection for the job and avoiding duplicated work. Their most important characteristic is that they should also result in improvements to the clarity and elegance of the code. Using better algorithms (as long as their details don't show through the layers of abstraction) can also fall into this category.

OTHER RULES TO LIVE BY

There are many other rules of life that can be extended to the OO design and programming domains. Here are a few more examples. Feel free to make up more and send them to me. Make posters out of them and put them up on your office wall. It'll make a nice counterpoint to those insipid posters about "Teamwork" and "Quality" that seem to be everywhere these days.

- Try not to care—Beginning Smalltalk programmers often have trouble because they think they need to understand all the details of how something works before they can use it. This means it takes quite a while before they can master Transcript show: 'Hello World'. One of the great leaps in OO is to be able to answer the question "How does this work?" with "I don't care".
- Just do it!—An excellent slogan for projects that are suffering from analysis paralysis, the inability to do anything but generate reports and diagrams for what they're eventually going to do.

continued on page 32

Sequential key allocation strategies in Smalltalk

Dayle Woolston & Chris Kesler

DATABASE APPLICATIONS COMMONLY require unique, sequential keys for inserting new records into tables. For example, a call tracking system may allocate a case number or a billing system may allocate a *customer id*. In some applications, the sequential key may be the preferred method of retrieval (as in locating cases in a call tracking system).

We present several client strategies for key allocation. One size does not fit all. System administrative policy, application features, and overall performance expectations must be considered carefully in selecting an algorithm for generating sequential keys. Accordingly, we examine the following factors:

1. Direct access versus stored procedure access
2. Single versus multiple key allocation
3. Error handling
4. Explicit locking versus browse mode concurrency management strategies

The code samples discussed use VisualWorks 2.0 Smalltalk from ParcPlace Systems and Sybase System 10 from Sybase. The client/server dialog is implemented using the basic mechanisms of the VisualWorks 2.0 Database Connect driver for Sybase.

DIRECT VERSUS STORED PROCEDURE ACCESS

By *direct access* we mean the application issues SQL to generate the next key. In this case, the application must be intimately knowledgeable of various schema management issues: Are the next key seed values all contained in the same table within different records, or does each data table have a corresponding key table? Also, how does the client arbitrate multi-application access of the key table? Should the table be locked during sequential key allocation or should the application select for browse?

Sequential key allocation is a good candidate for stored procedure implementation. The function is simple, highly dependent on the schema, and a focal point of client activity. Should one application implement key allocation improperly, it could wreak havoc on the database.

SINGLE VERSUS MULTIPLE KEY ALLOCATION

Conceptually, *single key allocation* is the simplest strategy. Each time a client requests a new key, it receives the next

ordinal number. This ensures that selects ordered on the key parallel selects ordered on the insertion timestamp for each record. This may or may not be an application requirement, and assumes the key is allocated only after the record has been validated. If the client should abandon the insertion and terminate, the key would be lost. Conversely, this strategy may be affected by when it is necessary in the end-user dialog to make the key available. If the user must know the key prior to server-based validation, it is not possible to guarantee that the key will parallel the insertion timestamp.

There are many performance factors affecting key allocation implementation. Requesting keys is relatively expensive because the client must communicate with the server over the network. Also, because hundreds of clients may compete for the next key, there is a potential system bottleneck. A solution is to have the server provide a range of keys each time the client makes a request. How many keys is enough? The answer depends on the application. In fact, objects managed by an application most likely require different ranges. For example, in a call tracking system, case numbers may be required to be sequential in time, whereas new customer numbers may have no such requirement. Models should provide single and multiple key allocation strategies. We now examine three implementations of key allocation.

SIMPLE UPDATE AND SELECT KEY ALLOCATION

In this first example, the database contains a table named KEYS. There is a record in KEYS for each data table requiring a sequential key. The method is called with the table name as its parameter and returns an integer value.

```
nextKeyFor: aTableName
| ans |
self session
    begin;
    prepare: 'UPDATE KEYS SET serial = serial + 1',
            ' WHERE tableName = "', aTableName, '"';
    execute;
    prepare: 'SELECT serial FROM KEYS where
            tableName = "', aTableName, '"';
    execute.
```

```

ans := self session answer atEnd; next.
self session commit.
^ans first first.

```

The method works because the first statement invokes a transaction that must place a write lock on the KEYS table. This first statement updates the serial value of the record whose tableName value contains the string aTableName. The next statement selects the serial value and returns the integer to the caller. The commit releases any locks; other clients may proceed to complete the same dialog with the server.

This code makes several assumptions. First, it does not explicitly check for any rollback condition. It may be that the client does not have rights to update the KEYS table. This method should encapsulate the execute statement with a handler that enforces a rollback on such conditions. The following code creates an example signal handler. In production code you will want to preallocate the signal handler.

```

nextKeyFor: aTableName
|ans noUpdateSignal|
noUpdateSignal := Signal new notifierString:
    'Unable to get the next key from ',
    aTableName.
noUpdateSignal
handle:
[:ex |
    self adminConnection rollback.
    Dialog warn: ex errorString. ex return]
do:
[self session
begin;
prepare: 'UPDATE KEYS SET serial = serial + 1',
    ' WHERE tableName = "', aTableName, '"';
execute;
prepare: 'SELECT serial FROM KEYS where
    tableName = "', aTableName, '"';
execute.
ans := self session answer atEnd; next.
self session commit].
^ans first first.

```

Second, there may be no record in the KEYS table containing aTableName as the value of its tableName column. (Perhaps some coding error misspelled the table name string.) In this case, the code would execute without error but always return the most recent key in use. The end result would be an insertion error if there were a unique index on the key column of the data table for which the key is intended. This condition can be detected by checking the rowcount attribute for the session after each statement gets executed. It should be 1 in each case.

The final assumption is that the key must be pre-incremented. This is the type of schema assumption that is often documented too casually, causing problems down the road. If this method services all applications accessing the database, then this assumption is probably adequately handled; however, this is very unlikely. The

more likely case is that the table will be accessed by heterogeneous clients: 4GLs, Smalltalks, C, C++, etc. This preincrement policy makes this function a good candidate for stored procedure encapsulation rather than direct implementation.

STORED PROCEDURE KEY ALLOCATION

The next example suggests how the nextKey method may contract with a stored procedure called nextKey.

```

nextKeyFor: aTableName
^useStoredProcedures
ifTrue:
[self session
prepare: 'nextKey ', aTableName, ', ',
    aNumber asString;
execute.
(self session answer atEnd; next) first first]
ifFalse:
[self embeddedSQLNextKeyFor: aTableName].

```

In this example, the database framework has some control over whether it uses stored procedures. If so, the first clause gets executed, otherwise execution gets redirected to another method (a new name for the previous example).

CONCURRENCY ISSUES

One of the vital issues we've avoided so far is that of *concurrency control*. Two separate clients cannot execute the preceding examples at the same time. The two options are *locking* and *browse* mode. Executing locks in this type of method is a pessimistic form of concurrency control. It requires signal handlers to detect the lock condition and possibly repeat attempts until the competing client clears the lock.

In our remaining example, we prefer an optimistic concurrency control strategy using what is known as browse mode. In browse mode, the client issues no lock requests. Instead, the row contains a column of type timestamp. The database service updates the timestamp value each time it performs a select against the row. In requesting the serial value, the client also requests the timestamp value. It then uses the timestamp value as a where clause restriction when updating the row to the incremented serial value. Only the client with the most recent timestamp succeeds in updating the serial value. This success indicates to the method that it may return a valid key to its caller. All other clients executing the same method execute their signal handler and make another attempt at fetching a new key.

Browse mode requires that the select statement end with the words "FOR BROWSE." The target table must have a unique index and, as noted, a timestamp column.

MULTIPLE KEY ALLOCATION IN BROWSE MODE

Our final example demonstrates a rich set of services. The method returns a collection of one or more sequential integer keys for the table aTableName, enabling the client to (possibly) cache multiple key values. We also use

browse mode to implement an optimistic concurrency management strategy.

The method is composed in four sections. The first section sets up the method by assigning a “1” to the attempts counter, allocating a stream to compose queries, and creating a signal instance to manage any exception conditions. The second section defines the exception handling clause. When an exception occurs, the connection must rollback the transaction and increment the attempts counter. If more than 10 attempts occur, notify the user with a dialogue box and error out. Otherwise, just try again. The third section composes and executes two SQL statements. The first statement retrieves the key and timestamp values; the second statement attempts to update the key value. This section checks the session rowcount value to verify that the update occurred. This value should be 1. The fourth and final section creates an ordered collection and assigns to it a sequence of anInteger integers, beginning with the first available key value.

```
nextKeyFor: aTableName incrementBy: anInteger
| currentData timestamp noUpdateSignal attempts oc
aStream |
attempts := 1.
aStream := (String new: 75) writeStream.
noUpdateSignal := Signal new notifierString:
    'Unable to get the next key from ' ,
    aTableName.
noUpdateSignal
handle:
[:ex |
self adminConnection rollback.
attempts := attempts + 1.
attempts > 10 ifTrue: [Dialog warn: ex
    errorString. ex return].
ex restart]
do:
[self adminConnection begin.
self adminSession
    prepare:
        'select key, timestamp from ' ,
        aTableName , ' FOR BROWSE';
    execute.
currentData := self adminSession answer atEnd;
    next.
timestamp := currentData last.
aStream
    nextPutAll: 'update ';
    nextPutAll: aTableName;
    nextPutAll: 'set key = Key+';
    print: anInteger;
    nextPutAll: ' where timestamp = ';
    sqlPrint: timestamp.
self adminSession
    prepare: aStream contents;
    execute.
    answer.
self adminSession rowCount < 1 ifTrue:
```

```
[noUpdateSignal raise].
```

```
self adminConnection commit].
```

```
oc := OrderedCollection new: anInteger.
```

```
lastUsedId + 1 to: lastUsedId + anInteger do: [:i | oc
add: i].
```

```
^oc.
```

Each of these examples is acceptable depending on assumptions that must be supported by the client. This last example, however, illustrates several important optimizations. First, the algorithm for fetching the next key and updating its row on the server is encapsulated by a signal handler. Second, the method checks the session rowcount value to ensure the update actually occurred. If it did not, the signal is raised and the operation attempted up to 10 times. Third, the SQL is constructed using a write stream; a faster strategy than successive string concatenation with commas.

Despite the optimizations, the code is written for clarity over performance. A production version would create the signal at initialization time and keep it in a class-side dictionary. It doesn't make sense to create a new signal during each request for a new key. Also, it is important to allocate a string for the write stream that is very nearly the size of the largest SQL statement. Performance profiling demonstrates that write stream creation time is dominated by the size of the string allocated for the stream. Don't allocate a 500-byte string and think the stream operations are saving you any time over string operations with the comma operator. The best solution is to create the string only once, creating the write stream on that same string over and over with each call to the method. Finally, the ordered collections provide convenient collection services. However, they are considerably slower than arrays. The final section should probably be implemented with an array.

It is not immediately obvious, but this strategy also allocates a separate connection/session for servicing next key requests. It may be that this method executes in the context of saving many objects to the database bounded by a single begin/commit pair. As each object gets saved, it must request the next available key, which (as illustrated) requires its own transaction control. This alternate administrative connection executes key retrievals within a separate transaction. (It is customary for browse mode strategies to employ two connections.)

CACHING KEYS

The remaining code illustrates how a client may cache multiple (sequential) keys. This strategy is appropriate in many cases; it can greatly reduce the number of database requests, thus enhancing overall client performance and reducing network traffic. Without such a strategy, each insert operation generates two database transactions, first to fetch a key and second to perform the insert.

```
nextKey
| key |
(keyCache isNil or:
[keyCache isEmpty])
```

continued on page 32

GETTING REAL *continued from page 19*

```
name first clusterInBucket: empCluster.  
name middle clusterInBucket: empCluster.  
name last clusterInBucket: empCluster.  
  
" cluster the address and its components "  
address := anEmp address.  
address clusterInBucket: addressCluster.  
address street clusterInBucket: addressCluster.  
address city clusterInBucket: addressCluster.  
address state clusterInBucket: addressCluster.  
address zip clusterInBucket: addressCluster. ].
```

This column has described how to determine if clustering objects might help application performance and how to cluster objects using ClusterBuckets. My next column will discuss how to measure overall system performance and steps for tuning multi-user Smalltalk for higher transaction throughput. 📖

THE BEST OF COMPIANG.SMALTALK

continued from page 23

- Avoid commitment—This is another way of expressing the principle of postponing decisions but one that might strike a chord with younger or unmarried programmers.
- It's not a good example if it doesn't work—This one comes from David Buck (dbuck@magnacom.com), who's fed up with looking at example and test methods that haven't been properly maintained as the code evolved. I can't think of a way to apply this to life but it's good advice anyway.
- Steal everything you can from your parents—A principle for those trying to make effective use of inheritance or moving into their first apartment.
- Cover your a**—Like in a bureaucracy, the most important thing is to make sure that it isn't your fault. Make sure your code won't have a problem even if things are going very wrong elsewhere. 📖

SEQUENTIAL KEY ALLOCATION

continued from page 26

```
ifTrue: [ keyCache := self nextKeys: self  
         keyCacheSize ].  
  
key := keyCache first.  
keyCache removeFirst.  
^key.
```

If you choose to make the array optimization in the nextKeys: method, this method must be changed to insert nil values into the array as each key gets returned rather than using the removeFirst selector. 📖

Dayle Woolston and Chris Kesler have been working with Smalltalk for 4 years building client/server database applications. They can be reached at dayle_woolston@novell.com and chris_kesler@novell.com.



Jan Steinman



Barbara Yates

The Demo Trap

“C’MON, GUYS, WE don’t have time for testing, we’ve gotta get the demo ready for the department heads meeting!” announced Aaron, the project leader, as he stared at the ceiling in exasperation. Things had been going well—too well, it seemed. MegaCorp had been having some difficult times with development schedules and after many studies, meetings, and much political maneuvering, management went along with Aaron Blake’s plan to transfer their traditional development environment into Smalltalk with a pilot project.

Aaron had been in this game awhile; he knew this project was a career maker or breaker, so he had planned the transition in careful detail. His team had gotten the training they needed, he had budgeted for the best tools available, and in a real coup for MegaCorp, he had even convinced them to bring in some experienced mentors, so that his team wouldn’t repeat someone else’s mistakes.

Most important, MegaCorp MIS Director, Andrea Saunders, had personally approved Aaron’s proposed development process, which was unlike anything anyone had ever seen within MegaCorp. MegaCorp needed rapid turnaround on various software projects and their traditional waterfall process had been running an average of 250% off original schedule. Their users required six-week changes; IS was quoting six-month changes but actually delivering in a year and a half!

It had been a hard sell. Andrea knew there was a problem, but she wasn’t ready to simply swap a new problem for an old one. “Whenever one of my guys comes crying about all the hoops they’re jumping through, I ask them, ‘ya got something better?’”

Luckily, Aaron was in his usual state of preparedness and had come to the meeting with an impressive presentation citing Barry Boehm and other development process scholars. He showed Andrea and her department heads how simultaneous design, implementation, and

testing could result in an evolving, incremental product that could deliver quick changes to users without sacrificing quality. He had no idea that all his careful planning could result in disaster!

EARLY GAINS

“Trigger” Larsen was Aaron’s best developer and the fastest squirt-gunner in the company. Trigger had gone off for a month of “Smalltalk immersion therapy” in an apprenticeship program, and had returned with stars in his eyes. “Hey Aaron, look what I got working today!” he’d often exclaim, which was a bit tiring, but it was the kind of nuisance Aaron could easily live with—much preferred over the usual complaining about short schedules and missed deadlines.

Trigger quickly assembled a GUI that was years ahead of anything MegaCorp had ever put together. True, the database wasn’t connected yet, and the legacy systems were not interfaced, but hey—it looked sexy and actually seemed to be doing something, unlike the “paint and draw” prototypes that other departments had been putting together for years.

Enthusiasm has its drawbacks and when combined with a little boastful competitiveness, it can have bad side effects. Trigger played racquetball with Denny Hicks, a developer on a “traditional development” team, and could contain his enthusiasm no longer. “Hey, you should see the neat stuff *we’ve* been doing!” Trigger shouted over the sound of ricocheting balls and pounding sneakers. It seemed an innocent enough boast at the time.

PROCEDURAL DISSONANCE STIRRING

It was kind of nice being ignored. Aaron knew this couldn’t last forever, but neither did he suspect it was the calm before the storm. The pace of a traditional waterfall project meant that the first third or so of the schedule was dominated by documents—but no one wanted to see the documents, they just wanted assurance that the documents were being produced.

Aaron was cheating a bit here on the cyclic development process. He followed the letter of the MegaCorp Software Procedures and Standards Manual by naming

Jan Steinman and Barbara Yates are cofounders of Bytesmiths, a technical services company that has been helping corporations adopt Smalltalk since 1987. Between them, they have more than 20 years Smalltalk experience. They can be reached at Barbara@Bytesmiths.com or Jan@Bytesmiths.com, or on the web at <http://www.bytesmiths.com>.

and listing all the required documents, and checking them off as produced, reviewed, and delivered at the appropriate intervals.

The “cheating” part was that they were “incomplete” by MegaCorp standards—he kept them in their Smalltalk development environment, so they could be updated readily, instead of “checking them in” to the mainframe, which generally meant no one ever changed them again, because rigorous procedures were involved with check-in and check-out.

They were also much smaller than usual because instead of an extensive boilerplate for every conceivable situation (half of which were marked “N/A”), Aaron insisted on plain English descriptions of the behaviors and interactions of the objects in the system.

A greater heresy that wouldn’t have passed muster if he hadn’t bribed the bored QA guys (by promising to train them in Smalltalk and actually let them write some test code) was the fact that none of the documents his group produced said *anything* about the internal structure of their objects! He knew he could get shot down for that, but he also suspected he’d need that flexibility later during performance tuning. “Isn’t it enough for now to say that an Employee can answer its Department without stating that it *stores* its Department somewhere?” Aaron mused to himself.

The net result of this was that after several months, MegaCorp treated Aaron as though he were “waterfalling” through analysis into design—it ignored him—while in reality, his team had Pretty Neat Stuff running. No one at MegaCorp had ever seen Pretty Neat Stuff running before at least 110% of the initial schedule had been spent!

“Hey Aaron!” shouted Jake Sather across the lunch room, “Denny tells me you’ve got a GUI going—can I get a look sometime soon?” Jake was Aaron’s peer, managing the “traditional development” group Denny Hicks worked in. “Sure, c’mon up this afternoon!” replied Aaron. (He had been catching some of Trigger’s enthusiasm as of late.)

WHAT IS OUR PRODUCT?

Jake’s demo went well—too well, it seemed. It would have been better if the demo had gone worse, say, if it had crashed a few times. It would have been better if Aaron had stalled Jake to prepare, because Aaron’s team had been practicing continuous integration and continuous testing, and consequently this was the best spur-of-the-moment demo Jake had ever seen at MegaCorp. Aaron looked at it and saw a half-finished prototype; Jake looked at it and saw a product.

The Friday staff meeting was not a pretty sight for Aaron. “You guys should see what Aaron’s been hiding from us!” Jake started out, “We all should be looking at this stuff!” They scheduled several demos that week. Marketing should get a look; Sales was on a boondoggle, so they would need their own demo; of course the tech writers would need a working copy; and users—what about the users? “We’ll have to find some; Aaron, can you look into that?”

None of Aaron’s careful planning accounted for the

coming weeks. Oh, he had planned for demos all right, but he didn’t anticipate the magnitude of interest that the “early GUI” generated.

But the interest in Aaron’s stuff was not passive. Everyone thinks they’re a “GUI guru,” when really they are more likely seduced by “neat stuff” rather than what actually is useful to an end user. Marketing immediately wanted “more color and icons,” while various VPs who drifted in and out of Aaron’s office put in their own personal GUI order. Meanwhile, Sales wanted to immediately ship everything they saw. Aaron was averaging at least two demos per week, each one ending with some “recommendation” of some kind. His schedule was slipping as his developers thrashed away implementing conflicting, unplanned changes.

Aaron fantasized about the want ad he would soon be placing: “Product Demo Manager for hire. Expert at jumping through upper management hoops with constantly changing but meaningless user interface stuff. If your product is demos, I’m your man!” He thought hard about what he was trying to accomplish, mentally crumpled up the half-composed ad, and picked up the phone. “Andrea, can we have a talk?” he asked.

RETURN OF SANITY

After hearing him out, Andrea Saunders agreed that Aaron’s project needed a shield. Demos would be limited to the period immediately following “cycle end,” which averaged every six weeks. “Must do” demos in between the scheduled ones would only be on the software that had recently been formally presented—no more pulling people off “real work” to prepare a special demo.

Trigger had completed the architecture work and much of the design and had given the implementation a good start. His outgoing, enthusiastic, and somewhat boastful personality earned him the new unofficial title of “appeasement engineer.”

“Half your job is to see that no one else on the team is impacted by demo activity,” Aaron told him, “and the other half is to do everything else you were doing with all your time before!” he joked. He knew that Trigger was up to the challenge, and could carry on limited peer review and coding while carefully tracking all the little tasks that producing a good demo requires, and still have time for manic squirt-gun fights.

The Demo Trap had slipped the schedule, and they weren’t going to ship all the functions they had originally planned, but their “continuous development” regime was paying off regardless. They had taken on “productization” early, and all they had to do between cycles was add functionality. Sales wanted what they had *now*, and Marketing was actually considering a “MegaSoftLite” offering to “selected beta partners.”

“If this keeps up,” Aaron mused, “it could be the first time in history that MegaCorp has shipped a product—albeit not the one they set out to build—ahead of schedule!” He sighed a contented sigh and went back to planning his next Smalltalk project. ☞



Jay Almarode

Tuning multi-user Smalltalk

AN IMPORTANT ACTIVITY before delivering any application is tuning the code to meet performance requirements. In single-user Smalltalks on the client machine, this activity typically involves using profiling tools to identify the methods where most time is spent. Once these methods are identified, several options are available, such as implementing the methods as primitives in C code, caching calculated values that are used repeatedly, or perhaps most important, producing a better design. This tuning activity might also involve analyzing the memory usage of the application, reducing the memory footprint of the application while it is running, minimizing the number of temporary objects that are created and then quickly garbage collected, and exercising more explicit control over garbage collection (especially in real-time systems).

These same tuning activities are applicable to multi-user, server Smalltalk as well. In addition, because server Smalltalk must accommodate concurrent transactions by many hundreds of users, and must handle many millions of objects being created and retrieved, there are additional ways in which applications can be tuned. In this column I discuss some of the techniques to tune multi-user, server Smalltalk applications.

A key component in tuning a large-scale, multi-user Smalltalk application is understanding and controlling the placement of objects on disk. Because the number and size of objects may prevent all that are being used in an application from being present in RAM at once, the proximity of objects may impact application performance. Obviously, the fewer disk pages to be accessed during the normal course of application execution the better performance. To tune the placement of objects on disk, server Smalltalk must allow developers to cluster objects that are frequently accessed together. In GemStone Smalltalk, objects are placed on disk based on their access patterns by default. More specifically, objects that are created or modified within the same transaction tend to be placed close together. In many cases, this default placement is sufficient.

However, GemStone Smalltalk does provide additional protocol to allow developers to discover where objects are

placed and to move them closer together for more efficient access.

The first step in tuning an application's performance for accessing objects is to understand the reading and writing characteristics of the application while it is running. In GemStone Smalltalk, you can send the messages "pageReads" or "pageWrites" to class System to get the cumulative number of pages that were read or written since the session began (i.e., since you logged into the server). Typically, it is useful to measure the number of pages read immediately before and immediately after an extensive calculation or query to determine if clustering objects together might be of benefit. For example, the following code returns the number of pages that were read to execute the given query.

```
| initialNumberOfReads |
initialNumberOfReads := System pageReads.
SetOfPersons select: [ :person |
    " find each person younger than their spouse "
    person isMarried and: [ person spouse age > person age ].
    ^ System pageReads - initialNumberOfReads
```

Pages are written to disk for two reasons: first, when internal buffers become full and must make room for new objects to be created; second, when the transaction is committed. Measuring the number of pages written at various times during the life of a transaction can help determine if buffer sizes need to be increased, whereas measuring the number of pages written just before and after a transaction is committed may help determine if more explicit control over clustering may help.

Clustering related objects together solves a specific problem: poor performance because of too much disk activity. One way to check how objects are clustered is to determine which pages the objects are stored on. You can send the message "page" to any object to get an integer identifying the disk page on which the receiver resides. This integer is a logical identifier of the page, not a pointer to a storage location.

Objects are stored persistently in structures called *extents*. An extent is a disk file or raw partition on disk. The repository of all objects can be maintained in multiple extents, possibly distributed among several disk drives on several machines. In GemStone Smalltalk, there is a single object, named SystemRepository, that is an instance of

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@gemstone.com.

class Repository. In addition to defining protocol to perform online backups and restores, to dynamically add new extents, and to create replicates of extents for purposes of fault tolerance, class Repository also has methods to provide information about the extent in which a page is located and the file name for a given extent. The next example shows how one can determine the file name where an object is actually stored on disk.

```
| extendId |
"MyObject is the object whose location we are interested in."
extendId := SystemRepository extentForPage: MyObject
page.
^ SystemRepository fileName: at: extendId.
```

By analyzing the reading and writing behavior of your application for excessive disk activity and determining the number and location of pages where objects reside, performance may be improved by explicitly controlling how objects are clustered together. Conceptually, you can think of objects as being written to disk on a stream of disk pages. When a page is filled, another page is chosen and objects are written to the new page. The stream of pages used for writing is called a 'bucket'. GemStone Smalltalk provides the class ClusterBucket to give programmers control over which stream of pages objects are written. Every object is associated with an instance of ClusterBucket and all objects assigned to the same ClusterBucket will be clustered together. When objects with the same ClusterBucket are written to disk, they are written to contiguous locations on the same page, if they will fit, or contiguous locations on several pages if not.

A ClusterBucket can be associated with a specific extent. Each ClusterBucket has an instance variable extentId that specifies which file the stream of pages will be written. You can find out what extents are available by executing the expression SystemRepository fileSizeReport. This returns a string that describes the extent identifier, file name, file size, and space available for each available extent. An example of how to set the extent for an existing ClusterBucket is the expression aClusterBucket extendId: 3.

You can create a new ClusterBucket by executing the expression ClusterBucket newForExtent: 4. Initially, there are seven existing instances of ClusterBucket maintained in a global array named AllClusterBuckets.

Some of these are available for application developers, whereas others are used to cluster system objects, such as kernel methods or source code strings. When new

instances of ClusterBucket are created, they are added to this global array and a ClusterBucket's position in this array is known as its cluster Id.

This provides a way to reference any ClusterBucket that ex-

ists through its clusterId, for example, by performing the expression ClusterBucket bucketWithId: 7.

To specify the ClusterBucket for a particular object, you can send the message "clusterInBucket: aClusterBucket". This will not immediately write the object to disk but indicates that when it is next written, the stream of pages in which it is written will be determined by the given ClusterBucket.

If you want to write the object to disk immediately, you can send the message "moveToDiskInBucket: aClusterBucket". Sending the "clusterBucket" message to an object will return the ClusterBucket to which the receiver is currently assigned. GemStone Smalltalk provides some convenience methods to help cluster objects. You can send the message "cluster" to any object to assign it to the current default ClusterBucket. You can use this message to build specialized clustering behaviors for your application classes. One such method already provided is clusterDepthFirst, which traverses through the named and indexable instance variables of the receiver, sending the "cluster" message to each object. The cluster method returns a boolean indicating if the receiver has already been clustered during the current transaction. This is used to prevent infinite recursion. There are also convenience methods defined in class Behavior to cluster classes and related objects. The clusterBehavior method clusters a class and its method dictionary. The clusterDescription method clusters the objects that describe the structure of a class, such as its instVarNames array, class variables, instance variable constraints, and class history.

To illustrate how to control object clustering, imagine a set of Employee objects based on the simplified schema illustrated in Figure 1.

Suppose most applications that access an instance of Employee also access the name and ssn as well; so we would like to cluster instances of Employee with their corresponding Name and 'ssn' String objects.

The addresses of employees are accessed less frequently and are typically accessed for all employees at once, so we would like to cluster all Address objects together. The following code shows how we can cluster these objects so that employees and their frequently accessed subcomponents are stored contiguously and employee addresses are grouped together separately.

```
| empCluster addressCluster |
" get the bucket previously created for Employees "
empCluster := ClusterBucket bucketWithId: 8.
" get the bucket previously created for Addresses "
addressCluster := ClusterBucket bucketWithId: 9.
```

```
TheSetOfEmployees do: [ :anEmp | | name address |
anEmp clusterInBucket: empCluster.
anEmp ssn clusterInBucket: empCluster.
```

```
" cluster the name and its components "
name := anEmp name.
name clusterInBucket: empCluster.
```

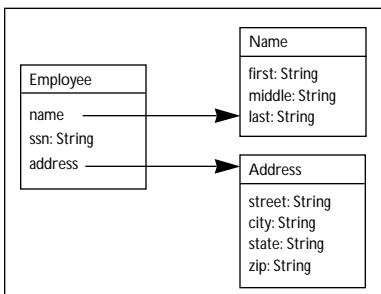


Figure 1. Employee schema.

continued on page 32

GETTING REAL *continued from page 19*

```
name first clusterInBucket: empCluster.  
name middle clusterInBucket: empCluster.  
name last clusterInBucket: empCluster.  
  
" cluster the address and its components "  
address := anEmp address.  
address clusterInBucket: addressCluster.  
address street clusterInBucket: addressCluster.  
address city clusterInBucket: addressCluster.  
address state clusterInBucket: addressCluster.  
address zip clusterInBucket: addressCluster. ].
```

This column has described how to determine if clustering objects might help application performance and how to cluster objects using ClusterBuckets. My next column will discuss how to measure overall system performance and steps for tuning multi-user Smalltalk for higher transaction throughput. 📖

THE BEST OF COMPIANG.SMALTALK

continued from page 23

- Avoid commitment—This is another way of expressing the principle of postponing decisions but one that might strike a chord with younger or unmarried programmers.
- It's not a good example if it doesn't work—This one comes from David Buck (dbuck@magnacom.com), who's fed up with looking at example and test methods that haven't been properly maintained as the code evolved. I can't think of a way to apply this to life but it's good advice anyway.
- Steal everything you can from your parents—A principle for those trying to make effective use of inheritance or moving into their first apartment.
- Cover your a**—Like in a bureaucracy, the most important thing is to make sure that it isn't your fault. Make sure your code won't have a problem even if things are going very wrong elsewhere. 📖

SEQUENTIAL KEY ALLOCATION

continued from page 26

```
ifTrue: [ keyCache := self nextKeys: self  
         keyCacheSize ].  
  
key := keyCache first.  
keyCache removeFirst.  
^key.
```

If you choose to make the array optimization in the nextKeys: method, this method must be changed to insert nil values into the array as each key gets returned rather than using the removeFirst selector. 📖

Dayle Woolston and Chris Kesler have been working with Smalltalk for 4 years building client/server database applications. They can be reached at dayle_woolston@novell.com and chris_kesler@novell.com.



Alan Knight

Principles of OO design, Part 2

LAST MONTH, WE reviewed some important principles of OO design, many of which apply equally well to life. The fundamental principle of OO is: *Never do any work that you can get someone else to do for you.* And there are secondary principles:

- Avoid responsibility
- Postpone decisions

This month we examine a few more principles.

MANAGERS DON'T DO ANY REAL WORK

The subject of “manager” or “control” objects can provoke a lot of debate in OO circles, much as the subject of “managers” does in other work environments. Some argue that the role of manager is inherently bad for software design and that one should avoid employing them. Others argue that, although many of them represent a throwback to outdated ways of thinking, they can be very useful under the right circumstances.

I definitely believe that managers can be useful, but it's important to distinguish between good ones and bad ones. For example, consider a program in which most of my classes are “record objects” (objects whose only behaviours are get and set methods). The real work is done by a control class that manipulates these objects with full access to all their data. At this point I have a procedural program dressed up in an OO disguise. The control object is in the most complete possible violation of the fundamental principle because it's trying to do all the work itself.

On the other hand, consider a window class like the VisualWorks ApplicationModel or the Visual Smalltalk ApplicationCoordinator. These are manager objects that coordinate the interactions between user interface widgets and the domain model. They're very important to good GUI design and it would be much harder to get a clean design without them.

People who are vehemently opposed to any kind of manager object are often stuck in the trap of trying to precisely model the world, taking the OO paradigm much too literally. One of my favourite quotes on this subject (from several years back) is from Jeff Alger, who wrote:

“The real world is the problem; why would you want to just simulate it?”

Alan Knight is cynic-in-residence at The Object People, 885 Meadowlands Dr. E., Ottawa, Ontario, K2C 3N2. He can be reached at 613.225.8812 or by email at knight@acm.org.

How can we tell a good manager object from a bad one? We apply the principle that managers don't do real work. A manager object should manage interactions between other objects and should be trying to do work itself, unless it's legitimate management work.

An example of legitimate management work is an ApplicationModel figuring out which menu items need to be disabled. An example of nonlegitimate work would be doing (nontrivial) calculations of values to be displayed in its fields. Those values should be calculated by the domain objects.

This rule can be tricky to apply in practice. It is always obvious whether something is legitimate management work or not. Always remember that this is just a specific example of the fundamental principle. If the manager can plausibly get someone else to do the work, it should do so.

Another difficulty is that the word “Manager” is sometimes tacked on to the end of a class name even though what it describes is not a manager at all. In a recent comp.object discussion, Robert Cowham (cowhamr@logica.com) described a DiscountPolicyManager object and worried about the desirability of introducing a manager object even though it seemed to make the design cleaner. The description was as follows:

A Discount Policy Manager is going to be passed, say, an Invoice object and will calculate the appropriate discount to be applied to that Invoice (using methods on the Invoice to find out about it) and then use a method on Invoice to add the discount to it.

Reading this description, it's clear that the Discount PolicyManager is really just a policy object as described in the previous section. It isn't a manager at all and should be called DiscountPolicy instead.

PREMATURE OPTIMIZATION LEAVES EVERYONE UNSATISFIED

The most fun you can have as a programmer is optimizing code. There's nothing quite so satisfying as taking some little piece of functionality and making it run 50 times faster than it used to. When you're deep in the middle of meaningless chores like commenting, testing, and documenting, the temptation to let go and optimize is almost irresistible. You know it's got to be done sometime and you feel like you just can't put it off any longer. Sometimes you're right and the time has come to make this piece of code really scream. More often than not, *continued on page 23*

the object. Actuator is also a candidate for use whenever special initialization actions must be taken once the identities of an object's attributes or collaborators are known.

Solution: Create a setting accessor method for the attribute. Move dependent initialization code into the accessor immediately after the value is set. Ensure that the object itself, when created, uses this accessor for initializing the attribute and that clients use it for changing the attribute's value during the lifetime of the object.


Implementation: Move code from initialization and other methods into a new accessor method. (If the object was initially designed for the given attribute to be constant, some research may be required to find all the initialization code that is dependent on the attribute.) Note that in some cases (e.g., when event handlers have been established on a collaborator), it may also be necessary to write code in the accessor to perform finalization actions before the collaborator can be replaced.

Consequences: Application of this pattern may be beneficial even when attributes aren't expected to change at runtime because it associates dependent initialization logic more closely with the attribute it applies to. Actuator can reduce the size of complex initialize methods by moving their logic into separate accessors.

Related Patterns: Application of this pattern is similar to Template Method in that it turns an initialize method with much attribute-specific logic into a skeleton that delegates to a series of lower-level accessor methods. However, unlike Template Method, those lower-level methods are concrete and not usually intended for overriding.

Actuator is also related to Observer in that dependent code runs in response to some change in state. However, Observer is intended for loose coupling between two or more objects at runtime, whereas Actuator is for setting up at development time, quick responses to changes within a single object.

COMING UP

The next article of my three-part series considers two families of patterns: validation patterns for checking and protecting domain objects and informational patterns for managing status and validation messages. The third and final article will review a family of optimization patterns. 

Reference

1. Gamma, E. et al. *DESIGN PATTERNS*, Addison-Wesley, Reading, MA, 1994.

Darrow Kirkpatrick is VP of Research and Development at Haestad Methods, Inc., which specializes in numerical modeling for hydrology/hydraulics, and has pioneered using Smalltalk for shrink-wrapped Windows applications. Darrow enjoys hunting for patterns while leading a team of talented software engineers who have become experts at coaxing Smalltalk to perform in the real world. He can be contacted at 203.755.1666 (voice) or by email at 75166.525@compuserve.com.

THE BEST OF COMPI.LANG.SMALLTALK

continued from page 20

you'll be happier in the long run if you can just hold off a little longer.

There are several reasons for this. First, time spent on optimization isn't being spent on those "meaningless" chores that are often more important to the success of the project. If testing and documentation are inadequate, most people won't notice or care how fast a particular list box updates. They'll have given up on the program before they ever got to that window.

That's not the worst of it. Premature optimization is usually in direct violation of the principle of postponing decisions. Optimization often involves thoughts like "if we restrict those to be integers in the range from 3 to 87, then we can make this a ByteArray and replace these dictionaries lookups with array accesses". The problem is that we've probably made our code less clear and we've greatly reduced its flexibility. It may have felt really good at the time but the other people involved in the project may not be entirely satisfied.

Of course this rule doesn't apply to all optimizations. Most programs will need some optimization sometime and this is particularly true in Smalltalk. As a very high-level language, Smalltalk makes it very easy to write very inefficient programs very quickly. A little bit of well-placed optimization can make the code enormously faster without harming the program.

There's also a large class of optimizations that I call "stupidity removal" that can be profitably done at just about any time. These include things like using the right kind of collection for the job and avoiding duplicated work. Their most important characteristic is that they should also result in improvements to the clarity and elegance of the code. Using better algorithms (as long as their details don't show through the layers of abstraction) can also fall into this category.

OTHER RULES TO LIVE BY

There are many other rules of life that can be extended to the OO design and programming domains. Here are a few more examples. Feel free to make up more and send them to me. Make posters out of them and put them up on your office wall. It'll make a nice counterpoint to those insipid posters about "Teamwork" and "Quality" that seem to be everywhere these days.

- Try not to care—Beginning Smalltalk programmers often have trouble because they think they need to understand all the details of how something works before they can use it. This means it takes quite a while before they can master Transcript show: 'Hello World'. One of the great leaps in OO is to be able to answer the question "How does this work?" with "I don't care".
- Just do it!—An excellent slogan for projects that are suffering from analysis paralysis, the inability to do anything but generate reports and diagrams for what they're eventually going to do.

continued on page 32

GETTING REAL *continued from page 19*

```
name first clusterInBucket: empCluster.  
name middle clusterInBucket: empCluster.  
name last clusterInBucket: empCluster.  
  
" cluster the address and its components "  
address := anEmp address.  
address clusterInBucket: addressCluster.  
address street clusterInBucket: addressCluster.  
address city clusterInBucket: addressCluster.  
address state clusterInBucket: addressCluster.  
address zip clusterInBucket: addressCluster. ].
```

This column has described how to determine if clustering objects might help application performance and how to cluster objects using ClusterBuckets. My next column will discuss how to measure overall system performance and steps for tuning multi-user Smalltalk for higher transaction throughput. 📖

THE BEST OF COMPLANG.SMALTALK

continued from page 23

- Avoid commitment—This is another way of expressing the principle of postponing decisions but one that might strike a chord with younger or unmarried programmers.
- It's not a good example if it doesn't work—This one comes from David Buck (dbuck@magnacom.com), who's fed up with looking at example and test methods that haven't been properly maintained as the code evolved. I can't think of a way to apply this to life but it's good advice anyway.
- Steal everything you can from your parents—A principle for those trying to make effective use of inheritance or moving into their first apartment.
- Cover your a**—Like in a bureaucracy, the most important thing is to make sure that it isn't your fault. Make sure your code won't have a problem even if things are going very wrong elsewhere. 📖

SEQUENTIAL KEY ALLOCATION

continued from page 26

```
ifTrue: [ keyCache := self nextKeys: self  
         keyCacheSize ].  
  
key := keyCache first.  
keyCache removeFirst.  
^key.
```

If you choose to make the array optimization in the nextKeys: method, this method must be changed to insert nil values into the array as each key gets returned rather than using the removeFirst selector. 📖

Dayle Woolston and Chris Kesler have been working with Smalltalk for 4 years building client/server database applications. They can be reached at dayle_woolston@novell.com and chris_kesler@novell.com.

Editors' Corner



John Pugh



Paul White

HERE'S A TECHNICAL QUESTION FOR YOU—do you ever use `become:` in your applications? What twiggled the issue was the fact that, for a long time now, our training materials have included an application that makes use of the `become:` operation. It was written to illustrate the power of Smalltalk and to introduce people to the issue of *object mutation*. But we know that in our own software development, we rarely use it; we're sure that's true of others as well. The question is, why not? And if it's not to be used, then should vendors continue to support it?

For those of you who may not be familiar with `become:`, it is defined as an instance method in the class `Object`. The intention of the message `send "object1 become: object2"` is to mutate `object1` into `object2`. However, there are two different implementations found in the various Smalltalks. Visual Smalltalk's implementation has always had the effect that all references to `object1` are made references to `object2`; `object1` is left with no references and is therefore garbage collected. IBM Smalltalk for Windows and OS/2 provide similar behavior. VisualWorks, on the other hand, implements a `become:` that swaps the references between the two objects—all the original references to `object1` become references to `object2`, and all the original references to `object2` are made references to `object1`; neither object gets garbage collected.

This, of course, leaves you with radically different behavior for the same message, which is one obvious reason for not using `become:`. The classic example of this problem is `someObject become: nil` yields radically different results! Another reason for the lack of use of `become:` is historical. Early implementations of Smalltalk did not always yield the expected results and/or perform well at all. As a result, many gave up on the operation and found another approach.

But the question still lingers—even if the issues listed were resolved, would it be appropriate to make use of `become:`? The answer, of course, is "it depends..."

*Here's a technical question
for you—do you ever use
`become:` in your
applications?*

The real issue is there are two different interpretations of what `become:` really means in an application. A statement such as "with `become:`, you can make an elephant become a mouse" is not appropriate because the two objects are not related to each other in any manner. This really represents replacement, not mutation.

Another interpretation of `become:` is that it provides a mechanism to allow an object to dynamically change its behavior. Consider this statement: "As a boy grows older, he becomes a man." The message `become:` can be used in this context to model the changing behavior of a person. For example, mobility is achieved by a young child through crawling; an older child walks. As a result, when the message `move Around` is sent to a person, the message is interpreted differently at different ages. A possible implementation would be to have a child object

actually mutate into an adult object. It is important that the object decides to do this itself and does not depend on other objects. This avoids code like `self age < 2 ifTrue: [self crawl] ifFalse: [self walk]`. Although the example is trivial, it is illustrative of the type of behavior changes we require from an object.

So, is `become:` necessary? Probably. Applications do exist where it would be beneficial and perhaps even appropriate. But in the end we find it is a neat idea, but for most problems we should find another solution. For example, in the person example cited, it's always possible to introduce a generic `Person` object and have them have roles or characteristics. So, a person may either have child or adult characteristics and its behavior is directly dependent on which characteristics it has. These are nontrivial systems to implement, but they give you the desired behavior.

As this is the Smalltalk Solutions issue, we hope many of you take advantage of the conference. It's the best opportunity for Smalltalk to show off its successes. Please drop by and introduce yourself!

Editors

John Pugh and Paul White
Carleton University & The Object People

SIGS Publications Advisory Board

Tom Atwood, *Object Design*
François Bancelhon, *O₂ Technology*
Grady Booch, *Rational*
George Bosworth, *ParcPlace-Digital*
Jesse Michael Chonoles, *Lockheed Martin ACC*
Stuart Frost, *SELECT Software*
Adele Goldberg, *ParcPlace-Digital*
Thomas Kaffer, *Rogue Wave Software*
R. Jordan Kriendler, *IBM Consulting Group*
Thomas Love, *Consultant*
Bertrand Meyer, *ISE*
Meilir Page-Jones, *Wayland Systems*
Cliff Reeves, *IBM*
Bjarne Stroustrup, *AT&T Bell Labs*
Dave Thomas, *Object Technology International*

The Smalltalk Report

Editorial Board

Jim Anderson, *ParcPlace-Digital*
Adele Goldberg, *ParcPlace-Digital*
Reed Phillips
Mike Taylor, *ParcPlace-Digital*
Dave Thomas, *Object Technology International*

Columnists

Jay Almarode, *GemStone Systems Inc.*
Kent Beck, *First Class Software*
Juanita Ewing, *ParcPlace-Digital*
Bob Hinkle, *Consultant*
Tim Howard, *FH Protocol, Inc.*
Ralph E. Johnson, *University of Illinois*
Alan Knight, *The Object People*
Mark Lorenz, *Hatteras Software, Inc.*
Jan Steinman, *Bytesmiths*
Rebecca Wirfs-Brock, *ParcPlace-Digital*
Barbara Yates, *Bytesmiths*

SIGS Publications Group, Inc.

Richard P. Friedman, Founder, President, and CEO
Hal Avery, Group Publisher
John McCormick, Editorial Director

Editorial/Production

Elizabeth A. Upp, Managing Editor
Elisa Varian, Director of Manufacturing
Andrea Cammarata, Art Director
Seth Booke, Assistant Managing Editor
Dan Olawski, Production Editor
Sue Mycka, Desktop Designer
Margaret Conti, Advertising Production Coordinator

Circulation

Elayne Glick, Circulation Director
Lawrence E. Hoffer, Marketing Manager
Byron Scarlett, Assistant Circulation Manager

Advertising/Marketing

Gary Portie, Advertising Manager, East Coast/Canada/Europe
Elisa Marcus, Advertising Manager, Central US
Michael W. Peck, Advertising Representative
Kristine Viksnins, West Coast Exhibit Sales
Sarah Olszewski, East Coast Exhibit Sales
212.242.7447 (v), 212.242.7574 (f)
Diane Fuller & Associates, Sales Representative, West Coast
408.255.2991 (v), 408.255.2992 (f)
Nancy Beuschel, Promotions Manager for Magazines

Administration

Margherita R. Monck, General Manager
David Chatterpaul, Senior Accounting Manager
Bibi Budhram, Accounts Payable



Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, THE SMALLTALK REPORT, THE X JOURNAL, REPORT ON OBJECT ANALYSIS & DESIGN, JAVA REPORT, OBJECT CURRENTS (ONLINE), OBJECT EXPERT (UK), and OBJECTSPEKTRUM (GERMANY)

Table of Contents

May 1996

Vol 5 No 7

Features

Tactical patterns for the real world: Validation and informational patterns 4

Darrow Kirkpatrick

This series on patterns for working with domain models continues, with a focus on patterns for validating objects and for managing status and validation messages.

Equality versus identity 10

Bobby Woolf

Experienced Smalltalk programmers know the difference between object equality and object identity—but what does it mean in terms of logical programming?

Columns



Deep in the Heart of Smalltalk 14

The active life is the life for me!

Bob Hinkle & Ralph E. Johnson

Active variables allow programmers to monitor all accesses to object state on a per-object basis.



Getting Real 21

Configuring server Smalltalk

Jay Almarode

Configuring and tuning multi-user, server Smalltalk systems is very different from tuning single-user client Smalltalk applications. The tools available for monitoring statistics can be invaluable in this process.



Managing Objects 23

Beware the octopus

Jan Steinman & Barbara Yates

If your organization has system diagrams with a bunch of circles connected by lines to a central circle, it may not be ready for object thinking.

Departments

Editors' Corner 2

CONFERENCE OVERVIEW Smalltalk Solutions '96: 27

Progress and new challenges

Product News 30

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar–Apr, July–Aug, and Nov–Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1996 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Bulk rate U.S. postage paid Lancaster, PA, permit 161. Canada Post International Publications Mail Product Sales Agreement No. 290386.

Individual Subscription rates 1 year (9 issues): domestic \$89; Mexico and Canada \$114, Foreign \$129; Institutional/Library rates: domestic \$199, Canada & Mexico \$224, Foreign \$239. To submit articles, please send electronic files on disk to the Editors at 885 Meadowslands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

POSTMASTER: Send domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sigs.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

Tactical patterns for the real world: Validation and informational patterns

Darrow Kirkpatrick

PREVIOUSLY IN THIS series of articles on patterns for efficiently implementing and managing domain models we looked at a family of patterns for dealing with instantiation issues. Interior Decorator showed how to share a set of useful behaviors that may be selectively needed throughout a family of classes in a broad hierarchy. Epitome showed how to consolidate and share the default values for an object's attributes. Actuator showed how to convert a constant attribute of an object to one that can vary during the lifetime of the object.

In this issue we explore two additional families of patterns: the first dealing with validation issues—checking and protecting domain objects, and the second dealing with informational issues—managing status and validation messages.

VALIDATION PATTERNS

Safeguard (Delayed Validation)

Problem. Where do you put complex validation logic; and how do you prevent invalid domain objects from being used?

Motivation. You want to allow user editing of a slope object that represents the simple equation

$$\text{slope} = (\text{elevation2} - \text{elevation1}) / \text{length}.$$

You wish to encapsulate in the slope class the domain validation rule that *length* may not equal zero, yet this means allowing some domain models to take on illegal values for *length* temporarily, so that it may be validated. You establish an overall domain model validation method, a Safeguard, to be checked before any attempt is made to calculate the model.

Applicability. Use this pattern when you need to validate domain objects to catch data-entry errors or other logical errors that would prevent correct calculation, and when the validation logic requires intimate knowledge

of the domain. It is particularly appropriate for top-level validation of complex models with many interrelated parts.

Solution. Associate complex validation logic with domain classes by writing validation methods. Allow data that may be invalid into the domain, but guard calculations with a validation method.

Implementation. Write a public validation method called `#isValid` for use by clients, delegating to a private `#validate` method that may be overridden by domain subclasses. The public method should reset error flags as necessary before the specific validation is invoked, and answer the result of the validation:

```
isValid
    "Answer a Boolean, whether this object is currently valid."
    ^self
        clearErrorFlag;
        validate;
        isErrorFlagSet

validate
    "Test all aspects of this object for validity and if any
    fail set the error flag."
    length > 0 ifFalse: [
        self setErrorFlag ]
```

Consequences. This pattern encourages the programmer to keep validation code near the domain state that it protects, improving encapsulation. But to maintain validation logic in the domain means you must allow bad data into the domain temporarily, and validate it later. A benefit of applying this pattern is that expensive validation occurs only on demand, rather than whenever there is a change to the model (which can be slow), or when indicated by a modified flag (which can be hard to maintain). A drawback is that clients must remember to check for validity before using the model.

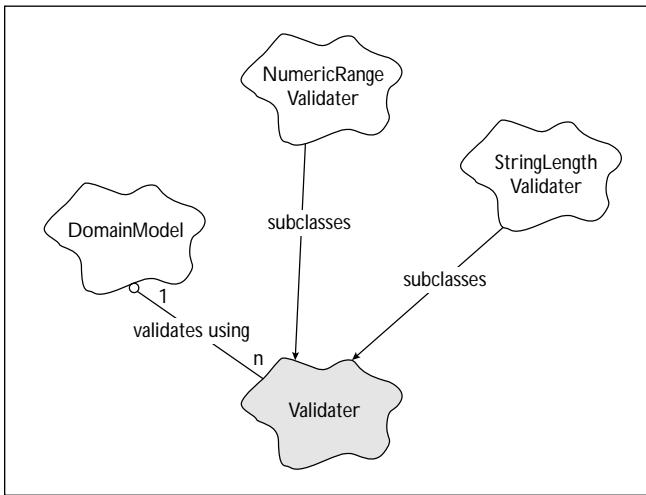


Figure 1. Validator class hierarchy.

Related Patterns. An alternative to this pattern is to use a Memento¹ to copy the domain object and make that Memento instead of the object available for editing. The Memento can then be validated before it is applied to the domain object. The drawback to this approach is the tendency of validation code to drift out of the domain into interface or broker classes.

See Verdict for more discussion of how the messages that result from validation may be managed. The `#isValid` method is an example of Template Method.

DEFLECTOR (DEFENSIVE SETTER)

Problem. How do you prevent attributes of certain classes from ever taking on illegal values?

Motivation. You maintain a global help level that user interface elements check to determine how much help and functionality they should reveal. Rather than require those clients to deal with illegal values, it is convenient to force the help level to remain within a certain range. To guarantee this range you implement a help level setter method that rejects illegal values, and you require all clients to use it.

```

helpLevel: anInteger
  "Set the help level for the system to the passed integer,
  if it is valid."
  (anInteger between: 1 and: 3 )
    ifTrue: [ helpLevel := anInteger ].
  ^helpLevel
  
```

Applicability. Often in a running system an attribute can be set from many different sources, such as the user interface, client code, or an initialization file. Some of these sources may be unreliable, yet the attribute must never become invalid. Use Deflector when attributes of critical objects must never take on illegal values and there is no opportunity to perform interactive validation before the value takes effect. The client that is setting the attribute does not require that the operation succeed for the system to remain stable, and is not prepared to handle an exception.

Solution. Perform validation against the passed value in the setter method itself. If the value is invalid, simply do not set it. There is no exception or error return. The current value of the attribute is always returned.

Implementation. Move validation code to the lowest level in the object implementation. If extra performance or bypassing validation are sometimes required, consider implementing a second, low-level, "basic" accessor that performs no validation before setting the attribute.

Consequences. This pattern leads to robust but possibly obtuse behavior. Because there is silent validation of attributes, users will be protected from—but not notified of—error conditions. Because there is some overhead to perform validation in accessors, this pattern is not appropriate for performance-critical code.

Related Patterns. Deflector is an example of early validation; Safeguard is an example of late validation.

VALIDATER (CONFIGURABLE VALIDATION)

Problem. How do you provide default validation of domain attribute values, while allowing end users to modify default validation logic safely?

Motivation. You are designing a system that models the flow of water, requiring a specific gravity to characterize the water. In certain situations it may be possible for an expert to use the system to model the flow of other kinds of fluids. To protect novice users you implement a default validation to check values entered for specific gravity against the range of legal values for water. However you objectify that validation as a persistent, editable object so that expert users may configure the system to accept values for other fluids.

Applicability. Use the Validator pattern when you wish to implement domain validation rules that can be relaxed or adjusted by the end user. The user-configurable portion of the rules must be represented by values that can be edited in a running program. The pattern as described here is for field-level attribute validation, such as range-checking, that requires no external context to perform.

Solution. Create an abstract validator class with concrete subclasses embodying state and behavior for different validation strategies.

Implementation. One approach to managing validators is to have domain classes maintain symbolic names to specify the type of validation each of their attributes should receive. These symbolic names are keys into a global or project-level dictionary of available validators. Have the object that is responsible for accepting edited attribute values—perhaps an Adapter—pass those values to the associated validator object for approval before being committed.

It may be convenient to have validators share the same

protocol as code blocks, so they can be used interchangeably. This way the same client code can perform validation with runtime-specified validators, or with more complex logic specified at development time via a block.

Consequences. This pattern assumes a more complex interaction with the user. A view should be in control of the validation process so the user can be notified and given the opportunity to correct domain attributes that fail validation. Also, you may need to provide separate editors for each validator subclass that can be configured by the user. The subtleties of editing validation parameters may confuse some end users.

Related Patterns. Validators are an example of the Strategy pattern. They objectify different algorithms for performing validation, and make them interchangeable. Often a Validator will be used by an Adapter that has responsibility for interfacing between an editor and domain model.

Reviewing the three validation patterns, Validator is another example of early validation, while Deflector is an example of the earliest possible validation, and Safeguard is an example of late validation. You might use Validator to support editing domain attributes from dialogs where field-level validation is required. You would use Deflector to protect attributes which are subject to change from any source, including other application code. And you would use Safeguard when performance is critical, or more context is needed to perform validation than is available from the attribute value alone.

INFORMATIONAL PATTERNS

Verdict (Visitor Message Token)

Problem. How do you manage the results of a complex and expensive validation across a series of domain objects so that their status or validity may be queried at a later time?

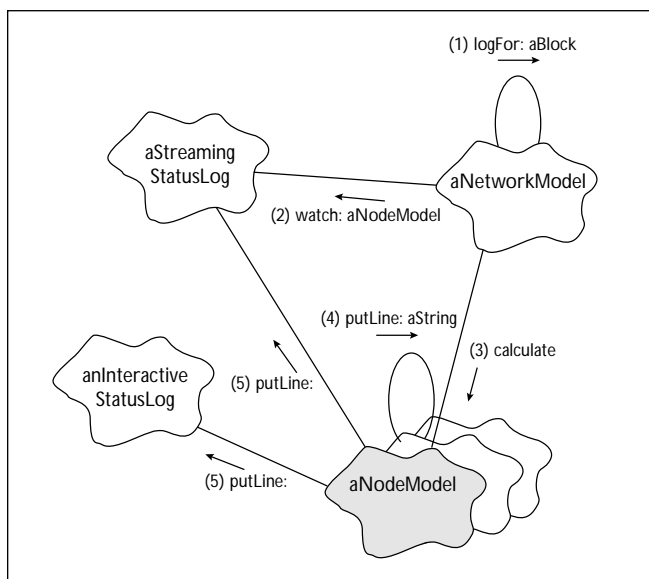


Figure 2. Ticker Tape object messages.

Motivation. You are designing the solution of a large network model. Each node in the network has associated constraints that must be met in order for the solution to be considered valid. If a node fails constraints the solution should proceed, but the failure must be stored so the user can be notified. To accomplish this you design the constraint checking algorithm to set a message symbol, a Verdict, into any node that fails constraints. Later, when the user inspects the node, the symbol is discovered and converted into a message for display.

Applicability. Use this pattern when it is not possible to interrupt validation, for example when validating domain objects requires context from an external object during an expensive traversal. And use it when the *results* of the validation must be stored for later use, for example when display of result strings may need to happen in a dialog during later editing of a domain object.

Usually it is an external, visiting object that leaves a token to be interpreted later. But rather than the visitor being another object, it might be a subclass that is providing validation behavior to add a Verdict into the object's message collection as validation proceeds.

Solution. Create a facility for storing within each domain object a collection of zero or more symbols that index into a global map of validation, warning, and error messages. When a validation is performed—and problems are found—add the appropriate symbols to the domain object's message collection. If there are no validation problems, remove the appropriate symbols from the object's collection, if they have been added during a prior validation.

Implementation. A more flexible solution for managing messages is to allow multiple categories by maintaining a dictionary in the domain object whose keys are types of messages (*constraint*, *warning*, and *error* for example), and whose values are sets of symbols representing specific messages. If a message dictionary is used, subclasses and clients can add their own message categories.

Consequences. This pattern moves validation behavior out of low-level domain objects into higher-level objects. Because of the threat to encapsulation this may not always be desirable, though it can be essential if additional context is required to perform the validation.

Applying this pattern throughout a domain hierarchy may result in wasted space if many objects don't need to maintain validation messages, especially if message category dictionaries are used. Consider applying the Interior Decorator pattern to save space. Note that the need to remove message tokens once objects are valid requires additional logic and can be prone to subtle bugs.

Related Patterns. A Verdict may be left by a Visitor performing validation. Or, subclasses may add Verdict messages by overriding portions of a validation implemented by a Template Method. A Safeguard method may use the *pres-*

ence of a Verdict message to indicate that the domain object is not valid.

TICKER TAPE (STATUS MESSAGE LOG)

Problem. How do you collect status information from a lengthy domain operation involving thousands of objects, none of which have visibility to the user interface?

Motivation. You are designing a complex network calculation. When standalone test suites are run during development, no status information is desired. However, when a user interface is present and calculations are occurring at runtime, feedback to the user is essential. Therefore you implement a publish and subscribe mechanism for domain model messaging. Domain models trigger messages as they calculate. Clients may subscribe to, collect, and present those messages if they wish.

Applicability. Use this pattern when one or more domain objects must perform a lengthy or complex operation for which status information may not always be desirable, and the formatting of the status information is a function of individual domain objects.

This pattern is also useful for debugging and tracing, and any time that status information must be collected for filtering or presenting later. It is not appropriate when heavy formatting or graphics are required—such as for WYSIWYG reporting.

Solution. Create a simple low-level protocol in domain objects for outputting formatted strings. The protocol should simply trigger an event with the string as argument. Create a family of status log classes that can subscribe to these events and present the status information with various levels of formatting and interactivity.

Implementation. The fundamental methods required are:

Model >> #logFor: aBlock to establish a status log for the duration of aBlock. This method is implemented in the top level domain object, which is performing the lengthy

operation. The method is responsible for instantiating the status log, traversing all the lower-level domain models so the log can subscribe to their events, evaluating the block (causing the lengthy operation to proceed), and then traversing again so the log can drop the models.

StatusLog >> #watch: aModel and StatusLog >> #drop: aModel in the status log object to subscribe to and cancel receiving the status events triggered by a domain model.

Model >> #putLine: aString to trigger the status event from the domain object. (Use a method of the same name to output the text in the status log.)

Consequences. Note that hooking up numerous domain object events to handlers in the status log object and releasing them afterwards may be expensive, but is relatively fast compared to the long operations for which this pattern is appropriate.

Related Patterns. Ticker Tape uses the Observer pattern to implement a publish and subscribe mechanism: the domain models are the subjects and the status logs are the observers.

Related Patterns. Ticker Tape uses the Observer pattern to implement a publish and subscribe mechanism: the domain models are the subjects and the status logs are the observers.

COMING UP

The concluding article in this series presents a family of patterns for dealing with optimization issues—implementing domain models that must perform well even though they incorporate extra levels of indirection to be persistent or transient. ☞

Reference

1. Gamma, E. et al., *Design Patterns*, Addison-Wesley, Reading, MA, 1994.

Darrow Kirkpatrick is Vice President of Research and Development at Haestad Methods, Inc., which specializes in numerical modeling for hydrology/hydraulics, and has pioneered using Smalltalk for shrink-wrapped Windows applications. Darrow enjoys hunting for patterns while leading a team of software engineers who have become experts at coaxing Smalltalk to perform in the real world. He can be contacted by phone at 203.755.1666 or by email at 75166.525@compuserve.com.

*This pattern moves
validation behavior
out of low-level domain
objects into higher-level
objects.*

Equality versus identity

Bobby Woolf

ONE OF THE trickier concepts in Smalltalk is the distinction between object equality and object identity. Any experienced Smalltalk programmer knows the difference: If two objects are equal, “=” (equal) returns true; if they’re identical, “==” (double-equal) returns true as well. But what does that difference actually mean in a logical programming sense? Should you design your code so that objects that are equal have to be identical? When should your code test with equal versus double-equal? The difference between equality and identity seems to be like fine art: You can’t say what it is exactly but you know it when you see it. Even though you know what it is, it’s very difficult to explain to someone else who doesn’t know. Yet understanding this difference is important if you want to use Smalltalk well.

A METAPHORICAL EXAMPLE

This little story illustrates the difference. (I can’t take credit for this story; it’s been floating around my company for years.) A person sits down at a restaurant table and notices the customer at the next table eating a delicious plate of lasagna. When the waiter arrives, the person points at the lasagna on the next table and says, “I want that.” How should the waiter fulfill this request? He could get another plate of lasagna just like it and serve it. Or he could take the plate of lasagna away from the customer at the next table and serve it to this person.

If the waiter brought a second serving, the second plate of lasagna would be equivalent to the first one. If the waiter took the serving off the next table, the second plate of lasagna would be identical to the first one. In this context, equality means that the two plates have the same properties but are not the same plate. Identity means that the two plates are really the same plate. The most practical difference is that two equal plates contain two servings of lasagna; two identical plates contain a total of one serving.

*What exactly makes
two objects equal is often
very subjective.*

SIGNIFICANCE IN SMALLTALK

When two seemingly separate objects have the same identity, that means they’re really just two different handles on the same object. A *handle* is a variable or literal you use to access an object. The variable does not really contain the object, it just contains a pointer to the object. It is often said that everything in Smalltalk is an object. It could also be said that everything in Smalltalk is a pointer to an object.

You can never actually work with the objects directly, you just work with the pointers to objects. Because everything is a pointer, there is never a need to distinguish between a pointer and the value itself.

POINTERS ARE SIMPLER

This explains some of the conventions that make Smalltalk code different (and simpler) than other languages. In C/C++, for example, a variable `yourPhone` might contain a `PhoneNumber`. If you copy `yourPhone` into `myPhone` (`myPhone = yourPhone`), whether the variables copy the value or share the same one depends on how the variables are declared, as shown in Listing 1.

C/C++ allows/forces you to manipulate the pointers specifically. Thus a programmer has to be careful to use them properly; a mistake can cause serious, often subtle bugs.

(By the way, you would not believe how much trouble I had writing that C++ example. I spoke to a couple of friends of mine who know C++ and told them the dilemma

Listing 1.

```
PhoneNumber myPhone, yourPhone;    // declares two instances of
                                   // PhoneNumber
myPhone = yourPhone;                // copies the PhoneNumber

PhoneNumber *myPhone, *yourPhone    // declares two pointers,
                                   // each points to a PhoneNumber

yourPhone = new PhoneNumber;        // initialization
myPhone = yourPhone;                // shares the PhoneNumber
*myPhone = *yourPhone;              // copies the PhoneNumber
```


I was trying to illustrate. Our conversation then got very complicated as they explained all the different things that the programmer would have to consider. Just trying to find a simple example that shows two variables that share a single object versus copies of the object was very difficult. Somehow I think that the difficulty I had preparing this example does more than the example itself to illustrate how complex C/C++ can be!

In Pascal, the programmer can declare a subroutine parameter to receive a variable either by value or by reference. If the variable is passed by value, it is copied, which is expensive for variables that contain a large amount of data. However, since the subroutine has its own local copy of the data, the subroutine can make all kinds of changes to the copy. It knows that those changes won't affect the rest of the program outside the subroutine. If a variable is passed by reference, the subroutine gets a pointer to the same data that the rest of the program is using. This pointer is inexpensive to create. However, the subroutine has to be more careful about the changes it makes to the parameter since those changes affect the original variable's data. Thus programmers that nonchalantly change a parameter's contents have to learn to be a lot more careful whenever that parameter is passed by reference.

Smalltalk programmers do not need to make these distinctions between a value and a pointer to the value. Any variable is just a pointer (see Listing 2).

So any Smalltalk variable declaration is equivalent to a C/C++ pointer variable declaration. Whereas Pascal parameters can be passed by reference (pointer) or value (copy), Smalltalk method parameters are always passed by reference. If you want to copy a variable's value in Smalltalk, you need to explicitly send it a message like copy. This makes Smalltalk's syntax simpler, its intent more explicit, and its code more consistent.

GARBAGE COLLECTION

Everything being a pointer helps explain how garbage collection works: When nobody's pointing to an object anymore, it gets garbage collected. Every variable is a pointer to a value. Code uses a variable as a handle to that object. As long as an object has a pointer to it, some code has a handle to access it, so it should not be garbage collected. But once there are no more pointers to an object, no one has a handle on it, so no one could access that object even if they wanted to. Since no one can access the object, it can safely be garbage collected and no one will miss it.

DISTINGUISHING OBJECTS

So how do you tell when two objects are actually the same object? You use the messages "=" (equal) and "==" (equal-equal or double-equal). *Equal* tells you whether two objects are equivalent, meaning that they represent two values where one is as good as the other. *Double-equal* tells you whether two objects are identical, which is to say that they are the same object. *Equal* tests for equality whereas *double-equal* tests for identity.

Listing 2.

```
| myPhone yourPhone |
yourPhone := PhoneNumber
    readFromString: '212-555-1010'. "initialization"
myPhone := yourPhone.             "shares the PhoneNumber"
myPhone := yourPhone copy.         "copies the PhoneNumber"
```

Double-equal is easy to design and implement. The double-equal method is defined in Object and cannot be overridden in subclasses. (Technically, double-equal can be implemented in subclasses, but the compiler ignores those implementors so they are never executed.) The implementation simply follows the two pointers to see if they point to the same address in memory. Even if two objects are alike in every possible way, if they're two different objects, they'll occupy two different locations in memory. Thus, they're copies: Changes to one are not reflected in the other. Yet it can be very difficult to tell them apart without changing them. The way to do this is to test if they're double-equal: Do they occupy the same location in memory?

Why is it important to know whether two handles point to the same object? Here's a simple example:

```
| set1 set2 |
set1 := Set withAll: #(a b c).
set2 := set1.
set2 add: #d.

Transcript cr; show: set1 printString.
Transcript cr; show: set2 printString.
```

When the Sets are displayed, set2 obviously contains #d, but does set1? Most new Smalltalkers would say that set1 does not contain #d because it was not sent "add: #d" like set2 was. The surprise is that set1 does contain #d just like set2 does because set1 and set2 are really two separate pointers to the same object. This code proves it:

```
| set1 set2 |
set1 := Set withAll: #(a b c).
set2 := set1.
Transcript cr; show: (set1 == set2) printString.
```

The result, true, shows that the two variables are really the same object. This is why when you change one of them, the other changes as well.

VisualWorks will actually show you an object's address, sort of, if you send it the message identityHash (previously called asOop). This returns a number that is unique for every object in the image. Thus two objects with the same address are really the same object and have the same identity-hash. This is helpful when you have two objects in two separate inspectors and you want to compare them to see if they're the same object. You can't very well use double-equal; which inspector would you run it in? But you can send identityHash to each object and visually compare the results.

WHAT IS EQUAL?

Although double-equal's implementation is clear and simple, equal's is anything but. Surprisingly, many developers do not seem to recognize this dilemma. Even for fairly oddball classes, programmers often think that implementing equal is perfectly straightforward. In one case, an experienced developer told me that if two views have the same model, they're equal (in his opinion). Since these two views might be two different kinds of widgets in two different windows, I had a hard time thinking of them as equivalent. Thus I find that what exactly makes two objects equal is often very subjective.

In VisualWorks, the most straightforward implementors of equal are in the ArithmeticValue hierarchy. Essentially, two ArithmeticValues (think of them as Numbers) are equal if the difference between them is zero. Similarly, if Dates or Times represent the same offset, they're equal. So the Magnitude hierarchy in general is pretty clear-cut.

Then again, Characters are Magnitudes, but what does it mean for them to be equal? ParcPlace says that the two characters must be the exact same one. However, couldn't it be said that "A" is, in a sense, equal to "a"? Maybe, maybe not (which helps explain Character>> same As:). The point is not that ParcPlace is wrong, but that when it comes to equal, the obvious answer is not necessarily so obvious once you think about it.

Consider the Collection hierarchy, the second-most fertile source of implementors of equal. For two Collections to be equal, they have to contain the same number of elements and the two elements in each position have to be equal. This seems reasonable. However, this eliminates Sets because they are unordered; two equal elements being in the same position in the two Sets is just coincidence. It also means that #(a b c) does not equal #(c b a); although the elements are the same, the positions are different. Is it obvious that equal should work this way? Maybe, maybe not.

Most other classes that implement equal—and there aren't a whole lot of them—do so pretty unimaginatively. They verify that both objects are of the same class/species and that their instance variables have the same values. Thus a BlockClosure is only equal to another BlockClosure that has the same method, outer Context, and copiedValues. Does that really happen a lot?

EQUALITY AND TYPE

For two objects to be equal, they have to be of the same type. Objects which are not of the same type are not comparable, so they cannot be equal.

However, what constitutes a type in Smalltalk is unclear. For the purposes of determining equality, there are four main ways of determining if two objects are the same type:

- Are they instances of the same class? BlockClosure>>=

(in VisualWorks) first checks that both objects are instances of the same class.

- Are they instances of the same species? SequenceableCollection>>= first verifies that the species of the two objects is the same.
- Are they instances from the same hierarchy? Interval>>= first confirms that the second object isKindOf: Interval.
- Do they claim to be the same type? String>>= initially confirms that the argument isString.

In each of these cases, equal makes sure that the two objects are of the same type by verifying that their classes are equivalent. This check is the first one made; if it fails, all other comparison is skipped.

Verifying that two objects are of the same type before making any other checks has an important benefit: equal can be used to compare any two objects and will not fail. If the argument is not of the right type, it will probably not understand the messages equal sends to it. This would cause a message-not-understood or similar error. Thus every

implementor of equal should first verify that the receiver and argument are instances of comparable classes.

*When it comes to equal,
the obvious answer is not
necessarily so obvious once
you think about it.*

EQUAL AVOIDS DUPLICATES

If you want to know whether two objects *should* be equal, here's the question to ask yourself: Should I be able to store both objects in a Set, or should the second be considered a duplicate of the first? A Set is a collection that eliminates duplicates. So what constitutes a duplicate? If the object being added is equal to an object that is already in the collection, the Set does not add the new object. This eliminates duplicates.

If two objects are equal, they cannot be stored in a Set together. If they're unequal, a Set will not eliminate either as being a duplicate of the other. So you can forget defining equal through logical semantics and theoretical dissertations; the simplest answer is: Do you want to be able to store them both in a Set? Of course now programmers will argue the semantics of whether both objects should be able to live in the same Set.

This also explains what an IdentitySet is. Whereas a Set eliminates duplicate elements, an IdentitySet eliminates identical elements. Thus a Set tests for duplication using equal; an IdentitySet uses double-equal.

An IdentitySet is slightly more efficient than a Set because double-equal is faster than equal. However, using an IdentitySet to eliminate duplicates is dangerous. The IdentitySet will eliminate duplicates as long as equal and double-equal work the same for the objects in the collection. Yet this will fail if another developer implements equal for these objects so that it works differently than double-equal. Thus it's safest to eliminate duplicates using a Set. Use an IdentitySet to eliminate identical objects, or to store objects where duplicates are guaranteed to be identical objects (such as Symbols).

EQUAL IS DOUBLE-EQUAL

Because it is often not obvious what makes two objects equal, the default implementor of equal is

```
Object>>= anObject  
^self == anObject
```

Thus for most objects, they're equal if (and only if) they're double-equal. If they don't have the same identity, they definitely don't have equality. Which is to say that equality doesn't make much sense for most objects. Identity always makes sense, which is why it's quite easy to design and implement. Equality, on the other hand, just doesn't make much sense for most types of objects.

You should avoid implementing equal arbitrarily. A standard protocol like equal is not very useful if its implementation is subjective and privy to the whim of the last person to implement it. If the user of your class can't tell what equal does without looking at your implementation, that method is not going to help him very much. In fact, it will hurt him if he's depending on all of the implementors in a hierarchy working polymorphically.

HASH

There is an easily overlooked but significant comment in VisualWorks' default implementor of equal:

```
Object>>= anObject  
"... If = is redefined in any subclass, consider also  
redefining the message hash."
```

This subtle suggestion is the only warning you get that equal and hash go hand-in-hand. This is because the chief user of hash (besides other implementors of hash) is findElementOrNil:. This is the method Set classes use to put an element in (add:) and find it again (includes: and remove:). For example, the implementor in Set contains these two lines:

```
Set>>findElementOrNil: anObject  
...  
index := self initialIndexFor: anObject hash  
boundedBy: length.  
[(probe := self basicAt: index) == nil or: [probe =  
anObject]] whileFalse: "keep looking"  
...
```

Notice that anObject is being compared in two ways, hash

and equal. This means that if two objects are equal, they need to hash to the same value. Otherwise, you could store an item and its equivalent together in the same Set.

For example, let's say you store "4.0" and "4" in a Set. Since they have the same hash value, the Set looks in 4.0's position, sees that it's equal to 4, and so doesn't store 4. If their hash values weren't equal, the Set would look in some other position, not find the 4.0, and store the 4 in the first empty slot it comes to. Similarly, if 4.0 is in a Set and you look for 4, you probably won't find it unless they have the same hash value.

By the way, the opposite is not true: two objects which hash to the same value do not need to be equal. A Set will avoid collisions more efficiently if unequal values have different hash values. Yet even if they do have the same hash value, the Set will handle the collision correctly because it realizes that they are not equal.

So whenever you implement equal, you should implement hash as well. If the implementor of hash is the same as its super-implementor, you don't need the new one, but it's important that you at least thought about it.

CONCLUSIONS

Here are the eight main points in this article:

- A handle is a pointer to an object. A variable is a handle.
- Everything in Smalltalk is a pointer to an object.
- Identity means that two handles hold the same object.
- Equality means that two handles hold equivalent objects, but equivalency is fairly subjective.
- For two objects to be equal, they have to at least be of the same type. Each implementor of equal should first check that both objects are of the same type.
- A Set considers two objects duplicates if they are equal. Duplicates in an IdentitySet are identical.
- Equality doesn't make sense for most types of objects, in which case equality is the same as identity.
- Two objects which are equal need to have the same hash value.

In my next article, I'll talk about three different types of instance variables. One of these types is instrumental in determining object equality. ☞

Bobby Woolf is a Senior Member of Technical Staff at Knowledge Systems Corp. in Cary, NC. He mentors Smalltalk developers in the use of VisualWorks, ENVY, and Design Patterns. He welcomes your comments at woolf@acm.org or at <http://www.ksscary.com>.



Bob Hinkle



Ralph E. Johnson

The active life is the life for me!

ALMOST EVERYTHING IN Smalltalk is an object, including language elements like classes, methods, processes, and contexts. Variables are a major exception to this rule of thumb. While global and class variables are objects in most implementations of Smalltalk, instance variables and temporary variables are not. That's too bad, because instance variables have many uses as objects. (Making temporary variables first-class seems less useful.)

For example, user interface widgets often wish to depend on a particular attribute of an object. VisualWorks has you represent these attributes with ValueHolders, which are objects that hold a single value. If you store an attribute in a ValueHolder, widgets can depend on the ValueHolder and be notified when that attribute changes. However, to change a design that doesn't use ValueHolders to one that does, you have to rewrite your program, changing accesses to attributes stored in ValueHolders. There are times when you want to keep the program as it is, but just change the way you store the attribute. In other words, you'd like to depend directly on a variable without explicitly having to store a ValueHolder in it. This feature is called active variables* and is very useful when you are debugging and fine-tuning a program.

In this column we use VisualWorks 2.0 to implement active variables in three steps. First, we define the class ActiveVariable and show how to convert an object's slots to contain instances of it. Next, in the largest step, we introduce a new class for an object containing one or more ActiveVariables. Finally, we use a new kind of MethodProducer to recompile methods in the new class.

Bob Hinkle is an independent Smalltalk consultant and writer. His current focus is the improvement of existing tools and the creation of new tools to revitalize the Smalltalk environment. He can be reached at hinkle@primenet.com.

Ralph Johnson learned Smalltalk from the *Blue Book* in 1984. He wrote his first Smalltalk program in the fall of 1985 when he taught his first course on object-oriented programming and design. He has been a fan of Smalltalk ever since. He is the only author of "*Design Patterns: Elements of Reusable Object-Oriented Software*" to regularly program in Smalltalk, and continues to teach courses on object-oriented programming and design at the University of Illinois.

ACTIVE IS AS ACTIVE DOES

To our knowledge, the first language to include active variables was LOOPS. Active variables descended to CLOS in the form of access daemons on slots. We're not the first to implement active variables in Smalltalk (see Messick¹ for example), and our specification is similar to what others have done. Our implementation of ActiveVariables is new, since we use the GenericCompiler and MethodProducer described in our previous columns. ActiveVariables satisfy the following high-level specification:

Class: ActiveVariable

Superclass: Object

Important instance variables:

name <String>

value <Object>

readDependents <Set of 2-argument BlockClosures>

writeDependents <Set of 3-argument BlockClosures>

Important instance methods:

value

Return the ActiveVariable's value, and also notify readDependents.

value: anObject

Set the ActiveVariable's value, and also notify writeDependents.

The read dependents are two-argument blocks, whose parameters are the ActiveVariable and its current value. These blocks are evaluated whenever the message #value is sent to the ActiveVariable, as follows:

Method for ActiveVariable

value

readDependents do: [:each | each value: self value: value].

^value

Similarly, the write dependents are evaluated whenever

* Source code for the active variables package is available by anonymous ftp from st.cs.uiuc.edu. Look for the file ActiveVariables20.st in pub/st_vw.

#value: is sent to the ActiveVariable. They are three-argument blocks, whose parameters are the ActiveVariable, its current value, and its old, over-written value:

Method for ActiveVariable

```
value: anObject
  "This method needs to return the new (i.e., being set)
  value, so the behavior is consistent when an
  ActiveVariable set replaces a := expression."

  | oldValue |
  oldValue := value.
  value := anObject.
  writeDependents do: [ :each | each value: self value:
    value value: oldValue].
  ^value
```

ActiveVariables are added to an object using #instVarAt:put:, the primitive method that gives direct access to object state. While this method is a gross violation of encapsulation, it's required for Smalltalk programming tools written in Smalltalk, such as the Inspector. We use the phrase "activating an object" to describe the process of adding ActiveVariables to the object's slots, and similarly define an "activated object" to be an object that has ActiveVariables implicitly stored in some of its slots. Once an object has been activated, its class must be changed, and ideally the slot conversion and class change should happen atomically, to prevent access errors. The object's new class has methods re-compiled to send #value and #value: to access any activated slots. We show how to create such a class in the next two sections.

ACTIVATED OBJECTS NEED ACTIVE CLASSES

Since an activated object needs specially compiled methods, it also needs a special class to store those methods. Furthermore, this class must be distinct for different objects, so that you can activate one of a class's instances without activating all of them. This kind of class, which is distinct for individual objects, is called a lightweight class, which we showed how to implement in *Debugging Objects*.² Activated objects are instances of ActiveClass, which is a new subclass of LightweightClass that adds specialized support for ActiveVariables. In addition, each ActiveClass needs some information for each of its variables that is activated. A new object called ActiveVariableSpecification maintains this information for the ActiveClass. These two new classes are specified by

```
Class: ActiveClass
Superclass: LightweightClass
Important instance variables:
  baseClass <Class>
  activeVariables <Collection of ActiveVariableSpecifications>
Important instance methods:
  activateObject: <Object>
  Create and install ActiveVariables in slots of the Object corresponding to the receiver's ActiveVariableSpecifications.
```

```
activeVariables
activeVariableIndexes
  Return respectively the collection of names and indexes of
  this ActiveClass' activated variables.
allActiveVariables
allActiveVariableIndexes
  Return respectively the collection of names and indexes of all
  activated variables in this ActiveClass and all of its transitive
  superclasses.
convertInstancesTo: <ActiveClass> addedIndexes:
  <Collection of SmallIntegers> from: <Behavior>
  Convert any instances of the receiver to have the input
  ActiveClass as their class. This involves changing their class,
  adding ActiveVariables to any slots numbered in the input collection,
  and recompiling methods that reference those slots.
noteNewActiveVariables: <Collection of
  ActiveVariableSpecifications>
  Stores any new ActiveVariableSpecifications in the active-
  Variables instance variable and converts the receiver's instances
  to support these specifications.
```

Important class methods:

destroyAllActiveClasses

Eliminates all active classes and their instances

Class: ActiveVariableSpecification

Superclass: Object

Important instance variables:

name <String>

index <SmallInteger>

globalReadDependents <OrderedCollection of 2-argument
BlockClosures>

globalWriteDependents <OrderedCollection of 3-argument
BlockClosures>

localReadDependents <IdentityDictionary mapping
objects to 2-argument
BlockClosures>

localWriteDependents <IdentityDictionary mapping
objects to 3-argument
BlockClosures>

Important instance methods

addDependentsFrom: <ActiveVariableSpecification>

Merges the ActiveVariableSpecification's dependents lists into the receiver's.

setDependentsOf: <Object>

Adds dependent blocks to the ActiveVariable in the Object's slot indicated by the receiver's index instance variable. All blocks from the global lists are added, as well as any associated with the Object in the local dictionaries.

setReadDependents: <Collection of 2-argument
BlockClosures>

setWriteDependents: <Collection of 3-argument
BlockClosures>

Respectively set the read- and write-dependent collections for this specification.

ActiveClasses create a new distinction between two kinds of

LightweightClass. Originally, a LightweightClass acted as an extension of an object's original class. The LightweightClass stood between the object and its original class, allowing the object to inherit methods from the original class as long as they weren't over-ridden in the LightweightClass. Most ActiveClasses will not be extensions of the original class, but instead they will completely replace the original class in the object's look-up chain.

Figure 1 illustrates these two different arrangements. In this example, aSet1 and aSet2 were both instances of Set, whose superclass is Collection. An extension LightweightClass named ExtensionToSet was created for aSet1 and inserted between aSet1 and Set. A replacement LightweightClass named ReplacementForSet was created for aSet2 and inserted between it and Collection, effectively removing Set from aSet2's lookup chain. aSet1 responds to #size using the method defined in Set, its original class, since that method is not over-ridden in ExtensionToSet. However, it responds to #isEmpty using the method defined in ExtensionToSet (which could still refer to Set's method #isEmpty by a super send). On the other hand, aSet2 responds to #size using the method from Collection. It responds to #isEmpty using the method defined in ReplacementForSet, which cannot access Set's method #isEmpty since Set is nowhere on Replacement ForSet's superclass chain.

There are two reasons why ActiveClasses are best implemented as replacement LightweightClasses. First, significant space can be saved when objects are activated from several classes in one class hierarchy. If ActiveClasses were extensions, and a DependentPart object and a CompositePart object were activated, the DependentPart's ActiveClass and the CompositePart's ActiveClass would both have recompiled ver-

sions of methods from their common superclasses. With ActiveClasses implemented as replacements, though, the two ActiveClasses would both inherit from an ActiveClass for VisualPart, eliminating that method duplication. The second justification is that extension classes must change the way they compile methods that use super. Using the same example, if a VisualPart method referencing super were naively recompiled in the DependentPart's ActiveClass, super would refer to VisualPart and not Object. Using replacements, the ActiveClass hierarchy parallels the original class hierarchy, guaranteeing that the meaning of super in recompiled methods will remain correct.

Using replacements instead of extensions doesn't remove all technical difficulties, of course. For example, suppose three objects with the same class are activated, only each activates a different subset of the class's variables. How many ActiveClasses should be created? On one extreme, we could create three ActiveClasses, one for each object. Should another object of the same class be activated in exactly the same slots, it could share the ActiveClass of one of these three, but otherwise a new ActiveClass would be required. This answer guarantees that each ActiveClass minimally matches its instances' needs for activation, but also has the potential to waste space. On the other extreme is the solution we've adopted, which uses only one ActiveClass for every class. In the above scenario, this ActiveClass will activate all slots needed by any one of its activated instances. As a result, some objects will have variables activated unnecessarily. This will not cause any incorrect behavior, though it will unnecessarily (but insignificantly, in our opinion) slow the objects' access to their variables. As another result, a new instance variable named activeVersion is added to Behavior, and thus to every existing and added class in the system. This variable will either hold the Behavior's unique active version or nil if it has none. In addition, we've added a number of new methods to Behavior. These new additions are summarized by

Class: Behavior

Added instance variables:

activeVersion <nil | ActiveClass>

Added instance methods

activateObject: <Object>

variableIndexes: <Collection of SmallIntegers>

readDependents: <Collection of 2-argument

BlockClosures>

writeDependents: <Collection of 3-argument

BlockClosures

Installs ActiveVariables in all slots of the Object indicated in the #variableIndexes: parameter. Adds the given read- and write-dependents to these ActiveVariables. Changes the Object's class to an appropriate ActiveClass.

activeSuperclass

If the Behavior's activeVersion isn't nil, returns the active Version's superclass. Otherwise returns the Behavior's superclass.

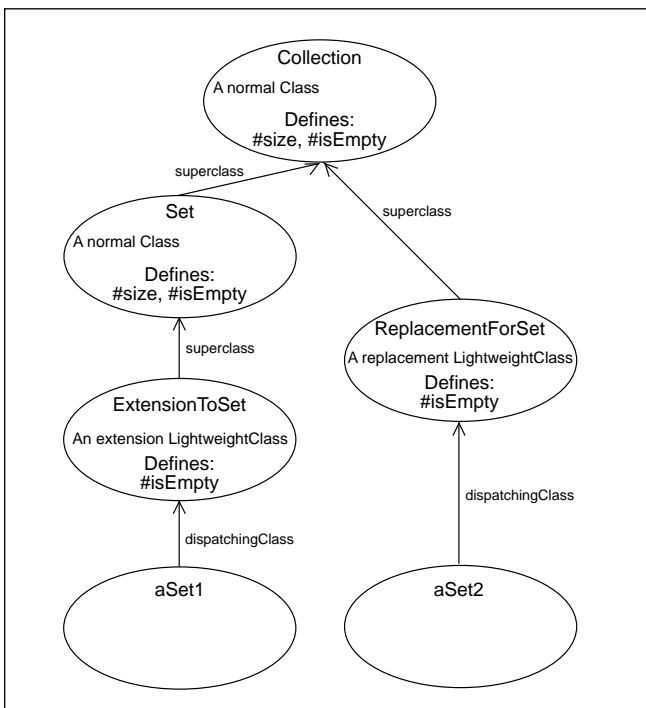


Figure 1. The difference between extension and replacement LightweightClasses.

```
buildActiveClassForVariables: <Collection of
    ActiveVariableSpecifications>
buildActiveClassForVariables: <Collection of
    ActiveVariableSpecifications>
    from: <Behavior>
```

Do whatever work is necessary, which can sometimes be quite a lot, to build an ActiveClass that activates all slots indicated by the input collection.

```
buildActiveSuperclassForVariables:
    <Collection of ActiveVariableSpecifications>
```

Obtains an ActiveClass for the Behavior's superclass that activates any variables in the input collection that are defined in that superclass or higher.

You activate an object by sending it #activateVariables: readDependents:writeDependents:. The first parameter is a collection of variable names to be activated, and the last two respectively are collections of read- and write-dependent blocks to be registered for all the slots being activated. The implementation in Object forwards the message to Behavior, which implements it as:

Method for Behavior

```
activateObject: object variableIndexes: indexCollection
readDependents: readBlocks writeDependents:
writeBlocks
    | specs activeClass names |
    names := self allInstVarNames.
    specs := indexCollection collect: [ :each |
        self activeVariableSpecClass new
            name: (names at: each) index: each;
            setReadDependents: readBlocks for: object;
            setWriteDependents: writeBlocks for: object].
    activeClass := self buildActiveClassForVariables: specs.
    activeClass activateObject: object
```

This method has three major steps, corresponding to its last three statements. The #collect: loop creates an ActiveVariableSpecification for each slot. A dependent registered with an ActiveVariableSpecification can be associated with one particular activated object, as is done above, or it can be registered globally. With the latter option, you can monitor all accesses to a particular variable across a group of activated objects. The second step sends #buildActiveClassForVariables: to create an ActiveClass that re-compiles all methods referencing variables in indexCollection. The final statement sends #activateObject: to the active class, which is defined as:

Method for ActiveClass

```
activateObject: object
    | oldActiveVariableIndexes newActiveVariableIndexes
      names |
    oldActiveVariableIndexes := object dispatchingClass
        allActiveVariableIndexes.
    (newActiveVariableIndexes := self
        allActiveVariableIndexes)
```

```
removeAll: oldActiveVariableIndexes.
names := baseClass allInstVarNames.
newActiveVariableIndexes do: [ :each |
    object instVarAt: each put:
        (self activeVariableClass
            name: (names at: each)
            initialValue: (object instVarAt: each))].
object changeClassToThatOf: self basicNew.
self allActiveVariables do: [ :each |
    each setDependentsOf: object]
```

The bulk of this message (up to the #changeClassToThatOf: send) creates and installs ActiveVariables in the newly activated slots of the input object. It's important to install ActiveVariables only in newly activated slots, since otherwise the code could produce limitless chains of nested ActiveVariables. The initial value for each new ActiveVariable is the old (non-active) value for the corresponding slot. After the ActiveVariables are installed, the receiver becomes the activated object's new class (and, as we mentioned before, this should ideally happen atomically with the slot conversion). Finally, the ActiveVariableSpecifications help initialize the newly activated object by copying their read- and write-dependents for that object, as well as all global dependents.

Behaviors respond to the #buildActiveClassForVariables: message used above by sending themselves #buildActiveClassForVariables:from: with the second parameter equal to self. This method constructs an ActiveClass for a given set of ActiveVariableSpecifications, which may require creating ActiveClasses for superclasses or converting existing ActiveClasses to support newly activated slots. Converting an existing ActiveClass may require work on its subclasses, since newly activated slots require the installation of ActiveVariables and the re-compilation of all referencing methods in the class's instances and also in the instances of any classes that inherit from it. Thus, the method that does the conversion (which has the longish name #convertActiveClassWithSuperclass:addNewActiveVariables:from:) calls back to #buildActiveClassForVariables:from: again. The sendingClass is passed in to ensure that every class gets converted only once.

Method for Behavior

```
buildActiveClassForVariables: variableSpecs from:
    sendingClass
    | newActiveSuperclass |
    newActiveSuperclass := self buildActiveSuperclass
        ForVariables: variableSpecs.
    self activeClass notNil
        ifTrue: [self convertActiveClassWithSuperclass:
            newActiveSuperclass
            addNewActiveVariables: variableSpecs
            from: sendingClass]
        ifFalse: [self buildActiveClassWithSuperclass:
            newActiveSuperclass
            withVariables: variableSpecs].
    ^self activeClass
```

`#buildActiveSuperclassForVariables:` generates an `ActiveClass` for the superclass if it defines any of the activated slots. It does so with another send of `#buildActiveClassForVariables:` from:, passing in the receiver as the sendingClass. The superclass needs to know who originated the recursive message to eliminate redundant operations during its activation.

Method for Behavior

```
buildActiveSuperclassForVariables: variableSpecs
| superclassActivatedSlots |
superclassActivatedSlots :=
    variableSpecs select: [ :each | each index between:
1 and: self superclass instSize].
^superclassActivatedSlots isEmpty
ifTrue: [
    self activeSuperclass]
ifFalse: [
    self superclass
    buildActiveClassForVariables:
superclassActivatedSlots
from: self]
```

The method `#convertActiveClassWithSuperclass:addNewActiveVariables:from:` is called from `#buildActiveClassForVariables:from:` when the receiver already has an activated version. It builds a new `ActiveClass`, figures out exactly which variables are newly activated, and then converts instances of the old `ActiveClass` to the new format using `#convertInstancesTo:addedIndexes:from:`.

Method for Behavior

```
convertActiveClassWithSuperclass: newSuperclass
addNewActiveVariables: variableSpecs from: sendingClass
| myActivatedSlots activeClass newActiveClass
addedIndexes |
myActivatedSlots :=
```

```
variableSpecs select: [ :each | each index between:
self superclass instSize + 1 and: self instSize].
activeClass := self activeClass.
myActivatedSlots := myActivatedSlots, (newSuperclass
variableSpecsNotIncludedIn: activeClass superclass).
newActiveClass := activeClass copy.
newActiveClass
    assignSuperclass: newSuperclass;
    noteNewActiveVariables: myActivatedSlots.
addedIndexes := myActivatedSlots asOrderedCollection
collect: [ :each | each index].
addedIndexes removeAll: activeClass
allActiveVariableIndexes ifAbsent: [].
activeClass
    convertInstancesTo: newActiveClass
addedIndexes: addedIndexes
from: sendingClass.
activeVersion := newActiveClass
```

The variable `myActivatedSlots` contains the activated variables that are defined in the receiver, plus any newly activated slots in `newSuperclass`. We need to allow for active superclasses that activate more variables than we request since every `Class` has just one `ActiveClass` version. For example, suppose we activate the variable “icon” in a `ScheduledWindow` and later activate the variables “application” and “label” in an `ApplicationWindow`. `ApplicationWindow` defines “application”, but its superclass `ScheduledWindow` defines “label”, so the activation process has to obtain an `ActiveClass` for `ScheduledWindow` using `#buildActiveSuperclassForVariables:`. This `ActiveClass` will activate “label”, as desired, but it will also activate “icon” because of the previous request. This technical point is an easy one to overlook in an implementation. (Or so we’d like to think, since we missed it in our earliest efforts!)

Figure 2 illustrates how activation spreads around the class hierarchy and why it’s important to identify the source of the `#buildActiveClassForVariables:from:` message. In the diagram, the dashed arrows represent the superclass relationship, and the thick arrows represent message sends. If “widgetFlags”, “components”, and “container” are activated for `aBorderDecorator`, activation must spread up from `ActiveBorderDecorator` to `ActiveCompositePart`, where “components” is defined, and to `ActiveVisualPart`, where “container” is defined. When `ActiveCompositePart` recompiles methods to activate “container” and “components”, it must also update all of its instances and all of its transitive subclasses’ instances. However, these activation echoes must spread away from the `ActiveBorderDecorator` `ActiveCompositePart`-`ActiveVisualPart` branch, since their methods and instances will be converted as a

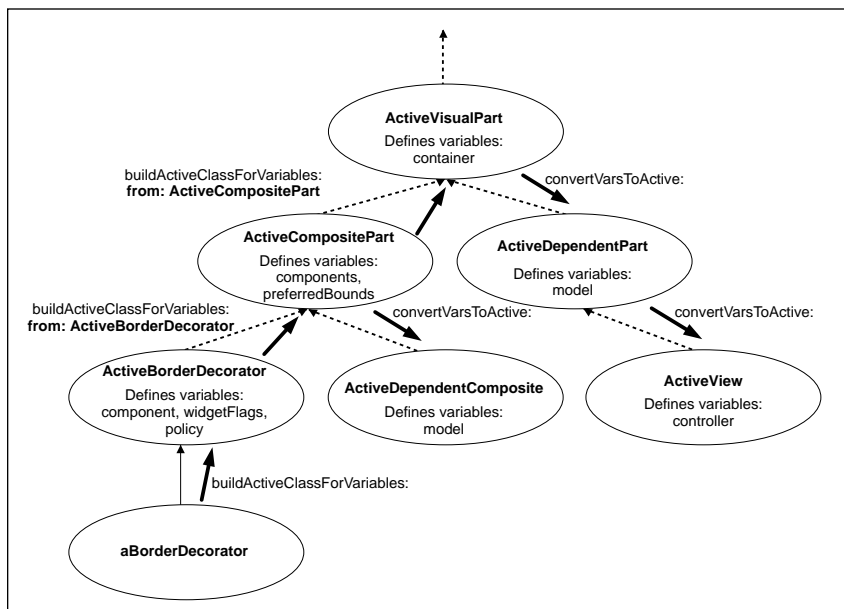


Figure 2. How activation spreads throughout the class hierarchy.

result of `#buildActiveClassForVariables:from:` sends `active` for each of them. Thus, when `ActiveCompositePart` activates “container” and “components”, it will send `#convertVarsToActive:` to `ActiveDependentComposite` but not to `ActiveBorderDecorator`. Similarly, when `ActiveVisualComponent` activates “container”, it will send `#convertVarsToActive:` to `ActiveDependentPart` (and thus indirectly to `ActiveView`) but not to `ActiveCompositePart`.

The method `#convertInstancesTo:addedIndexes:from:` is used when an existing `ActiveClass` is changed by the addition of new `ActiveVariables`. It converts all the instances of the old `ActiveClass` to the new one. The argument `indexCollection` indicates the newly activated variables.

Method for ActiveClass

```
convertInstancesTo: newActiveClass addedIndexes:
indexCollection from: sendingClass
| templateObject instances |
(instances := self allInstances) isEmpty
ifFalse: [
    templateObject := newActiveClass basicNew.
    self convertIndexesToActive: indexCollection in:
        instances.

    instances do: [ :each |
        each changeClassToThatOf: templateObject].
    newActiveClass updateInstanceDependentsIn:
        instances].
self subclasses do: [ :each |
    each baseClass == sendingClass
    ifFalse: [
        each assignSuperclass: newActiveClass.
        each withAllSubclasses do: [ :sub |
            sub updateInstancesForActivatedIndexes:
                indexCollection]]]
```

The message `#convertIndexesToActive:in:` cycles through the passed-in collection and creates an `ActiveVariable` for each newly activated slot. Each instance then has its class changed to the new `ActiveClass`. The `#updateInstanceDependentsIn:` message iterates over the old `ActiveClass`'s instances and adds the dependents from its `ActiveVariable` Specifications. Finally, the old active class moves its subclasses (except for `sendingClass`) to the new `ActiveClass`. It changes their superclass, and then has them and all of their subclasses activate the appropriate slots and recompile methods that reference any newly activated slots. This step, finally, is where the `sendingClass` parameter is used. The `sendingClass` shouldn't be converted, but only other subclasses of the old `ActiveClass`. The `sendingClass` subclass itself is processed in a `#activeClassForVariableSpecs:from:` context lower down in the execution stack. When that context resumes, that `sendingClass` will be converted appropriately, so it's vital (to avoid duplicate `ActiveVariables`, for example) that it not be converted here. Thus, as figure 2 showed, when class activation flows up the class hierarchy, any activation echoes flowing back down the hierarchy must flow away from classes passed up as `sendingClass` parameters.

The method `#buildActiveClassWithSuperclass:withVariables:` is called from `#buildActiveClassForVariables:from:` when the receiving Behavior doesn't already have an active version.

Method for Behavior

```
buildActiveClassWithSuperclass: newSuperclass
withVariables: variableSpecs
| myActivatedSlots |
myActivatedSlots := variableSpecs select: [ :each |
    each index between: self superclass instSize + 1
and: self instSize].
activeVersion := (ActiveClass newWithBase: self)
    copyAllMethods.
myActivatedSlots := myActivatedSlots, (newSuperclass
    variableSpecsNotIncludedIn:
        activeVersion).

activeVersion
    assignSuperclass: newSuperclass;
    noteNewActiveVariables: myActivatedSlots
```

In `ActiveClass>>copyAllMethods`, the new `ActiveClass` copies all methods defined in the receiving Class, which is its base class. The new `ActiveClass` inherits from the superclass determined earlier. In `#noteNewActiveVariables:`, the new `ActiveClass` stores `ActiveVariable` Specifications in its `activeVariables` instance variable and recompiles methods accessing activated slots to use `#value` and `#value:`.

THE ACTIVE COMPILER

The process of converting active classes relies on the `#recompileMethodsReferencing:` message, which is called whenever an `ActiveClass`' set of `ActiveVariable` Specifications changes.

Method for ActiveClass

```
recompileMethodsReferencingAny: indexCollection
| m |
self selectors do: [ :each |
    m := self compiledMethodAt: each.
    (m usesAny: indexCollection)
    ifTrue: [self recompile: each]]
```

This method recompiles any method of the receiver that uses, by reading or by writing, any named slot whose index is in the parameter `indexCollection`. This recompilation ensures that all activated slots are accessed only by `#value` and `#value:` sends. These `#value` and `#value:` accesses are inserted by a special object called `ActiveProducer`.

`ActiveProducer` subclasses from the `MethodProducers` we discussed in our previous two columns. It introduces a new name scope, `ActiveNameScope`, which in turn creates an instance of `ActiveLocalScope` for any `ActiveClass`. During compilation, local name scopes create instances of subclasses of class `Variable` for each name in a parse tree. The choice of subclass affects the code generated to access the variable. For example, `InstanceVariable` emits primitive bytecodes for direct instance access. `StaticVariable`, which is used for global names, emits more specialized code. When the compiler encounters a write to a variable, it sends `#emitStorePop:value:from:` to the corre-

sponding Variable object. StaticVariable implements this method as:

```
emitStorePop: codeStream value: value from: assignment  
"Emit code to assign a value to the variable."
```

```
self checkStore: codeStream from: assignment.  
codeStream pushConstant: binding.  
value emitValue: codeStream forAssignment:  
assignment.  
codeStream noteSourceNode: assignment.  
codeStream sendNoCheck: #value: numArgs: 1.  
codeStream pop
```

The underlined statements are the ones that actually generate code. The first generates bytecodes to push the StaticVariable's binding, which will be an Association, onto the stack. The next causes the assigned value to generate its own bytecodes. The third statement generates a #value: message send, which will store the assigned value into the Association's value instance variable. ActiveLocalScope instantiates a new class, ActiveInstanceVariable, to represent active named slots. ActiveInstanceVariable overrides the load- and store-emitting messages in a way similar to StaticVariable. For example:

```
emitStorePop: codeStream value: value from: assignment  
"Emit code to assign a value to the variable."
```

```
self checkStore: codeStream from: assignment.  
codeStream putLoadInst: index scope: scope.  
value emitValue: codeStream forAssignment:  
assignment.  
codeStream noteSourceNode: assignment.  
codeStream sendNoCheck: #value: numArgs: 1.  
codeStream pop
```

This is the same as StaticVariable's implementation except for the underlined statement. In contrast to the StaticVariable above, the ActiveVariable is accessed using the normal load for an instance variable. Since this method generates a #value: send, the #value: message will be sent to the ActiveVariable at run-time, allowing it to store the new value in its value instance variable and to alert its writeDependents.

Because StaticVariables are compiled specially, they also require special handling in the Decompiler. ActiveInstanceVariables don't require this as implemented, since they are only supported by ActiveClasses, which never decompile their methods. If you are interested in supporting ActiveVariables on Classes, you will have to add analogous special handling for ActiveVariables to the Decompiler.

CONCLUSION

The implementation we've described can be used to monitor instance variable accesses on a per-object basis. That's useful in its own right, and in addition we'll use active vari-

ables in our next column to implement watchpoints. Our watchpoints will be arbitrary expressions you can enter into a Debugger, which will then alert you if and when the value of the watchpoint expression ever changes.

There are a few limitations of our implementation that bear further investigation. Most notably, once an active version of a class is created, there is no connection between methods in the Class and its ActiveClass. Thus, you might change methods in or add methods to the class, but these changes would not be reflected in the ActiveClass. As a result of this limitation, you will have to purge ActiveClasses periodically after you've made programming changes. Some day we hope there will be a detailed dependency mechanism that would alert interested parties whenever a class' definition changes. (In fact, such a mechanism is one of the many potential projects for a future column.) In that case, we could use the dependencies to keep ActiveClasses in synch with their base class. Another extension that you may find interesting to attempt is to support the activation of indexed instance variables as well as named instance variables. In some ways this is an easier problem, since indexed variables can only be accessed via messages, but there are still some interesting issues that must be resolved to integrate indexed variables into our scheme.

We have three main goals when writing these columns. First, we try to present projects that can help improve the quality and productivity of Smalltalk programmers, and we hope that you find these ActiveVariables useful for exploring and debugging your code. Second, we try to do some interesting and perhaps unusual Smalltalk programming, to give you an idea of the things that are possible. In this column we leveraged the LightweightClasses and MethodProducers of our previous articles to implement ActiveVariables. We also showed another way to specialize the compilation process for a very particular need. We want to delve into the heart of the Smalltalk environment to help you understand it better and see possible ways to extend it for your own benefit. This month we spent most of our time discussing new code, but in the process we hope you learned a little about Behaviors and the compilation process. Finally, we are interested in hearing from you if you have comments or questions. Are we achieving our goals as far as you are concerned? Are there particular areas of the environment you'd like to understand, or advanced projects you'd like to see implemented? If you have any thoughts or feedback, please send them to Bob Hinkle by email at hinkle@primenet.com. ☐

References

1. Messick, S. L. and K. L. Beck. "Active Variables in Smalltalk-80." *Technical Report CR-85-09*. Computer Research Lab, Tektronix, Inc. 1985.
2. Hinkle, B., V. Jones, and R. E. Johnson. "Debugging Objects." *The Smalltalk Report*, (2) 9, July-Aug. 1993.



Jay Almarode

Configuring server Smalltalk

IN THE PAST, I've described major functional and performance differences between client Smalltalk and server Smalltalk. The client Smalltalk virtual machine operates as a single process that manages objects in virtual memory. Server Smalltalk, on the other hand, operates in a multi-process architecture where the domain of objects can extend beyond the range of virtual memory. Server Smalltalk must coordinate the creation, synchronization, and termination of multiple processes that perform such tasks as: execute a user's Smalltalk code, perform background garbage collection, coordinate multiple users transaction activity, serve disk pages to clients across a network, and manage shared page caches. To provide the needed performance, server Smalltalk is implemented to take advantage of the features of server-class machines, such as shared memory, asynchronous IO, raw disk partitions, and SMP CPU configurations.

Obviously, configuring and tuning multi-user, server Smalltalk systems is very different from tuning single-user, client Smalltalk applications. In client Smalltalk applications, the main considerations when tuning are execution speed, runtime memory footprint, and image size. When tuning server Smalltalk systems, additional considerations are system-wide transaction throughput, amount of data transfer to clients, and disk IO rates. The design of the overall system configuration must consider different hardware and operating system parameters, such as the amount of swap space, file system buffers, availability of raw disk partitions, the number of semaphores, or the amount of shared memory.

Due to the multiprocessing nature of server Smalltalk, system designers have a number of options when configuring applications to run in a client/server environment. One configuration option is deciding where to execute the behavior of server objects. Each session that is logged into the server has its own virtual machine process to execute server object behavior. Therefore, applications can be configured to create that process on whatever machine it wants. One option is to link the session into the same

process as the client virtual machine. This means that both virtual machines are executing within the same virtual memory address space.

Another option is to have the session reside in its own separate process, and communicate with the client Smalltalk through remote procedure calls. This allows a system designer to configure the system so that some session processes reside on client machines, some on the server machine, and others on a third machine. Note that none of these configuration choices impact application code. Application code does not need to know where the

execution of server object behavior takes place, and the configuration can be changed with no modifications to application code.

Figures 1—3 illustrate some possible configurations for the location of the session processes. In Figure

1, a single session process is linked with the client process. This makes for fast replication of server objects in the client Smalltalk, but might cause much data transfer over the network when many server objects are read or written. In Figure 2, the session process is separate and resides on the same machine as the client Smalltalk. This configuration enforces data integrity because server objects and client objects reside in different virtual memory address spaces. If a bug in the client application should cause random memory locations to be written with bad data (sometimes called "wild stores"), the server objects are protected by operating system features that prevent one process from writing over another. Depending upon the application, this configuration might suffer from too much data transfer over the network as well.

In Figure 3, multiple session processes reside on the same server machine. One client application has a single session logged into the object server, while the others have multiple sessions logged in. This configuration enforces data integrity also, and data access is faster since the session processes are on the same machine where the disks re-

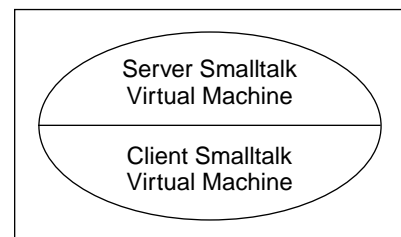


Figure 1. Server and client virtual machines linked.

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@gemstone.com.

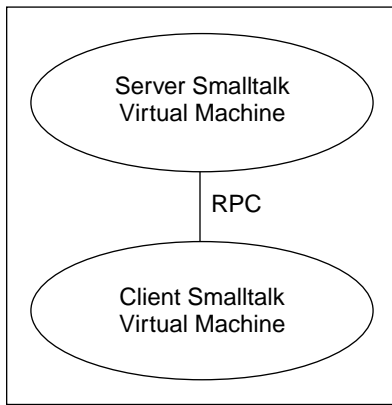


Figure 2. Server and client virtual machines separate.

side. The key advantage of this configuration is that many sessions can take advantage of a shared page cache of server objects. In most applications, a large percentage of objects are read only, and fewer are actually written. In many cases, multiple users are reading the identical objects. Classes

and methods are prime examples of objects that are read only during normal application execution. When multiple sessions can share objects in the shared page cache, it saves space and decreases access time, since common objects remain in the cache. A shared page cache can exist on other machines as well; a shared page cache can be created on any machine where a session process is to execute. In general, it is good practice to create a shared page cache on any machine where more than one machine will execute.

There are a number of parameters to tweak when configuring the multiple processes that make up server Smalltalk. The three main kinds of processes in which to configure memory requirements are the server process, the shared page cache and its monitoring process, and the session processes. For each kind of process, various statistics are available to monitor and observe the effects of changing configuration parameters. In GemStone, you can get statistics about various processes that make up the system by executing the expression "SystemcacheStatistics:aProcess Slot". This message returns an array of information according to the kind of process being described. To get a description of each element in the array, you can execute "SystemcacheStatisticsDescription". For statistics gathering purposes, each process is assigned a process slot, and a process executing Smalltalk code can get its own process slot by sending System myCacheProcessSlot. Among the information that you can retrieve for every kind of process is its process name, process ID, session ID, page reads and writes, and

cache hits and misses. For the remainder of this column I will discuss the configuration parameters of the three main kinds of processes and describe the relevant cache statistics for tuning performance.

The server process has a number of responsibilities, including synchronizing the transactions of the clients, arbitrating the locking of objects, and allocating object identifiers for clients to use when creating new objects. As the server allocates resources such as transaction records, locks, or object identifiers to each session, it stores this information in its private page cache. The size of this private page cache should be adjusted according to the number of sessions that are typically logged into the server most of the time. If the server's private page cache is filled up, then it overflows into the shared-page cache, affecting the performance of other sessions.

There are a number of statistics that help measure system throughput. For the server process, one can look at the #TotalCommits statistic to get the total number of transactions committed since the server process started. This can be used to measure systemwide transaction throughput. Another relevant statistic is the #NumberOfCommit Records. This is the number of outstanding transaction records that are currently being maintained by the server process. A large number could mean that there is a long-lived transaction that is preventing the server from reclaiming resources allocated for sessions created later.

The shared-page cache is where multiple sessions share pages of objects. When a session process needs to access an object, it first checks to see if the page containing that object is already in the cache. If so, it reads the page directly from shared memory. If the page is not present, the process reads it from the disk into the cache, where it becomes available to other processes. When configuring the shared-page cache, a system designer considers the total size of the object repository, as well as the number of sessions that will utilize the shared page cache simultaneously.

There are a number of statistics one can look at to monitor the activity of the shared page cache. One statistic that provides some indication of the utilization of the cache is the #NumberOfFreeFrames. This is the number of unused page frames in the shared page cache. Another statistic, #SharedAttached, is the number of pages that are being utilized by more than one process. This indicates the amount of sharing in the cache. When pages in the shared cache are written, they are scheduled to be written to disk when the transaction commits. The statistic #GlobalDirtyPage Count gives the number of pages in the shared cache that have been modified, but are not eligible for writing to disk because they have not been committed yet. If this value is large, then large transactions that write a lot of objects may be taking up space in the cache, or the server's private page cache may be too small (so it is using the shared page cache for the overflow). This statistic can be compared against #LocalDirtyPageCount, which is the total number of pages that have been modified and are eligible for writing to disk.

As described earlier, each session executes the behavior of server objects with its own

continued on page 28

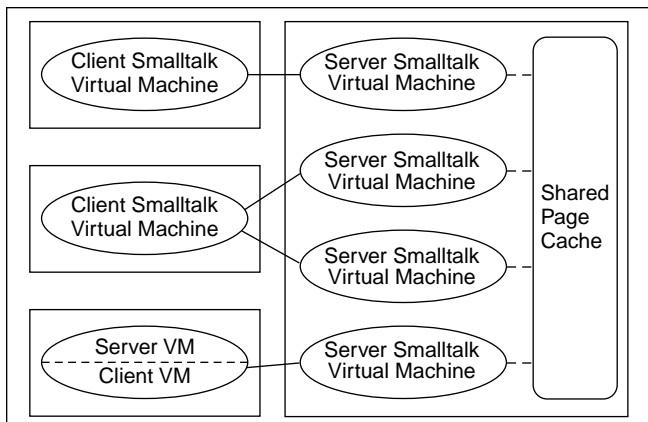


Figure 3. Server virtual machines sharing pages.

decide, 'Why don't we use something less complicated, like C++.'

The study STIC released last year found that companies adopting Smalltalk were more likely to have followed a formal process in choosing a programming language. "If we can get people to do real comparisons, then Smalltalk has a significant advantage," Phillips concluded. "Smalltalk seems to have to fight its way into an organization, but once it's there, it does pretty well." Smalltalk projects also were twice as likely to achieve their expected goals. "The Smalltalk industry has the opportunity to grow and prosper because of the successes that are there. It's a matter of getting the word out," Phillips said.

To Adele Goldberg, the issue is not just teaching Smalltalk, but teaching systems building as opposed to programming. "Too many university computer science curriculums stop at teaching data structures and algorithms," she said. It's not surprising it's so hard to recruit people capable of building extensible, adaptable systems. "The most significant part about a system is that once we start it up, there's a maintenance issue. You want it to run indefinitely." And while people can learn the syntax for programming in Smalltalk in an afternoon, "they don't get the systems building part," Goldberg said.

Her solution is LearningWorks, a modified version of the Smalltalk implementations she used to teach programming to 12-year-olds. Its interface is organized into

a neat binder of several "books" used for system planning, experimentation, and development, and it feeds students the modern Smalltalk class library a little at a time. Using the internet as a medium for distributing this free tool, she plans to have Open University students collaborate on building LearningWorks systems as class projects.

Students can start by experimenting with rehearsal worlds that illustrate key concepts and provide a context for exercises in organizing behaviors and allocating responsibilities, Goldberg said. Businesses could train their employees by having them create LearningWorks books that represent the essence of the company's framework.

Reg Krock of the Ontario manufacturing firm Maksteel was one of the people who approached Goldberg after her talk to express interest in obtaining a copy of LearningWorks. "One reason is that we have a 67-year-old president of our company. I could give that to him, and he would actually play with it."

Computer systems are the only part of the business that Maksteel's president doesn't fully understand, which makes it harder for him to manage, Krock said. "There's always been a language gap between the CEO and the CIO. What I'd like to do is take some of the mystique out of it." ■

David Carr is a freelance writer specializing in the object-oriented programming industry. He can be reached at davecarr@pcnet.com.

GETTING REAL

continued from page 22

virtual machine. Each session process has two caches in which to access objects, in addition to the shared page cache. One cache, called the temporary object cache, is where new objects are created. As the execution of server Smalltalk code causes new objects to be created, they are created in a section of memory carved out just for that purpose. This area of memory is garbage collected by generational scavenging techniques, since many newly created objects die early and can be garbage collected soon after their creation. If this cache should become full, then some objects from it must be written to disk, where garbage collection is more expensive. To determine the appropriate size for the temporary object cache, a system designer must consider the total size of all new objects created during a single transaction.

The other cache utilized by the session is the private page cache. This cache is a private area in which to read and write pages of objects. This cache is usually small, since the session primarily uses the shared page cache to read and write objects. If the system is configured not to allocate a shared page cache on the machine where a particular session is executing, then its private page cache size should be increased accordingly.

A session's process can get a variety of information about itself. To monitor garbage collection activity in the temporary object cache, a session can get the #Time

InScavenges statistic to find out the CPU time spent performing in-memory garbage collection, #NumberOfScavenges to find out the number of times the in-memory garbage collector has been executed, or #NumberOfMakeRoomInOldSpace to find out the number of times the oldest generational space filled up (a large number may indicate that the session's temporary object cache size is too small). A session can also find out how well it is using the shared page cache. It can get the #NumberAttached statistic to find out the number of pages that the process is currently using in the shared page cache, and #LocalPageCacheHits and #LocalPageCacheMisses to find out how many times a page was found or not in either the shared page cache or the private page cache. A session can measure its transaction activity by looking at the #NewObjsCommitted statistic to find out the number of newly created objects committed by the most recent transaction, and #NumberOfCommits and #NumberOfFailedCommits to get a cumulative number of successful or failed transactions since the session began.

The statistics described above are but a sampling of the kinds of information to look at when configuring a multi-user Smalltalk system. The key to successfully configuring and tuning such systems is understanding the multi-process nature of clients and servers, and how different memory spaces and caches are used. Fortunately, tools are available to gather these statistics over long periods of time and then graph the results to analyze overall system performance. Without the ability to gather statistics about each process in system, tuning is a shot in the dark. ■



Jan Steinman



Barbara Yates

Beware the octopus

WE PAID A visit to MegaCorp the other day to see how Aaron Blake's new Smalltalk project was getting on. We didn't have much time with Aaron before a major requirements meeting. There was some small talk about the weather as we walked from reception to the departmental meeting room, then Barbara asked, "So, is everything fine with the project these days?"

"We've been attacked by octopi," Aaron exclaimed, shaking his head and throwing his hands in the air in exasperation. He reacted to our puzzled looks only by saying, "In a few minutes, you'll see what I mean!"

MEGACORPUS CENTRALITIS

MegaCorp MIS Director Andrea Saunders, had just introduced the first speaker, an owlish-looking accounting type. He put a slide on the overhead. (See Fig. 1.)

"...and so in closing, I must stress that it is essential that objectification of the central MegaCorp information function must continue to support the Master Records Processor module and the key report generational subsystems." Several bobbing heads snapped to dazed attention as the lights came on.

"We will certainly keep the needs of Accounting in mind as we proceed with the transition plan," said Andrea, "perhaps Jake can give us an idea on legacy maintenance planning?"

"Accounting is certainly important, but it's only one piece of the picture," Jake began as he placed his opening slide on the overhead projector. (See Fig. 2.)

"I knew I was coming on after Linus, so I highlighted accounting's important part in this by coloring it red," Jake continued.

"Did you know the octopus turns red when it is angry?" Aaron whispered during the presentation. "Linus and Jake

are always battling over what function is more central to MegaCorp. It started a few years ago during down-sizing, but it's continued even through the good times."

"The bottom line is that the MIS Infrastructure's central role in MegaCorp best enables it to be of service to all departments during this period of transition to object-oriented technology." Jake returned to his seat as the lights once again came on.

"Well, if there are no more questions for Jake," Andrea said, "I believe he has someone from his technical staff here to show us some more detail—Jake?"

Jake half-rose from his seat, saying "Yes, Denny Hicks is here today to show us the architecture we're planning to use during transition—take it away, Denny!"

Denny was your typical bright, but introverted, computer nerd. He bashfully shuffled forward with his slides

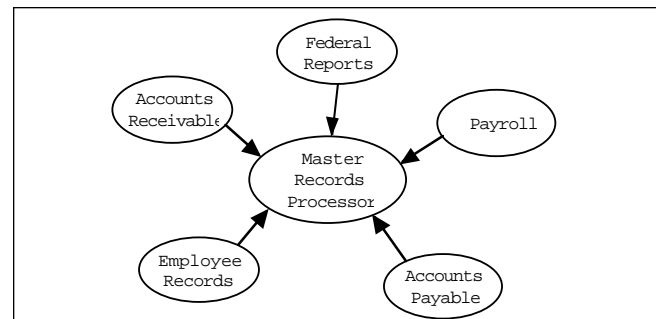


Figure 1. MegaCorp accounting requirements.

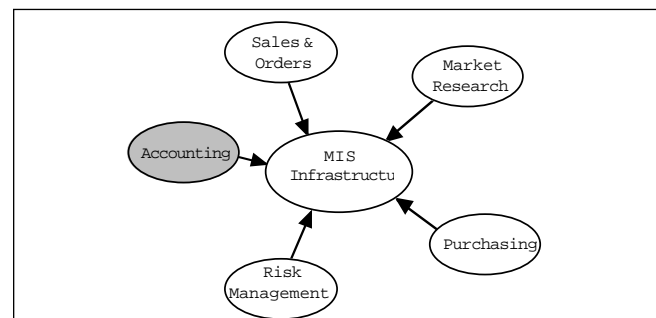


Figure 2. MegaCorp information systems requirements.

Jan Steinman and Barbara Yates are cofounders of Bytesmiths, a technical services company that has been helping companies adopt Smalltalk since 1987. Between them, they have over 20 years Smalltalk experience. They do not bill for hours their cat, Smalltalk, helps them with clients. They can be reached by email at Barbara@Bytesmiths.com or Jan@Bytesmiths.com, or via <http://www.bytesmiths.com>.

in one hand and a trembling coffee mug in the other. Jake helped him put his first slide on the screen. (See Fig. 3.)

"First, entry of a sale creates a sales record object," began Denny with quavering voice, "which CIMPR" (Denny pronounced it "simper") "sends to the weekly receivables report object and marketing report object, and logs it against the salesman's employee record object for commission tracking."

"Look out, dofleini* everywhere!" whispered Aaron.

"I don't know," hissed Trigger Larsen, Aaron's best technical whiz "It looks pretty vulgarus† to me!" Aaron and Trigger shared an interest in marine biology, but unfortunately Trigger did not share Aaron's sense of polite discretion, nor was he always able to leave things on the racquetball court. Denny had trounced him yesterday, and he was still smarting.

"If those are objects, what kind of behavior do they have?" Trigger piped up. Heads turned, Jake scowled, and Denny stammered and trembled, spilling the cafeteria's inky-black coffee on the diagram.

"Looks like our octopus is trying to hide," murmured Trigger, completely without remorse.

"Hush up and listen!" scolded Aaron, "You'll be in the hot-seat in a few minutes, and you'd better be prepared for the issues these people are raising, which are just as valid as they would be if they had a clue about objects!"

Denny actually managed to do quite well now that his slides were ruined. The lights came on, he put his hands in his pockets, stared at the ceiling, and briefly, simply described the transitional system he and Jake had cooked up. The key point was that they had struggled to decouple data from processing. The legacy systems being phased out were mazes of spaghetti, with in-line database calls scattered everywhere.

"Looks good!" said Andrea, as she shot an icy glance at Trigger. "Now Aaron is going to give us a run-down on the end architecture that we're shooting for."

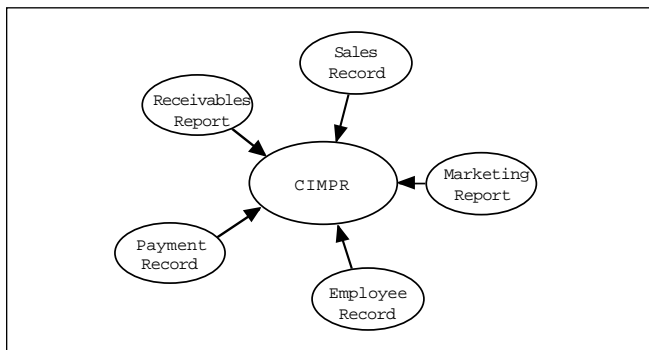


Figure 3. Central information management processing resources.

* *Octopus dofleini* is the giant North Pacific octopus.

† *Octopus vulgarus* is the much smaller common octopus.

A CURE FOR CENTRALITIS

Aaron walked empty-handed to the front of the room and turned off the slide projector. He began in a quiet voice, and side discussions hushed as people strained to listen.

"... are not best served by highly centralized functions, but rather by a society of collaborating objects," Aaron began. "Objects are specialists, and we're going to have to change not only our technology, but our whole way of thinking about our business functions and processes.

Linus, your reports are vital to you and the departments you serve, but we must learn to think of them as transient artifacts—mere side-effects of the communication between objects."

"Jake, you know the PC revolution is turning things up-side down. We will need to distribute data processing for marketing, sales, purchasing, and accounting over a variety of platforms in different departments.

You've already taken the first step, by separating the what from the how, but now we're going to have to put them together again in much smaller pieces, so MegaCorp isn't dependent on a central MIS group. Don't worry about jobs, though— there's going to be plenty of work to do! But we're going to be much more effective at doing it by using Smalltalk for distribution of right-sized components."

"How many salesmen don't have a pretty good idea what their commission is? Denny, wouldn't it be nice if the SalesAgent object made the sale, and told a MarketWatcher object about it, who would go around gathering up similar data from other SalesAgents? Then all you'd have to do is query the MarketWatcher whenever you wanted to, even if the machine that runs CIMPR happens to be down!"

"We're all making good progress here—sometimes it takes many little steps to make a big change. But remember that the weaknesses of centralized processes are what caused us to look at Smalltalk in the first place!"

"Thank you, Aaron!" Andrea smiled. She was glad she put him on last, even though everybody came to the meeting wanting to find out what all this Smalltalk stuff was all about. Leading with the new would have been disaster, because people view the new through old biases, but by letting them voice their concerns early, she was able to assure them that there was a path forward.

The meeting broke up, and Barbara, Jan, Aaron, and Trigger headed over to the nearby Red Coach (affectionately called the "Dead Roach" among MegaCorp cognoscente) for lunch.

"What's the special today," Aaron asked the waitress.

"Calamari," she replied. "No thanks!" we all shouted in unison, then burst out laughing. She looked at us as if we were from Mars, saying "Why don't you take a few more minutes to decide; I'll be right back." ☐

Editors' Corner



John Pugh



Paul White

AFTER THE PARCPLACE-DIGITALK merger, Smalltalkers could be forgiven for looking forward to a period of relative quiet and stability in the industry. Not so! In a move that further demonstrates IBM's commitment to Smalltalk, IBM acquired Object Technology International (OTI). OTI will become an independent subsidiary of IBM Canada.

IBM has been dependent on OTI technology since the introduction of its VisualAge for Smalltalk product. The product uses OTI's Smalltalk and also OTI's Envy technology for configuration management and version control. So is this acquisition good news for Smalltalk? Yes! IBM now owns the technology on which its product depends. Moreover, IBM gains access to one of the leading groups of Smalltalk researchers and innovators in the world. In partnerships with organizations like Siemens, Tektronix, Texas Instruments, and Lockheed, OTI has demonstrated that Smalltalk technology can succeed in domains from banking to embedded systems. We expect that OTI's leadership will be very influential in setting IBM's future Smalltalk strategy. As one attendee at Smalltalk Solutions put it, "Did IBM take over OTI or did Dave Thomas take over IBM?" Either way, Smalltalk is the winner!

Is there a downside to IBM's acquisition? Possibly. Many Parcplace-Digitaltalk customers, whether they use VisualWorks or Visual Smalltalk, depend on OTI's Envy/Developer technology. Although ParcPlace-Digitaltalk has announced its intention to incorporate Team/V into its merged product, many long standing clients will likely not wish to move to Team/V in the short term. Envy support will be a requirement for these folks for some time to come. In the spirit of "competition but collaboration", there seems every indication from IBM that continued Envy support will be forthcoming.

We have just returned from Smalltalk Solutions

'96 in New York City. We will be featuring reviews of the conference from David Carr in this issue and Alan Knight in the next issue, but here are a few personal observations. Attendance was up compared to last year and the conference facilities were much better. Plans are afoot for a European version of Smalltalk Solutions to be held in 1997. The use of Smalltalk is growing throughout many parts of Europe but is particularly strong and mature within the German-speaking countries.

A "Smalltalk-Abend" or birds-of-a-feather session held during OOP '96 in Munich was attended by 320 conference attendees. We'll keep you informed as the plans for Smalltalk Solutions Europe unfold.

The sessions that attracted the biggest attention dealt with subjects like the Web (predict-

ably), distributed object computing, and server-based computing with Smalltalk. Skip McGaughey from IBM kicked off the conference with a keynote describing the opportunities that exist for Smalltalk as we enter the world of electronic commerce. A constant theme throughout the talk was the message that "We shouldn't try to match the competition (Visual Basic, Delphi, PowerBuilder, Java, Forte) feature for feature but rather concentrate as a Smalltalk industry on delivering business value to our customers." The call for a low-end, cheap, less resource-hungry, entry-level Smalltalk was one that resonated throughout the conference. IBM, or at least Skip McGaughey, seemed to have got the message.

Finally, some good news for readers of the *Smalltalk Report*: we will be expanding the number of editorial pages in upcoming issues, which will allow us to include more product reviews, book reviews, and product news in addition to features and columns. At Smalltalk Solutions, we conducted a reader feedback survey—thanks to those of you who contributed. Expect to see some of your ideas appear in future issues.

*Did IBM take over OTI
or did Dave Thomas take
over IBM? Either way,
Smalltalk is the winner!*

Conference overview:

Smalltalk Solutions '96: Progress and new challenges

David Carr

THE SMALLTALK FAITHFUL reconvened in New York this March after a year of dramatic change. Since last year's Smalltalk Solutions, ParcPlace and Digtalk merged, Easel became part of VMark, and OTI became a wholly-owned subsidiary of IBM. Meanwhile, the Web emerged as a potentially significant application development tool, propelling Java into position to challenge Smalltalk. These changes and challenges figured in the keynote talks of IBM's VisualAge Marketing Manager Skip McGaughey and STIC's Executive Director Reed Phillips, while ParcPlace founder Adele Goldberg discussed Smalltalk as a teaching tool for college students and computer professionals.

Smalltalk is gaining worldwide market acceptance as a solution for serious business problems, according to McGaughey. No other technology is as scalable and robust, "But we've got to translate this rich technology into business value."

Global economic commerce is accelerating the pace of change enormously. The trend is most obvious on the World Wide Web, where fortunes are being made overnight. McGaughey feels the value of that opportunity is "almost unimaginable. It's going to radically alter our management, and how we live, how we work, and how we play. Look what happened to Netscape." The explosion of interest in the Web, in turn, has transformed Java into a serious contender. "Java is going after not just market; it's going after mindshare," McGaughey said; Smalltalk needs to keep moving to stay competitive.

There are great opportunities overseas, McGaughey said, citing projects in Turkey, Brazil, China, and the former Soviet Union. Another huge opportunity is a vast number of legacy systems. In one bank alone, IBM found 54,000 date routines that need to be changed. "We estimate there is \$400 billion in code that needs to be rewrit-

ten between now and the year 2,000. Here is the challenge: why not rewrite them in Smalltalk?"

"IBM is absolutely, fundamentally, wholeheartedly committed to Smalltalk," McGaughey asserted. About OTI's acquisition: "We've set it up as a wholly-owned subsidiary, but all the Smalltalk people in IBM now report to (OTI President) Dave Thomas. So maybe it's the other way around—that OTI acquired IBM."

IBM recognizes that it needs more successful partnerships like the one with OTI, which produced IBM Smalltalk. "We want to have 1,000 vendors out there building parts and creating components, and they have to know where we're going. For instance, Object Share is taking out our visual programming environment and putting in their own on

top of IBM Smalltalk. We have to enable our competitors because they provide value."

Smalltalk is going through a period of transition, enjoying unprecedented success at the same time that it is "being attacked from the bottom by Visual Basic, Delphi, and Java," he said. Different strategies are required now that Smalltalk is coming into the mainstream. "We need to send different kinds of messages to make different kinds of sales. Instead of selling competitive advantage, we need to sell, 'This is safe, this will scale.'"

"We're beginning to sell to a different kind of clientele," Reed Phillips agreed. "They don't really like technology, and they don't like taking risks." Those promoting Smalltalk for corporate systems now point to prominent success stories and provide more references, he said.

Still, misperceptions persist. Outsiders see Smalltalk's performance as a major weakness, while only 2% of Smalltalkers agree, he said. And many corporate programmers know so little about the language that it seems too exotic to be a practical choice, "So a lot of people

The study STIC released last year found that companies adopting Smalltalk were more likely to have followed a formal process in choosing a programming language.

decide, 'Why don't we use something less complicated, like C++.'

The study STIC released last year found that companies adopting Smalltalk were more likely to have followed a formal process in choosing a programming language. "If we can get people to do real comparisons, then Smalltalk has a significant advantage," Phillips concluded. "Smalltalk seems to have to fight its way into an organization, but once it's there, it does pretty well." Smalltalk projects also were twice as likely to achieve their expected goals. "The Smalltalk industry has the opportunity to grow and prosper because of the successes that are there. It's a matter of getting the word out," Phillips said.

To Adele Goldberg, the issue is not just teaching Smalltalk, but teaching systems building as opposed to programming. "Too many university computer science curriculums stop at teaching data structures and algorithms," she said. It's not surprising it's so hard to recruit people capable of building extensible, adaptable systems. "The most significant part about a system is that once we start it up, there's a maintenance issue. You want it to run indefinitely." And while people can learn the syntax for programming in Smalltalk in an afternoon, "they don't get the systems building part," Goldberg said.

Her solution is LearningWorks, a modified version of the Smalltalk implementations she used to teach programming to 12-year-olds. Its interface is organized into

a neat binder of several "books" used for system planning, experimentation, and development, and it feeds students the modern Smalltalk class library a little at a time. Using the internet as a medium for distributing this free tool, she plans to have Open University students collaborate on building LearningWorks systems as class projects.

Students can start by experimenting with rehearsal worlds that illustrate key concepts and provide a context for exercises in organizing behaviors and allocating responsibilities, Goldberg said. Businesses could train their employees by having them create LearningWorks books that represent the essence of the company's framework.

Reg Krock of the Ontario manufacturing firm Maksteel was one of the people who approached Goldberg after her talk to express interest in obtaining a copy of LearningWorks. "One reason is that we have a 67-year-old president of our company. I could give that to him, and he would actually play with it."

Computer systems are the only part of the business that Maksteel's president doesn't fully understand, which makes it harder for him to manage, Krock said. "There's always been a language gap between the CEO and the CIO. What I'd like to do is take some of the mystique out of it." ■

David Carr is a freelance writer specializing in the object-oriented programming industry. He can be reached at davecarr@pcnet.com.

GETTING REAL

continued from page 22

virtual machine. Each session process has two caches in which to access objects, in addition to the shared page cache. One cache, called the temporary object cache, is where new objects are created. As the execution of server Smalltalk code causes new objects to be created, they are created in a section of memory carved out just for that purpose. This area of memory is garbage collected by generational scavenging techniques, since many newly created objects die early and can be garbage collected soon after their creation. If this cache should become full, then some objects from it must be written to disk, where garbage collection is more expensive. To determine the appropriate size for the temporary object cache, a system designer must consider the total size of all new objects created during a single transaction.

The other cache utilized by the session is the private page cache. This cache is a private area in which to read and write pages of objects. This cache is usually small, since the session primarily uses the shared page cache to read and write objects. If the system is configured not to allocate a shared page cache on the machine where a particular session is executing, then its private page cache size should be increased accordingly.

A session's process can get a variety of information about itself. To monitor garbage collection activity in the temporary object cache, a session can get the #Time

InScavenges statistic to find out the CPU time spent performing in-memory garbage collection, #NumberOfScavenges to find out the number of times the in-memory garbage collector has been executed, or #NumberOfMakeRoomInOldSpace to find out the number of times the oldest generational space filled up (a large number may indicate that the session's temporary object cache size is too small). A session can also find out how well it is using the shared page cache. It can get the #NumberAttached statistic to find out the number of pages that the process is currently using in the shared page cache, and #LocalPageCacheHits and #LocalPageCacheMisses to find out how many times a page was found or not in either the shared page cache or the private page cache. A session can measure its transaction activity by looking at the #NewObjsCommitted statistic to find out the number of newly created objects committed by the most recent transaction, and #NumberOfCommits and #NumberOfFailedCommits to get a cumulative number of successful or failed transactions since the session began.

The statistics described above are but a sampling of the kinds of information to look at when configuring a multi-user Smalltalk system. The key to successfully configuring and tuning such systems is understanding the multi-process nature of clients and servers, and how different memory spaces and caches are used. Fortunately, tools are available to gather these statistics over long periods of time and then graph the results to analyze overall system performance. Without the ability to gather statistics about each process in system, tuning is a shot in the dark. ■

Product News

VISUALWAVE SERVER

ParcPlace-Digitalk announced the shipment of its VisualWave Server, an internet application server for deploying client/server/web applications, and a corresponding training course. Applications created with the VisualWave Development Environment can be instantly deployed across the enterprise in a client/server environment or to the World Wide Web. The product enables deployment of mission-critical applications to the Web, and reliable management of Web applications including a fully dynamic environment, application management, configuration and performance tuning, session management, and support for standard Web servers. Features include the capability to install or update applications without interruption of service; the capability to maintain session information for a unique user over many page interactions; customizable URLs; and a wide variety of configuration options for optimizing performance.

The VisualWave Server is available for Windows NT, SunOS, Solaris, HP-UX, AIX, and Digital UNIX and is priced at \$4,995. The 3-day training course is available for \$1,275 per person in Sunnyvale, Portland, Chicago, and New York. ParcPlace-Digitalk Inc., 999 E. Arques Ave., Sunnyvale, CA 94086-4593, (v) 800.759.7272 or 408.481.9090, f: 408.481.9095; <http://www.parcplace.com>.

WINDOWBUILDER PRO/V 3.1

Objectshare Systems, Inc. announced WindowBuilder Pro/V 3.1 for VisualSmalltalk 3.1. Highlights of the new fea-

tures include support for Windows 95 controls, unlimited undo/redo, support for model objects, pool dictionary management, a portable window and font system, and much more. WindowBuilder Pro/V 3.1 is now shipping for VisualSmalltalk 3.1 for Windows (3.1, 95, and NT) and OS/2.

Objectshare Systems, Inc., 5 Town and Country Village, Ste. 773, San Jose, CA 95128-2026, 408.970.7280, f: 408.970.7282

CORBA-COMPLIANT SMALLBROKER

IONA Technologies Ltd. and DNS Technologies Inc. announced that IONA will distribute SmalltalkBroker, a Smalltalk-based CORBA 2.0-compliant object request broker and supporting development tools jointly developed by the two companies. SmalltalkBroker provides interoperability between Smalltalk objects and other objects such as C++ and Java.

IONA Technologies, Inc., 55 Fairbanks Blvd., Marlboro, MA 01752, 800.672.4948, f: 508.460.6099, info@iona.com, <http://www.iona.com>

Product News 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 *The Smalltalk Report*, Product Announcements Dept., 885 Meadowlands Drive, #509, Ottawa, ON K2C 3N2, Canada, 613.225.8812 (v), 613.225.5943 (f)

Editors

John Pugh and Paul White
Carleton University & The Object People

SIGS Publications Advisory Board

Tom Atwood, *Object Design*
François Bancelhon, *O₂ Technology*
Grady Booch, *Rational*
George Bosworth, *ParcPlace-Digital*
Jesse Michael Chonoles, *Lockheed Martin ACC*
Stuart Frost, *SELECT Software*
Adele Goldberg, *ParcPlace-Digital*
Thomas Keffer, *Rogue Wave Software*
R. Jordan Kriendler, *IBM Consulting Group*
Thomas Love, *Consultant*
Bertrand Meyer, *ISE*
Meilir Page-Jones, *Wayland Systems*
Cliff Reeves, *IBM*
Bjarne Stroustrup, *AT&T Bell Labs*
Dave Thomas, *Object Technology International*

The Smalltalk Report

Editorial Board

Jim Anderson, *ParcPlace-Digital*
Adele Goldberg, *ParcPlace-Digital*
Reed Phillips
Mike Taylor, *ParcPlace-Digital*
Dave Thomas, *Object Technology International*

Columnists

Jay Almarode, *GemStone Systems Inc.*
Kent Beck, *First Class Software*
Juanita Ewing, *ParcPlace-Digital*
Bob Hinkle, *Consultant*
Tim Howard, *FH Protocol, Inc.*
Ralph E. Johnson, *University of Illinois*
Alan Knight, *The Object People*
Mark Lorenz, *Hatteras Software, Inc.*
Jan Steinman, *Bytesmiths*
Rebecca Wirfs-Brock, *ParcPlace-Digital*
Barbara Yates, *Bytesmiths*

SIGS Publications Group, Inc.

Richard P. Friedman, Founder, President, and CEO
Hal Avery, Group Publisher
John McCormick, Editorial Director

Editorial/Production

Kathleen M. Major, Managing Editor
Elisa Varian, Director of Manufacturing
Seth J. Booke, Assistant Managing Editor
Dan Olawski, Production Editor
Sue Mycka, Desktop Designer and Cover Design
Margaret Conti, Advertising Production Coordinator

Circulation

Elayne Glick, Circulation Director
Lawrence E. Hoffer, Director, New Business Development
Byron Scarlett, Assistant Circulation Manager

Advertising/Marketing

Gary Portie, Advertising Manager, East Coast/Canada/Europe
Elisa Marcus, Advertising Manager, Central US
Michael W. Peck, Advertising Representative
Kristine Viksnis, West Coast Exhibit Sales
Sarah Olszewski, East Coast Exhibit Sales
212.242.7447 (v), 212.242.7574 (f)
Diane Fuller & Associates, Sales Representative, West Coast
408.255.2991 (v), 408.255.2992 (f)
Nancy Beuschel, Promotions Manager for Magazines

Administration

Margherita R. Monck, General Manager
David Chatterpaul, Senior Accounting Manager
Bibi Budhram, Accounts Payable



PUBLISHERS OF JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, THE SMALLTALK REPORT, THE X JOURNAL, JAVA REPORT, OBJECT CURRENTS (ONLINE), THE X SPOT (ONLINE), OBJECT EXPERT (UK), and OBJEKTSPKTRUM (GERMANY)

Features

Smalltalk SQA: What to test?

4

Jeff McKenna

The need to develop processes and tools for software without sacrificing high productivity is discussed.

A strategy for using instance variables

7

Bobby Woolf

Bobby presents some guidelines for using instance variables cleanly and effectively. This helps improve commonly written code for initialization, accessing, equality, persistence, and application layers.

Controlling image size when using GemStone

12

John Bentley

Controlling image size is important. These various replication schemes can aid in performance optimization.

Tactical patterns for the real world:

18

Optimization patterns

Darrow Kirkpatrick

The third and final article in the series on patterns for working with domain models presents patterns for dealing with optimization issues—handling Smalltalk objects that must perform well, while incorporating extra levels of indirection to be persistent or transient.

Columns



Managing Objects

20

Documents on the Web

Jan Steinman & Barbara Yates

The Web can be a powerful communication tool, but like all tools, it can be misused. Steinman & Yates demonstrate techniques

for offline HTML from your "hot" Smalltalk documentation.



The Best of comp.lang.Smalltalk

24

Smalltalk Solutions

Alan Knight

Knight reviews the recent Smalltalk Solutions Conference and his impressions of the Smalltalk industry.



Getting Real

27

Multi-user canonicalization

Jay Almarode

Almarode demonstrates one approach for solving the problem of canonicalization of objects using multi-user Smalltalk.

Departments

Editors' Corner

2

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar-Apr, July-Aug, and Nov-Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1996 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Bulk rate U.S. postage paid Lancaster, PA, permit 161. Canada Post International Publications Mail Product Sales Agreement No. 290386.

Individual Subscription rates 1 year (9 issues): domestic \$89; Mexico and Canada \$114, Foreign \$129; Institutional/Library rates: domestic \$199, Canada & Mexico \$224, Foreign \$239. To submit articles, please send electronic files on disk to the Editors at 885 Meadowlands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

POSTMASTER: Send domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sigs.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

Smalltalk SQA: What to test?

Jeff McKenna

AS DEVELOPERS USE Smalltalk more frequently to develop production systems, the quality of Smalltalk-based applications has become an important issue. This is the first of a series of articles on Smalltalk SQA, the goal of which is to discuss the issues of SQA in the Smalltalk development environment. Each article will focus on a specific aspect of Smalltalk SQA. Note that Software Quality Assurance in Smalltalk is not a solved problem. This series hopes to contribute to the solution.

Many of the points to be made in these articles apply to software developed using any development environment. However, the use of Smalltalk makes the impact of not “taking care of business” more visible because of the high development productivity. We need to develop processes and tools to help us develop robust software without sacrificing high productivity.

THE TOP QUESTION

When considering SQA, the first question we ask is this: *What do we test?* The first answer is easy: *The software.* This answer is insufficient. For any real project there will be insufficient time or resources to *completely* test the software. This means that we must decide how to deploy those resources most effectively.

For applications with a GUI, the answer seems easy: *Test the interface.* The need for such testing is real and the success of GUI testing tools attest to that reality. The GUI testing tool vendors are working to make sure that their tools work with the dialects of Smalltalk. Many projects are using GUI testing as their sole method of testing.

This reminds me of quality control activity in the automotive industry in the 50s and 60s. TV ads from that time show hordes of white-coated, quality technicians poring over a car fresh off the assembly line. The technicians assured us that they would find every defect before it reached us.

Unfortunately, those technicians did not find enough of the defects. The automotive industry went through a very traumatic time over quality. The industry finally came to the same conclusion that other industries have. In short, it is *not* possible to insure sufficient quality with *only* inspections at the end of the manufacturing process.

This is exactly what we are doing when we test only by using GUI testing tools.

In our experience, the GUI is the least stable part of an application. Typically, it is the last part finished. If we are to have the SQA work proceed with speed that we have come to

expect in development, we must do more work in parallel. Basing all testing on the GUI means that little testing can occur until near the end of a development cycle. Since the GUI may change as well, GUI based tests will tend to be unstable.

GUI-based testing is necessary but it is not sufficient. If it was, software would be of much higher quality than we experience. In addition, when the software is “headless,” GUI-based testing is not even possible. I am not saying here that GUI-based testing should not be done. If my resource constraints are so severe to limit my testing technique to a single one then GUI testing would be it.

The automotive industry solved its quality problems by utilizing a number of techniques. One common thread of these techniques is that they maintain quality at each stage of production. The automotive worker can stop the assembly line if cars of “bad” quality are being produced. Component vendors ship components of top quality, that is, no defects. We hear the result of these changes in the language of the advertising: “Quality is Job #1” or “Quality is built in.”

“What to test?” is now a question that has a clearer answer. We must test the final product and the components that make up that product. We must build the quality in.

*We must build
the quality in.*

COMPONENTS

Before we can discuss how to build quality in, we must first establish our definition of a component. In automotive manufacturing, example components are the screw, the bolt, the starter and the transmission. In transmission manufacturing, example components are the gear, the switch, the screw and the bolt. Higher "level" components contain lower level components. Different higher level components often share the same lower level components. Both the starter and the transmission may use 1/4 6-32 screws and bolts. We see in the automotive industry two of the component constructs we find in our industry: Encapsulation and Aggregation. A bill of materials illustrates the use of these constructs.

Initially most of us felt that the class was the lowest level component in Smalltalk, the lowest level of reuse. We now know that the class is not typically the lowest level component. The lowest component is a combination of classes and methods with a clear purpose and intent. An example of multiclass capsulation into a component is the Graphics Component of many textbooks:

- Graphic
 - Ellipse
 - Circle
 - Rectangle
 - Square
 - Line

Taken together, these classes form a logical grouping. If the class Graphic implements a method `Graphic>>isGraphic`, we should also define `Object>>isGraphic`. If we do so, the `Object` method is part of the Graphics Component.

The major code control systems used with Smalltalk today acknowledge the levels of components under different names. Team/V calls low level components packages and higher level components clusters. ENVY does not map directly in this way. It calls low level components applications or subapplications depending on the reuse strategy and the high level components configurations or applications.

My view is that these are all components and that a single construct is sufficient. Low-level components just do not contain any subcomponents. ENVY could drop the distinction between application and subapplication while Team/V could merge cluster and packages.

The decomposition of an application into components is not an easy task and there is no right answer. Having too many is confusing and costly and having too few inhibits parallel development. The criteria to be considered in defining a component are very similar to those used in defining classes. Does the component do one thing? Is it maintainable by one person? Is it cohesive? Does it have a narrow interface?

SQA uses a component differently than development uses that component. We suggest that the component is the unit of acceptance and rejection. This moves SQA activities into the production process. These activities must determine the quality of each component.

SOFTWARE COMPONENT

We define a software component as code encapsulated with documentation and tests. The key concept is that the code itself is not verifiable or reusable without its documentation and its tests.

If this sounds very old, it is. As an industry we have *known* this for a long time. We have not acted on that knowledge. The question in my mind is how to build process and structure that encourages the practice. If both documentation and tests are an active part of the development then the practice is encouraged. In particular, components are reusable only when they include documentation and tests.

From the point of view of testing, the primary goal of documenting the software component is to present the public interface. This should include message definitions including the returned objects, required message sequences and error conditions. A major SQA task is to determine if the component tests cover the component public interface. The vehicle for this is the documentation.

My experience is that the process of *writing* the documentation improves the code, *if* the developer corrects errors, renames methods and fleshes out behavior as part of the documentation effort.

At a minimum, component tests must exercise the component public interface (verification). More complete testing will do a number of things: (1) test to determine if all code is executed (coverage); (2) test to determine if reaching limits does not fail (stress); (3) test the error handling (failure analysis); or (4) test the internal details (implementation). A primary SQA task is to make sure that the component tests are sufficient to determine if the component meets the project quality standards.

In the past, we often found the developers writing the lowest level tests, unit tests, and leave the fleshing out of tests to SQA, system tests. With Smalltalk that divide should be less clear. I view the SQA tasks as work to be done to delivery a quality software component. Who exactly performs those tasks is not too important.

My experience is when developers write and use tests, their code improves, exactly the same as when they write documentation. Testing will uncover errors. The availability of tests also makes incremental development work better. Developers will know when their rework is complete.

The automotive worker now knows that quality is an issue for him or her to address directly. Quality is built in. Development of software in Smalltalk needs to do the same.

In the next article, I will take up the public/private problem and discuss regression testing. ☞

Jeff McKenna is Founder and President of MCG Software, Inc. of Wilsonville, OR. MCG Software offers testing frameworks for Smalltalk. Jeff has been involved with software for more than 33 years and been involved with Smalltalk since 1982. Jeff was chairman of OOPSLA '94. He may be reached at mckenna@acm.org.

A strategy for using instance variables

Bobby Woolf

I HAVE BEEN DEVELOPING a strategy for using instance variables that you might find helpful. This strategy provides guidance for several common programming tasks, such as properly initializing instance variables and providing accessors to use them. It shows how to implement equality methods and helps guide the initial decisions in making an object persistent. Finally, it explains why the instance variables in various application layers tend to behave differently.

Although this strategy probably isn't perfect, it is one that I find useful. The strategy doesn't consist of hard and fast rules you should always obey, just suggestions you should consider and trends you can look for. I can't guarantee that following these guidelines will make you a better programmer, but they should help.

TYPES OF INSTANCE VARIABLES

I've noticed that not all instance variables are created equal. Some seem to be more important than others. When using instances of a particular class, I notice that I'm constantly inspecting certain instance variables to make sure their values look reasonable, yet I consistently ignore other instance variables. So I've been trying to figure out how to distinguish the important ones from the unimportant ones.

In looking at how I use instance variables, I've found that there are three types, which I call identity, status, and cache. When looking at a new class, I try to distinguish these types to help figure out how the class works. When one of my own classes doesn't work well, I look at how I'm using these types; often I find inconsistencies; when I clean those up, the class works better. As I help other people develop their classes, I look for these types. If possible, I encourage the developers to identify each instance variable's type and use it "correctly."

I describe the three types in the following subsections.

Identity variables

Identity variables are how you distinguish two instances

of a class. If both objects have the same identity values, they represent the same entity. Once an identity value is set, it usually doesn't change. After all, if you recognize an object because it has a certain identifier, and that identifier changes, how will you recognize it again next time? An object's identity values must be set for the objects' state to be valid. Also, there are usually no good default values for identity variables. Multiple objects with the same default values would be indistinguishable. Examples of typical identity variables include `uniqueID`, `name`, and a tree node's parent.

Status variables

When developers talk about instance variables—the variables that maintain an object's state and are accessed through getter and setter methods—they're usually talking about what I call "status variables." *Status variables* maintain an object's internal state and its relationships to other objects. These relationships may be aggregate or associative. Whereas identity values don't change, status values change constantly to reflect the object's changing state. Like identity variables, status variables must be set in order for the object's state to be valid; otherwise its internal state is undefined and inconsistent. If a status value is lost (set to an invalid value such as `nil`), the object's state cannot be recovered. Finally, status variables have suitable default values. (If nothing else, `nil` can be used as the default value, but that's often not a very good one. See my previous discussion on the Null Object pattern.¹) Taken together, these default values describe the object's initial state. Examples of status variables include `address`, `employer`, and a tree node's children, as well as the various settings represented on a GUI using check boxes, radio buttons, etc.

Cache variables

Cache variables cache the results of expensive calculations. Their values are derived from the values of identity and status variables. When those values change, the

cache values must be recalculated. So cache values change as frequently as the values they are based on change. Cache values are optional; the object's state is still valid without them. If a cache value is lost, it can easily be recalculated. A cache variable's default value is usually *uncalculated*, a flag indicating that the value hasn't been calculated yet. The most common flag for uncalculated is nil, but there can be other such flags. For an example of a cache variable in VisualWorks, see `CompositePart>>preferredBounds`. A composite calculates its preferred bounds by merging those of its components; it caches the result for efficiency.

RAMIFICATIONS

These definitions are comforting, but they alone don't make your code any better. Yet you can improve your code by recognizing these types and writing your code accordingly.

Initialization

There are three approaches to initialize a variable:

1. Let a collaborator set its value explicitly.
2. Set its value to a default constant.
3. Set its value to the result of a calculation.

Each of these approaches is used to initialize a different type of instance variable:

1. *Identity initialization*—Initializes the identity variables.
2. *Creation initialization*—Initializes the status variables.
3. *Lazy initialization*—Initializes the cache variables.

Identity variables are initialized by the collaborator which creates the object. The collaborator should accomplish this via an instance creation method on the class side. Two examples of instance creation methods in VisualWorks—besides the standard ones like `new`, `basicNew`, and `new:—`are `Point class>>x:y:` and `Dependent Partclass>>model:`. An instance creation method on the class side should be implemented via a corresponding identity initialization method on the instance side. For example, `Point class>>x:y:` uses the identity initialization method `Point>>setX:setY:` to create the new instance:

```
Point class>>x: xInteger y: yInteger
  ^self basicNew setX: xInteger setY: yInteger
```

```
Point>>setX: xPoint setY: yPoint
  x := xPoint.
  y := yPoint
```

The instance creation methods in `Circle` and `Interval` are implemented the same way. I prefer to name this identity initialization method `init...`, so the name I would have used for `Point>>setX:setY:` would have been `initX:y:`. I put these methods in the “initialize-release” protocol.

Status variables should be initialized to their default values when the new instance is created. The standard name for the method that performs creation initialization is “initialize”. VisualWorks has tons of examples of

this, such as `SortedCollection>>initialize`. Another example is `OrderedCollection>>setIndices`; it isn't called “initialize” but it should be because it serves the same purpose.

Cache variables do not need to be initialized until they are used. In fact, initializing them is usually expensive and should be avoided until you know the values are needed. The easiest way to do this is to build lazy initialization into their accessors. VisualWorks doesn't use this technique much, but two examples are `CompositePart>>preferredBounds` and `SliderView>>marker`. You might implement `Circle` with `radius` as an identity variable and `diameter` and `area` as cache variables:

```
Circle>>radius
  ^radius

Circle>>diameter
  diameter isNil ifTrue: [self computeProperties].
  ^diameter

Circle>>area
  area isNil ifTrue: [self computeProperties].
  ^area

Circle>>computeProperties
  | r |
  diameter := radius * 2.
  r := self radius asLimitedPrecisionReal.
  area := r class pi * r * r
```

Developers often use lazy initialization with variables that are not caches, but I avoid this. Although caches are expensive to initialize, other variables usually aren't, so I see no compelling advantage in using lazy initialization on those other variables.

Often status variables are initialized in terms of identity variables, which means that an identity initialization method (in the form of `initA:b:...z:`) has to be run before the creation initialization method. Here's a hypothetical example of an instance creation method that will do this:

```
Example class>>x: newX y: newY
  ^self basicNew initX: newX y: newY initialize
```

`HelpBrowser class>>on:` is implemented this way because `HelpBrowser>>initialize` ends-up using the value of `on:`'s parameter.

Accessing

Developers often automatically create getter and setter methods for all of their instance variables and put them in a public protocol like “accessing.” I prefer to be a little more selective and only create accessors for certain types of instance variables.

Identity variables need getters but no setters. The getters may be public or private. Setters are usually not necessary because the identity variables' values typically don't change. The only “setter” that is required is the identity initialization method (`initA:b:...z:`). Any setters you do

provide should definitely be private. Status variables use getters and setters in the conventional manner. These methods can be public or private.

Cache variables have getters but no setters. The getters, which can be public or private, contain lazy initialization. I prefer to implement the lazy initialization via a compute... method, as shown earlier in Circle>>computeProperties. If the calculations for one cache variable calculate others in the process, group the initialization for all of those variables together in one compute... method. Don't implement setters; they could be used to set the caches to values that are inconsistent with the object's state. Instead of setters, I implement flush...

methods which reset the variables back to their uncalculated state (usually nil). If one change invalidates a number of caches, I flush them all in one method.

For example, let's say that the Circle described earlier caches both diameter and area and that radius can change. Some more of the code would be

```
Circle>>radius: newRadius
    radius := newRadius.
    self flushProperties
```

```
Circle>>flushProperties
    diameter := nil.
    area := nil
```

The compute... and flush... methods are private ones. The cache getter methods with the lazy initialization send the compute... methods (see Circle>>diameter). The setter methods for the status (and identity) variables send the flush... methods (like Circle>>radius:). A particular setter does not need to flush all of the object's cache variables, only the ones that were calculated from it.

Equality versus identity

In my previous article, I talked about the difference between object identity and object equality. Object identity is very clear cut. If two variables contain identical objects, they are double-equal, which means that they both point to the same address in memory. Thus the two variables actually contain the same object.

Object equality is not so straightforward. If two variables' values are equal but not identical, they contain separate objects that are equivalent. The question is: What makes objects equivalent? In theory, they represent the same value. In practice, for Smalltalk, it means that a Set considers them to be duplicates.

I contend that two objects are duplicates if their identity variables are equal; their status and cache values are irrelevant. Because identity values rarely/never change, this means that two objects that are sometimes equal are always

equal. Changes in their status don't affect their equality. Thus if one object is a duplicate of another, it will be so through its entire lifetime, which is how it should be.

Just as implementors of equal (=) use identity variables, so do implementors of hash. If two objects are equal, their hash values need to be the same. So the same variables which are used for determining equality are also used for calculating hash values.

Persistence

When an object needs to store itself persistently, it shouldn't necessarily store all of its instance variable values the same way. Some instance variable types are persistent, others are not.

When storing an object in a relational database, its identity values belong in the database table's key columns. Just as identity variables should uniquely identify an object, a row's key column values should be unique from other rows. Status variables that represent state have simple values that are stored directly in table columns. Those maintaining relationships to other objects become database joins. There is generally no need to store cache values persistently. Rather than consume database space, just recalculate them after reading the object out of the database.

The storage issues for an object database are similar to those of a relational one. An object's identity values serve as its keys for retrieving it from the database. Status values are simply stored with the object. And cache values do not need to be stored at all, although they can be for completeness.

Database proxies also make use of instance variable types. A proxy must contain the identity values for its real object. That way it will be able to load the real object out of the database. Because a proxy is supposed to be lightweight, it shouldn't contain status or cache variables. Ideally, as much of the proxy's behavior as possible will be implemented just using the identity values. This will help maximize the amount of work the proxy can perform and minimize the number of real objects that need to be read from the database.

Dictionaries, Smalltalk objects that act somewhat like simple databases, also make use of instance variable types. Each element is stored in a Dictionary by a key that must be unique. That key is often an identity variable. That variable's value must not change while the element is stored in the Dictionary. Thus an identity variable makes a much better key than a status variable does.

Application layering

A Smalltalk program contains four main layers: view, application (mediator), domain, and infrastructure.² Most of the variables in application models and view objects are status variables. Identity variables are concentrated in domain objects. Infrastructure objects tend not to contain much state at all; they mostly point to domain

I see no compelling advantage in using lazy initialization.

objects in some way (which can be an identity or status relationship).

Exceptions

These guidelines are not rules that are engraved in stone. Identity values *can* change during an object's lifetime. It's *sometimes* helpful for an instance creation method to initialize some status variables. A proxy *may* want to contain certain status values because they're used so often. However, I try to stick to these guidelines when possible. When I make an exception, I like to have a good reason.

Here are some interesting exceptions to these guidelines that I've found in VisualWorks.

Set's tally variable—Its behavior is a cache. If its value were ever lost, it could easily be recalculated. However, it's implemented as a status variable. That is because its value only changes by ± 1 each time, a simple and well-defined transformation on the old value. For a large Set, it is much easier to add or subtract 1 than to flush the value and recalculate it from scratch.

Model's dependents variable—Its behavior is a typical status variable. However, when storing a Model persistently, this variable must be treated specially. Dependents are usually transient and thus are not stored when their parent is.

Point's x and y variables—Are these identity variables or status? Once a Point is created, can its x and y values change? Generally, changing their values is a bad idea, but there are plenty of examples where it works just fine. The same goes for the instance variables in Rectangle, Circle, Date, etc.

OBJECTIONS

As I discuss these ideas with other developers, I hear certain objections repeatedly. Here are some of them and my replies:

"Initialize is expensive"—Not if it's used properly. I use it to initialize status variables, ones which have readily available default values. If an implementor of initialize is expensive, it's probably doing more than just initialization. Which leads to...

"This status variable is expensive to initialize"—Then it's a cache variable. Cache variables require calculation to initialize; that's why they're lazy initialized. Status variables are initialized with simple default values that need no calculation.

"This status variable is hardly ever used"—Then get it out of that object! Every time you instantiate an instance of that class, you're sucking up memory for variables that probably won't be used. If there are a number of these variables, you're wasting a lot of memory. Refactor the class into two or more classes that separate the variables that are usually used from those that usually aren't. By the way, each of the pointers to these optional separate objects is a status variable, but it can be implemented as a cache.

"Lazy initialization is more efficient"—Not for identity and status variables. They're only initialized once. Why

have the getters check every time to make sure they're initialized? They already have been. Lazy initialization is fine for cache variables because they get flushed periodically. But for identity and status variables, you always use them, so initialize them once and get it over with.

A WELL-DESIGNED OBJECT

Let's take a look at how you would use these guidelines to design a class. First of all, we assume that the class' implementation requires a number of instance variables.


- Some of their values are computed from the values of others. These are cache variables.
- Some are required as part of the object's state and have suitable default values. These are status variables.
- Some others are also required but do not have good default values. The object's collaborators must set these values when they create the object. These are identity variables.

Once you've established these designations for your variables, follow the other guidelines to help implement the class properly. The identity values should not change. They should be used in implementors of equals and hash and as database keys. The cache variables should have lazy getters as well as flush and compute methods. The status variables should be used to maintain the object's current state.

CONCLUSIONS

Here are the main points in this article:

- There are three types of instance variables: identity, status, and cache.
- Identity values don't change, status do, and cache are calculated from identity and status.
- Each type is initialized differently: identity initialization from collaborators, creation initialization, and lazy initialization.
- Identity variables are used for =, hash, and as dictionary and database keys.
- Status variables store an object's state and relationships to other objects.
- Cache variables require flush and compute methods.
- These are guidelines only; there are exceptions.

In my next article, I'll talk about how to display an object as a String. It turns out that identity variables are very helpful for doing this. 

References

1. Woolf, B. "A Hierarchy that Acts Like a Class," *The Smalltalk Report* 5(4), Jan. 1996: 4-10.
2. Brown, K. "Remembrance of things past: Layered architectures in Smalltalk applications." *The Smalltalk Report* 4(9), July-Aug. 1995: 4-7.

Bobby Woolf is a Member of Technical Staff at Knowledge Systems Corp. in Cary, NC. He mentors Smalltalk developers in the use of VisualWorks, ENVY, and Design Patterns. Comments are welcome at woolf@acm.org, or at <http://www.ksscary.com>.

Controlling image size when using GemStone

John Bentley

IN MOST SMALLTALK applications, controlling the image size is an important goal. Larger images require more memory for execution, which can degrade performance. When using an object-oriented database (ODB), controlling the image size can be challenging. This challenge comes from the manner in which objects are stored and retrieved in an ODB.

One of the main advantages cited for using an ODB is the ODB's capability to preserve the object relationship web. Each object is stored with its relationship web intact. When an object is retrieved from the ODB, the related objects are also retrieved without having to manually reconstruct the relationship web (see Fig. 1).

In contrast, in a relational database paradigm when an object needs to be retrieved the developer must reconstruct the object being retrieved from a table. Then, the developer must also find and reconstruct the related objects, and finally, re-establish the relationship web. So, when considering the cost of manually rebuilding the object web, it is easy to see that using an object-oriented database can save a great deal of effort.

When retrieving an object from an ODB, related objects are retrieved automatically. The application developer no longer has to know which related objects are needed. This also means that the application developer

does not necessarily know how many objects are being retrieved. This could be a problem. For example, if the application's equivalent of the Smalltalk dictionary was retrieved, then all objects in the database would be copied into the image. It would be possible to retrieve most, if not all, of the objects in the database. This can be a real problem as the database usually holds more objects than an individual Smalltalk image can handle.

To prevent overloading the image with objects it is important to understand the mechanisms provided for the retrieval of related objects. While preventing the extreme case is vital, it is not the only reason. Another reason to control the retrieval of related objects is to keep the image size minimal. It is possible to create an application that does not retrieve any related objects until they are accessed. This set up would keep the image size small but could also make the application's performance bound to database access. It is important to retrieve enough related objects to perform the task at hand. The control mechanisms for retrieval gives the application developer the ability to balance image size versus database access.

Each ODB vendor provides different mechanisms for controlling object retrieval. For this reason, it is easiest to discuss the control mechanisms in the context of a particular object-oriented database implementation. In this case, the database used as an example is GemStone, from GemStone Systems, Inc. The Smalltalk listings are in reference to GemStone 4.0 and VisualWorks 2.0.

All listings refer to a typical employee payroll system. The payroll systems' object model is shown in Figure 2. The application calculates employees' salaries and addresses envelopes for distribution. The application also provides estimates at the company level for weekly and yearly payroll. The purpose of this example is to illustrate concepts and is not meant to be representative of typical ODB applications.

REPLICATION CONTROL

When an object is retrieved from GemStone, the object is "replicated" in the Smalltalk image. The replicated object, referred to as a replicate, is a copy of the GemStone object with a "link" to its database counterpart. The replicate

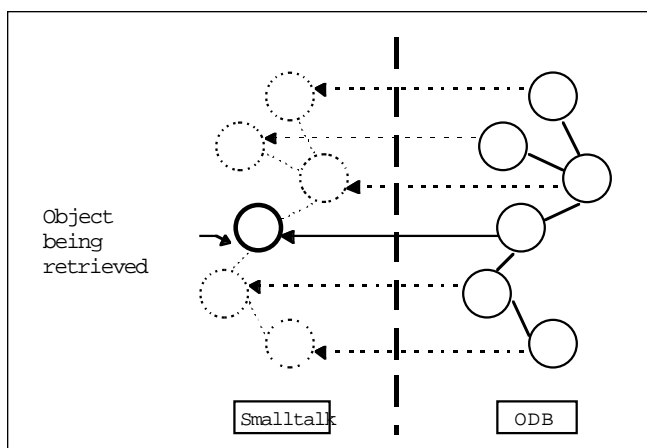


Figure 1. When using an OODB, related objects are automatically retrieved.

maintains the link to its database counterpart so that updates may occur between the database and the image. In GemStone, this controlling of the retrieval of related objects is referred to as replication control.

The GemStone Smalltalk Interface (GSI) provides two techniques to control replication. One of these techniques is "replication tuning." Replication tuning allows the developer to control how objects and their relations are replicated in the image. The other technique is avoiding replication. In most cases, objects are replicated into a Smalltalk image to perform behavior. GemStone is an active database, allowing behavior to be defined and executed in the database. The behavior is defined using the GemStone Smalltalk dialect which is called Smalltalk DB. Using behavior defined in Smalltalk DB, database objects can respond to messages without being replicated into the client image thus eliminating the need for replication.

REPLICATION TUNING

There are three common approaches to tune replication using GemStone. The first approach is to change the GemStone to Smalltalk-replication level. This defines the level of relationship replication for the image. The second approach is class mapping. This provides a variety of ways to control how GemStone and Smalltalk instance variables relate. The final approach is to tune replication by changing the "no stub level." This controls relation replication levels on database updates.

When an object is initially retrieved from the database, it is represented by a *proxy*. A proxy is merely a Smalltalk object that references a database object. To replicate the object in Smalltalk, the message `#asLocalObject` is sent to the proxy. The result is a Smalltalk duplicate of the database object that maintains a reference to the database object. Listing 1 shows how the global `MyCompany` would be retrieved and replicated.

When the proxy replicates itself, it checks to see how many levels of relationships need to be replicated. This is known as the GemStone to Smalltalk replication level. If the level is two, the proxy replicates itself and the objects

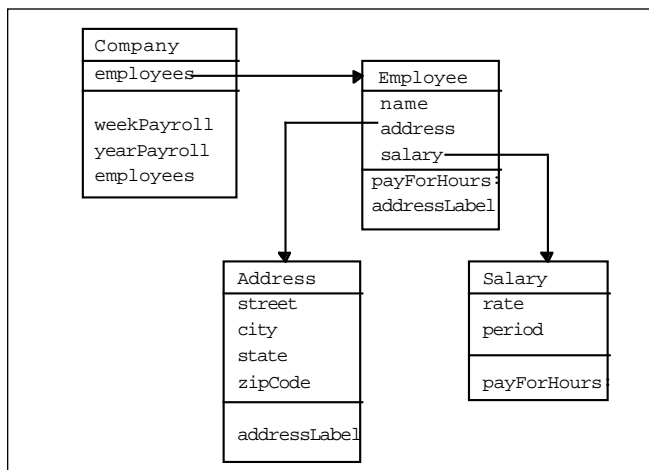


Figure 2. The payroll tracking system's object model.

referenced by the object's instance variables. If the level is three, the objects referenced by the instance variables of the objects referenced by the primary object's instance variables are replicated as well. If the level is zero, all related objects are replicated. Any related objects not replicated are represented by stubs. A stub is a stand-in object that replicates itself when sent a message.

The GemStone to Smalltalk replication level is defined in an instance method in the `GSSession` class. The `GSSession` class defines behavior associated with a database session. The replication level is defined in the `#defaultGStoSTLevel` method. `GSSession` sets the replication level for the entire application and sets the default replication level for the application.

Defining the replication level in a method provides flexibility. For example, in one application, it might be appropriate for `#defaultGStoSTLevel` to unconditionally answer three. In another application, it might be appropriate to provide a different level depending on the platform being used. The `#defaultGStoSTLevel` could be defined as is shown in Listing 2.

In addition, there are special cases where a more spe-

Listing 1. Retrieving and replicating.

```

| proxy replicate |
"Retrieve the GemStone global MyCompany."
proxy := GSI currentSession at: #MyCompany.
"proxy references the GemStone global MyCompany."

"Replicate MyCompany."
replicate := proxy asLocalObject.
"replicate is a copy of the GemStone global
MyCompany.
It maintains a link to its GemStone counterpart."

```

Listing 2. Platform-dependent replication level.

```

"The following methods allow the default replication
level to be varied on a platform basis."

"Methods for GSSession."

defaultGStoSTLevel
"Answer the appropriate level for the current
platform."
^self platformLevels at: CurrentPlatform

platformLevels
"Answer the platform level dictionary. If unset,
initialize."
(platform is Nil)
ifTrue: [ (platform := (Dictionary new))
add: #windows->3;
add: #macintosh->2;
add: #unix->5.
].
^platform

```

cific replication level is required. In these cases, `#asLocalObjectToLevel:` is used instead of `#asLocalObject`. The `#asLocalObjectToLevel:` is sent to a proxy to replicate the object to the level specified as the argument.

In Listing 1, `MyCompany` was replicated using the default level. If the default replication level was three then `MyCompany`, its `employees` collection and all contained employees would be replicated. However, if all of the contained employees are not needed, it might be better to only replicated two levels, as is shown in Listing 3. With the replication level set to 2, each employee object would not be replicated until accessed.

Thus far replication has been tuned at the instance basis. Replication can also be tuned on a class basis. This is done using the second replication tuning mechanism, class mapping. Class mapping allows `Smalltalk` and `GemStone` classes to have different structures. In the general case, `Smalltalk` and `GemStone` classes will have identi-

Listing 3. Replicating to a specific level.

```
| proxy replicate |
proxy := GSI currentSession at: #MyCompany.

replicate := proxy asLocalObjectToLevel: 2.
"This bypasses the default level and only replicates
MyCompany and the collection. Employees are not
replicated until accessed."
```

Listing 4. Custom mapping employee.

```
In GemStone

Object subclass: Employee
  instVarNames: #( 'name' 'address' 'salary' )
  classVars: #()
  poolDictionaries: #()
  inDictionary: Payroll
  constraints: #[]
  instancesInvariant: false
  isModifiable: false

In Smalltalk:

Object subclass: #Employee
  instanceVariableNames: 'name address salary'
  classVariableNames: '
  poolDictionaries: "
  category: 'Payroll'

stValues: anArray
"anArray contains four proxies for the GemStone
instance variables.
Note that only the first three, name, address, and
salary, are referenced."
  name := anArray at: 1.
  address := anArray at: 2.
  salary := anArray at: 3.
```

cal structure. In the example application, the `Employee` class has three instance variables: `name`, `address`, and `salary`. Both `GemStone` and `Smalltalk` would have `Employee` classes with these same three instance variables defined.

In some cases, there may be an instance variable in the `GemStone` class that should not appear in the `Smalltalk` class, or vice versa. For example, the `GemStone` `Employee` class might have an instance variable called `allEmployees` which references the collection of all employees in the system. This means when a single instance of `Employee` is replicated, every employee in the database would also be replicated. To prevent `allEmployees` from being replicated in the image, the `Smalltalk` class would not have the `allEmployees` instance variable. In this case, the `Smalltalk` class's instance variables would need to be mapped such that `allEmployees` was not replicated into `Smalltalk`.

The GSI adds methods to the class `Object` to handle class mappings. The methods, `#stValues` and `#stValues:`, handle the mapping of instance variables between `Smalltalk` and `GemStone`. `#stValues` is used when an object is being stored in `GemStone`. `#stValues` stores the instance variables into an array and answers that array. `#stValues:` is used when replicating an object being retrieved from `GemStone`. It takes an argument of an `Array` and copies this array's values into the instance variables. Both of these methods directly map the instance variables to an array, such that the instance variable stored at `n` is stored in the array at `n`.

Whenever a class' instance variables are defined differently in `Smalltalk` and `GemStone`, the `#stValues` and `#stValues:` methods must be redefined in the `Smalltalk` class to properly map the instance variables. For example, to remove `allEmployees` from the `Smalltalk` `Employee` class, the `#stValues:` would need to map `name`, `address`, and `salary` from the array, but not acknowledge `allEmployees`. The `Smalltalk` code for this is shown in Listing 4.

The final method of replication control addresses the case in which a replicated object is being updated to reflect changes made in its `GemStone` counterpart. When the database object is changed, the associated `Smalltalk` replicate object is "stubbed." The replicate object is then transformed into a stub instead of being reloaded. This "stubbing" provides a performance enhancement that allows *laissez-faire* loading of updated objects.

The behavior of a stub object is to replicate itself when accessed. The database object is replicated in its place. When replicated, it also replicates its related objects. Replication is still controlled using the `#defaultGStoSTLevel` defined in `GSSession`.

There are certain circumstances in which stubs are inappropriate. In this context, the most interesting circumstance deals with performance. If an object is stubbed, it will be replicated using the default replication level for the application. But, if an object was initially replicated using a custom level, this default behavior is undesirable. In this case, it is better to prevent the object from being stubbed.

Stubbing is controlled by the `#noStubLevel` method defined by `GemStone` in the class `Object`. The "no stub"

level defines the levels at which stubbing is not allowed. A “no stub” level of zero, the default, indicates that the receiver can be stubbed. A “no stub” level of one, prevents the receiver from being stubbed. A “no stub level” of two prevents the receiver and the objects referenced by its instance variables from being stubbed, and so on.

In Listing 2, MyCompany was replicated to level two. This was done because the default replication level of three would have also replicated the employees. If MyCompany was stubbed, then on the next access, MyCompany would be replicated using the default replication level of three, replicating all contained employees. To prevent this, the #noStubLevel method is overridden in the Smalltalk Company class to answer 2, as is shown in Listing 5.

AVOIDING REPLICATION

In general, objects are retrieved from a database to interact in the client. In most cases, these objects can only answer messages when replicated in the client image. However, in GemStone, database objects can respond to messages without being replicated in the image. This removes the requirement for objects to be replicated into the client environment when needed to perform a task. These objects can respond according to the behavior defined in the database.

In GemStone, behavior can be defined in two places; the image and the database. This duality is often referred to as the two-space model. The two-space model is powerful but complex. Behavior in the image can be the same or different than the database. If the same method needs to exist in both the image and database, the developer must define the method in the image’s dialect as well as in Smalltalk DB (see Fig. 3).

The two-space model is more complex than using a “non-active” database but does provide much greater flexibility to the application developer. For example, a query needs to be performed to find all the employee that make more than \$800 a week. In a non-active database, the behavior for calculating pay would need to be in Smalltalk. To perform the query, every employee object would need to be replicated in the image to find those who make more than \$800. An active database provides the ability to define salary calculation in the database. Only those employees that made more than \$800 would need to be replicated.

Many non-active database provide the ability to perform queries against instance variables in the database. To optimize the query, an instance variable called weeksPay could be added to Employee to store the value.

Listing 5. Changing Company’s “No Stub” level.

```
"Methods for Company class"

noStubLevel
"This prevents Company and the employees collection
from being stubbed."
^2
```

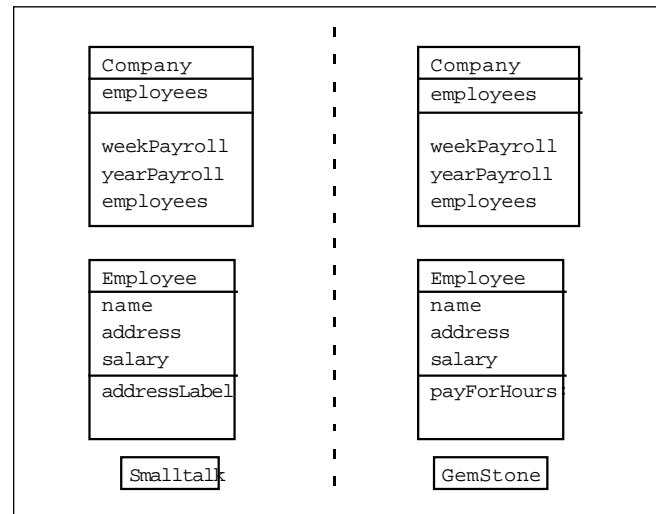


Figure 3. Behavior can be defined in both GemStone and Smalltalk.

However, in this example, if the query changed to use a month’s pay, the database schema would need to be changed and the current instance of Employee would need to be migrated. In the active database, only changes to behavior are needed.

GemStone’s Smalltalk DB is a Smalltalk dialect that has been specialized for database functionality. Smalltalk DB is similar to the VisualWorks, Visual Smalltalk, and IBM Smalltalk dialects but is not syntactically equivalent to any of these. Smalltalk DB provides the Common Language Data Types as defined in the original Smalltalk language specification. Smalltalk DB does not specify any user interface classes. However, Smalltalk DB does provide extensions to optimize queries and handle large collections.

To access behavior defined in Smalltalk DB, messages can be sent to a database object via its proxy. Proxies

Listing 6. Sending messages via proxies.

```
"Sending the message to a proxy using
remotePerform."
| proxy |
proxy := GSI currentSession at: #MyCompany.
proxy remotePerform: #yearPayroll

"Sending the message to a proxy using the gs prefix."
| proxy |
proxy := GSI currentSession at: #MyCompany.
proxy gsyearPayroll

"Sending the message to a replicate using the gs
prefix.
Convert the result to a replicate."
| replicate |
replicate := (GSI currentSession at: #MyCompany)
asLocalObject
replicate as GSOBJECT gsyearPayroll asLocalObject
"replicate asGSOBJECT answers a proxy"
```

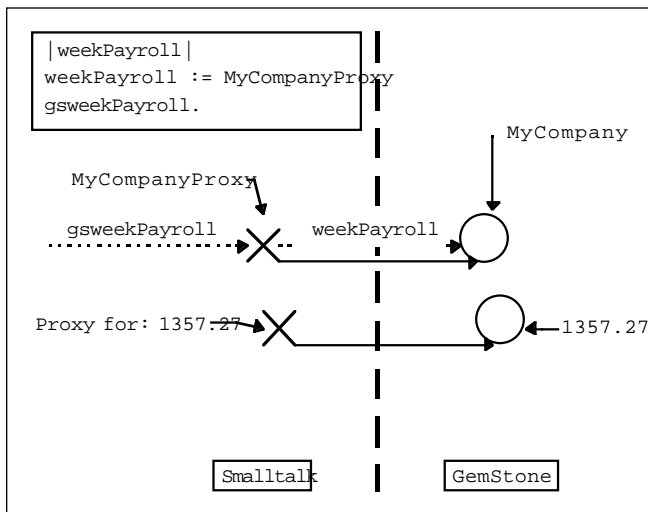


Figure 4. Messages can be sent to database objects via proxies. The resulting object is a proxy.

understand a set of messages similar to the #perform: set of messages defined in the class Object. These messages are #remotePerform:, #remotePerform:withArgs, #remotePerform:with:, #remotePerform:with:with:, etc. In addition to the #remotePerform: series, a proxy also sends messages prefixed with "gs" to its database object. When a message is sent to a database object via a proxy, the resulting object is also a proxy (see Fig. 4).

Database behavior is usually used when the receiver needs to collaborate with several other database objects that are not replicated in the image. Executing this behavior in the database eliminates the overhead of replicating objects when only the results are needed. In the example application, estimated payroll for the year could be calculated in the database. By using database behavior, all of employees for the company do not need to be replicated. Listing 6 contains the Smalltalk code needed to accomplish this.

Using #remotePerform: provides access to database behavior but at the price of dealing with proxies through-

out the application. While proxies are Smalltalk objects, they are merely reference points to database objects. The GSI provides a more seamless method for accessing database behavior, namely forwarders.

Forwarders are Smalltalk objects that forward messages to their database object counterparts. When a message is sent to a forwarder, the forwarder sends a #remotePerform:withArgs: to the proxy for the database object. The behavior is executed by the database object and returns the result. By default, the result of a message sent to a forwarder is a replicate. Forwarders allow the application to access database behavior without having to deal with intermediary objects (see Fig. 5).

In general, forwarders are used to represent the principal access points of the system. These access points contain the major collections that are used to find specific instances of objects. First, the search message is forwarded into the database. Then, only the resulting set of objects are replicated, removing a lot of overhead.

Forwarders can be used to virtually eliminate the use of replicates. This can be very useful when dealing with under-powered client machines. By default, messages sent to a forwarder result in a replicate. By appending messages with "fw", the forwarder will answer with forwarders instead of replicates. In an application that only uses forwarders, the #doesNotUnderstand: message on the forwarder class could be modified to return forwarders by default.

In the example application, replication tuning was used to minimize the number of replicated objects related to MyCompany. Payroll messages are being sent to the database objects to avoid replication. Since MyCompany is only a container for employee objects, it is best to reference MyCompany with a forwarder. This prevents all of MyCompany's employees from being replicated. Payroll messages can be sent to MyCompany without regard to proxy conversions. Some sample code is shown in Listing 7.

When using forwarders, it is important to remember there is a trade off. A forwarder saves space in the image by avoiding replication. The messages often have to trav-

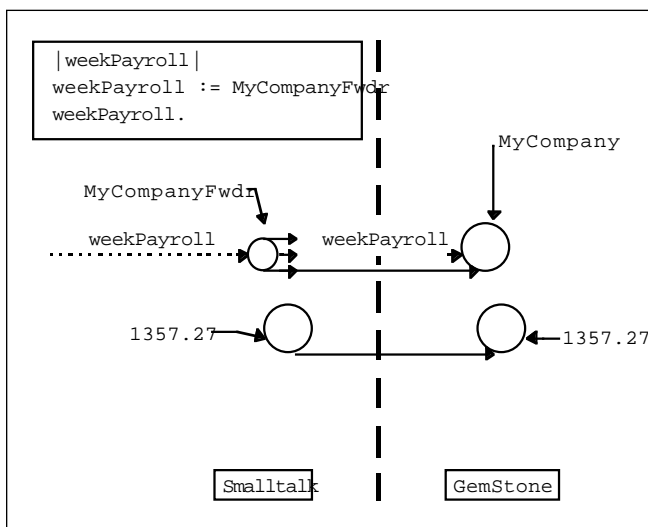


Figure 5. Messages sent to a forwarder are automatically sent to the database object. The resulting object is a replicate.

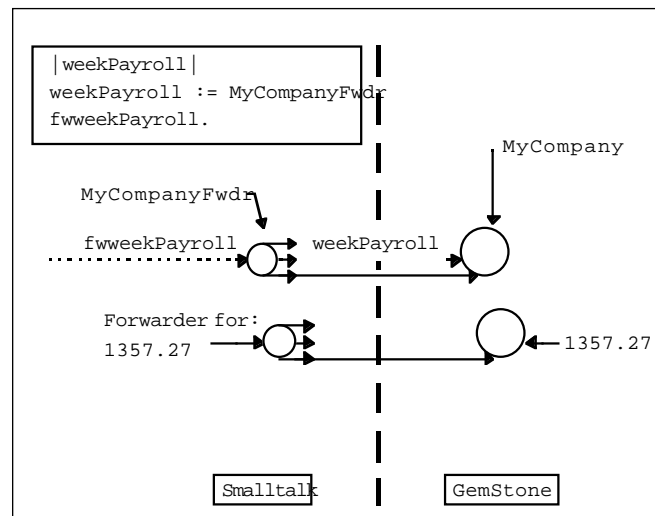


Figure 6. Messages sent to forwarders can be prefixed with a "fw" which causes the resulting object to be a forwarder.

el across the network to be answered. Network latency could cost more than replicating the object. Also, there is the concern of overloading the server with user requests. Essentially, forwarders are not a “silver bullet.” As with anything else in software development, it takes trial and error along with educated guesses to determine the optimal solution for each implementation.

CONCLUSIONS

In general, controlling the size of the image is an important goal with respect to system performance. This is especially true when using an object-oriented database such as GemStone. Object-oriented databases provide

Listing 7. Forwarder examples.

```
"Get the estimate for the weekly payroll."
| forwarder |
forwarder := (GSI currentSession at: #MyCompany) as
Forwarder.
forwarder weeklyPayroll

"Find all employees that live in Raleigh.
Keep the resulting collection as a forwarder."
| forwarder |
forwarder := (GSI currentSession at: #MyCompany)
asForwarder.
forwarder employees fwselect: [ :emp | emp address
city = "Raleigh"
```

mechanisms for controlling the flow of objects between database and image. In GemStone, image size control is provided by the ability to tune and avoid replication.

While limiting replication helps control image size, it is also important to consider database access time. Objects can be retrieved without replicating any related objects. This would prevent image growth but each subsequent access would require a database access, negatively impacting the user's performance. Replication control allows you to balance controlling image size while providing object caching for performance.

In summary, when building a Smalltalk application using GemStone, consider carefully the two-space model. When deciding where to execute behavior, look to see where the objects reside. If most of the objects needed to perform the operation are in the database, then define that behavior in the database. This minimizes the need for replication. If the object is needed for heavy interaction in client, replicate the object. This minimizes network latency.

Of course, there are no simple answers. Every application has a different object model and different hardware constraints. Plan to spend time trying out a variety of replication schemes as a part of performance optimization. The “right” solution is the one that works for you. ☐

John Bentley is Member, Technical Staff at JumpStart Systems, Inc., Raleigh, NC. He can be reached via e-mail at jbentley@jumpstart.com or by phone at 919.832.0490.

Tactical patterns for the real world: Optimization patterns

Darrow Kirkpatrick

THIS IS THE final article in a three-part series on patterns for efficiently implementing and managing domain models. The first article presented a family of instantiation patterns—patterns that aid in creating or initializing objects.

The second article presented a family of patterns for dealing with validation issues. Safeguard showed where to put complex validation logic; and how to prevent invalid domain objects from being used. Deflector showed how to prevent attributes of certain classes from ever taking on illegal values. And Validator showed how to provide default validation of domain attribute values, while allowing end users to modify default validation logic safely. The second article also presented a family of patterns for dealing with informational issues. Verdict showed how to manage the results of a complex and expensive validation across a series of domain objects so that their status or validity may be queried at a later time. And Ticker Tape showed how to collect status information from a lengthy domain operation involving thousands of objects, none of which have visibility to the user interface.

This final article presents a family of patterns for dealing with optimization issues—handling domain models that must perform well while incorporating extra levels of indirection in order to be persistent or transient. So far, the patterns presented have been language-neutral. Although the following two optimization patterns require a specific Smalltalk dialect to implement as described, the principles they embody are generic.

OPTIMIZATION PATTERNS

Avatar (Soft Schema Evolution)

Problem: How do you efficiently implement a persistent object that must be able to add attributes without causing a file shape change, yet must be very fast to access?

Motivation: (Background: The Visual Smalltalk ObjectFiler is able to save and load collections that contain only primitive objects like strings and numbers without requiring a file shape change as the collections change in size. Thus by

implementing a property dictionary inside a higher-level domain object you can add simple state to the domain object after it is designed without having to provide code for schema evolution. Unfortunately, because access to a dictionary is via hashed lookup, the access speed for these additional properties is several times slower than for attributes kept in instance variables.)

You are designing an object that holds various calculation parameters that must be accessed efficiently inside tight loops, and must be stored persistently within each project. It is likely that this object will add parameters over time as additional calculation constants are made available for user editing, yet you do not wish to force a file schema change each time a parameter is added. To solve this problem you create two classes with the same interface: one optimized for persistence and one optimized for access, and you design a mechanism to automatically convert between them as required. Avatar is the transient incarnation of a faster memory form of a flexible persistent object.

Applicability: Use this pattern when you need to implement a simple, persistent object with the following characteristics: it needs to be able to add and hold properties without explicitly mutating shape, yet access to the properties must be extremely fast, not paying the overhead of generalized property dictionary access.

Though the implementation of this pattern presented here is Smalltalk-specific, the general principle of mutating between a flexible persistent representation and a fast memory representation of an object should be applicable in any language that provides object persistence mechanisms.

Solution: Implement a pair of classes with the same interface. One uses a property dictionary to store attributes, the other uses instance variables. The property dictionary object is persistent and has an activation method that mutates it into the instance variable object when it is read from disk. The instance variable object is always used for the in-memory representation of the object. It has a sur-

rogate creation method that answers the property dictionary version of itself for save operations.

Implementation: The Visual Smalltalk ObjectFiler adds two important methods to the Object class: `fileInActivate:` and `fileOutSurrogate:`, as hooks for transforming an object at load and dump time, respectively. To implement the Avatar pattern, the property dictionary object's `fileInActivate:` method instantiates the instance variable object; the instance variable object's `fileOutSurrogate:` method instantiates the property dictionary object. As new attributes are added to the object over time, lazy initialization in accessing methods allows new code to work for older files containing objects that lack those attributes. Note that it may be possible to share the interface of these two classes using inheritance: generally the instance variable class would subclass the property dictionary class—using its interface but not its dictionary.

Consequences: Application of this pattern results in two class implementations that must be maintained in parallel. This is only justified when profiling indicates a performance-critical situation.

Related Patterns: This pattern is related to Bridge¹ in that it provides multiple implementations for the same interface. However, where Bridge provides parallel interface and implementation hierarchies that can vary independently, Avatar simply provides two subclasses whose instances are swapped back and forth.

SPEEDWAY (FAST LIBRARY INTERFACE)

Problem: How do you minimize the cost of referencing a class within a demand-loaded library component?

Motivation: You maintain a library of mathematical classes for performing numerical methods. Because memory is precious and this library is needed only during calculations it is referenced indirectly and loaded on demand at the first reference to a mathematical class. The indirect reference consists of a symbolic class reference via the Smalltalk dictionary, plus a search through a collection of library interfaces to find the library containing the class. This indirection is relatively slow compared to numerically intensive code, especially when it appears inside iterative calculations. To optimize the reference you encapsulate and cache it inside an object that provides a speedy gateway to the library.

Applicability: Use this pattern whenever you indirectly reference a component from within performance-critical code, and the indirection is prohibitively expensive.

Solution: Write a class to act as a fast gateway to the library. The class implements one instance variable for each public class in the library. In performance-critical code, instead of embedding indirect library references, send a message to the Speedway object. The corresponding

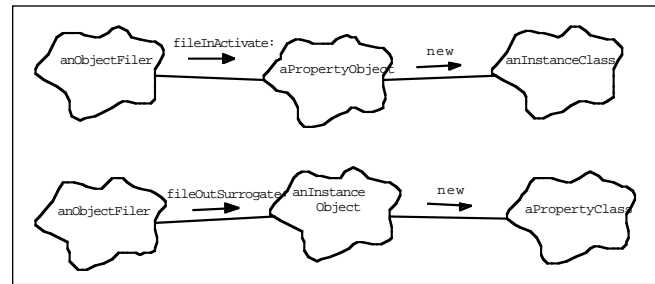


Figure 1. Avatar objects transforming from and to disk.

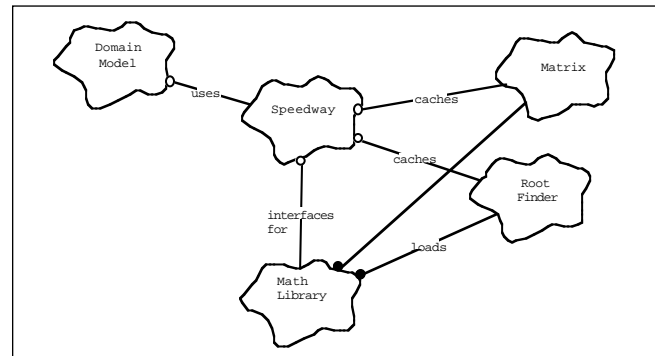


Figure 2. Class hierarchy of math library speedway and client.

method in the Speedway initializes its appropriate instance variable to point to the actual class, first triggering a library load if necessary. Thus in future references the indirection previously in the code is optimized to a single message send which requires a single lazy initialization test.

Implementation: The Speedway is typically a Singleton object managed by a parent class of the domain model, so it can easily be accessed by any domain object. When preparing a runtime application you need to flush the cached library references to avoid statically binding the library component to the image.

Rather than lazy initializing a single variable at the first library reference it may be desirable to populate the entire Speedway. This may provide a slight performance enhancement and allow consolidating some initialization code.

Consequences: This pattern requires maintaining an additional class. Since the library indirection described here is not slow relative to most code, this technique is justified for performance reasons only in critical code. However Speedway has another benefit—it objectifies and documents the public interface to a library subsystem, as far as that interface is limited to symbolic class references.

Related Patterns: Speedway is similar to Facade in that it provides a public interface to a subsystem. But the intent is different: Speedway provides an optimization whereas Facade usually provides some additional thin layer of behavior to make high-level use of the subsystem easier. Note that it may be useful to implement a Facade that incorporates Speedway's caching behavior.


Another approach to solving the library interface prob-

continued on page 34

lem is to implement class Proxy objects. When a class Proxy is referenced it loads the necessary library and then *becomes* the real class object. This approach requires a more sophisticated and intrusive implementation, but results in complete transparency for clients who use library classes.

CONCLUSION

This series documented families of implementation patterns used by our development group for creating domain models in engineering applications. In presenting a particular “handbook” of tactical patterns for a specific domain, these articles are examples of a management or documentation pattern. The application of this higher-level management pattern results in a pattern language—a concise narrative of the principles pervading a body of code.

Pattern languages are useful at different levels of abstraction in different domains by different teams. Through pattern languages Alexander, Beck, Gamma, Helm, Johnson, Vlissides, and others have given us a powerful tool for communicating the craft of software engineering to each other. I hope these articles will encourage other engineers to discover and publish handbooks of domain-specific patterns that have helped them to deal with the demands of creating real-world software. 

Reference

1. Gamma, *et al.* *Design Patterns*, Addison-Wesley, Reading, MA, 1994.



Jan Steinman



Barbara Yates

Documents on the Web

ONE YEAR AGO, we launched this column with an essay on project documentation. We followed up in September 1995 with some code sketches for implementing a hyperliterate documenting system for VisualWorks under Envy.

Now the World Wide Web has taken the planet by storm. This is both a crisis and an opportunity. Those who ignore the Web risk ending up as roadkill on the infobahn. On the other hand, those who approach the Web with a thoughtless, knee-jerk reaction are just as likely to fail.

The latter approach can significantly reduce productivity. “MegaCorp’s goal is to have all project documentation available on the Web,” sounds nice, until it is followed with “therefore, all programmers will immediately attend HTML training, so they can create their documentation for the Web.” This makes it very difficult to fulfill our first principle of hyperliterate programming:

The documentation for a thing must be on the same conceptual level as that thing.

The developers will have to “shift mental gears” to get out of Smalltalk mode and into HTML mode. Knuth’s initial concept of literate programming required programmers to learn and use a complex textual mark-up language, which muddled their conceptual space. Eventually, tools for LaTeX emerged, but literate programming never quite reached the masses, largely because of the cognitive dissonance between coding programs and coding documentation.

“No problem, we’ll invest in Web authoring tools to make the HTML part easy.” This also kills productivity, which is now a victim of ignoring our second principle of hyperliterate programming:

The documentation for a thing must constantly and accurately describe that thing.

If developers are leaving Smalltalk to write Web pages,

no matter how good their Web authoring tools, neither their code nor their documentation is going to be as good as it would be if they combined the two activities in one environment. One or the other will suffer, and it is almost always the documentation that is not “constantly and accurately” describing the code.

In reality, the only principle of hyperliterate programming that Web authoring partially fulfills is the third one:

The documentation for a thing must be accessible; by creators, their peers, reusers, reviewers, end-user documenters, and the merely curious.

But wait, who does Web authoring give improved access to? Certainly not the creator or their peers, who must now have both Smalltalk and a Web browser running in order to do their job. Probably not reusers, once they make the initial Web query and then need deeper access. Probably not all reviewers, some of whom will want detailed information from within Smalltalk.

So, Web authoring provides increased access to end-user documenters and the merely curious, at the expense of the two usage roles most involved with development—creators and their peers!

Finally, our fourth principle of hyperliterate programming states:

The documentation for a thing must be measurable, quantitatively and especially qualitatively.

Anyone who can type “du -s” in the root directory of a UNIX Web tree will get a gross measure of the documentation, but what will that tell us? Web authoring is inappropriate for hyperliterate programming because it is file-based—you will need an entire new suite of tools to do quantitative and qualitative analysis. If you’ve got a perl guru in your group, that might work, but why not leverage the Smalltalk talent you’ve been carefully growing? It’s unlikely that files of HTML code will ever be measured as part of a repository-centric metrics program.

THE WEB IS AN EXPORT TARGET

Just because we don’t believe “Web authoring” is appropriate for hyperliterate Smalltalk development doesn’t mean we think the Web useless. “Web authoring” means

Jan Steinman and Barbara Yates are co-founders of Bytesmiths, a technical services company that has been helping companies adopt Smalltalk since 1987. Between them, they have more than 22 years of Smalltalk experience. They can be reached at Barbara@Bytesmiths.com or Jan@Bytesmiths.com, or via <http://www.bytesmiths.com>.

the creation of Web documents by humans, which is great if you want to create a “cool site” or impress others with your HTML prowess, but it is at odds with the principles of hyperliterate programming, which require keeping documentation as close to the code as possible, in terms of granularity of concept, as well as physical location.

The knee-jerk problems happen when some vice-president says “We gotta get on the Web!” and the ripple effect causes otherwise bright people to do stupid things, such as dictate that all Smalltalk documentation will be in HTML.

Fortunately, HTML is easy to generate from your Smalltalk-resident documentation. If you followed the implementation sketch for Smalltalk hypertext we presented in September 1995, you already have a good start.

That implementation used a special emphasis for VisualWorks *Text* that allowed it to treat Smalltalk expressions specially. This was but a small step from “real” hypertext. We have since extended so it has a notion of both an *anchor*, or visible text with human-meaningful presentation, and an *action*, which is a block of Smalltalk source code. To distinguish this, we call it a “clickAction.”

VisualWorks has been criticized for being divorced from platform capabilities, but its *Text* class provides an abstraction for styled text that is much more powerful than thoughtlessly abdicating all presentation to platform widgets. *Text* has an efficient tagged-character facility that allows you to associate arbitrary objects with runs of characters. Three common tag types are:

- A *simple* emphasis *Symbol*, such as `#bold` or `#italic`,
- a *compound* emphasis *Array* of other emphases, such as `#(#bold #italic)`,
- a *parameterized* emphasis *Association* between a *Symbol* and an arbitrary object.

This last capability is used for things like colored text and different fonts or font sizes.

We defined a new parameterized emphasis for *Text* that contains the *Symbol* `#clickAction` associated with Smalltalk source code for a block. We made changes to *ParagraphEditor* so that double-clicking one of these “clickActions” causes the block to be evaluated—instant hypertext! To enable this, you need a “global method” that can discriminate clickActions:

Object

`isClickAction`

“When used as an emphasis in a *Text*, do I function as a hyper link? Hardly!!”

`^false`

SequenceableCollection

`isClickAction`

“When used as an emphasis in a *Text*, do I function as a hyper link? I do if any of my contents does.”

`^self`

`detect: [:object | object isClickAction]`

`transform: [:ignored | true]`

`ifNone: [false]`

CharacterArray

`isClickAction`

“When used as an emphasis in a *Text*, do I function as a hyper link? Strings and Symbols are never considered active emphases. This override keeps the superclass method from examining each of my Characters to see if they are hyper links.”

`^false`

Association

`isClickAction`

“When used as an emphasis in a *Text*, do I function as a hyper link? I do if my key is `#clickAction`, in which case my value better be block-like, but I don’t check that here.”

`^#clickAction == key`

Text

`hasClickAction`

“Do I contain any hyper links?”

`^runs values`

`detect: [:emph | emph isClickAction]`

`transform: [:ignored | true]`

`ifNone: [false]`

`hasClickActionAt: characterIndex`

“Do I have a hyper link at the given
<characterIndex>?”

`^(self emphasisAt: characterIndex) isClickAction`

You may notice the strange method `#detect:transform:ifNone:`, which is like `#detect:ifNone:`, except that when the first block answers true, the value is passed through the second block. We discovered that we usually use the result of `#detect:ifNone:` this way, and so made it a bit easier to do:

Collection

`detect: booleanBlock transform: transformBlock ifNone: exceptionBlock`

“Evaluate <booleanBlock> with each of the receiver’s elements as the argument. Pass the first element for which <booleanBlock> evaluates to true through <transformBlock> and answer the result, or answer the evaluation of <exceptionBlock> if no elements assert <booleanBlock>.”

`^transformBlock value: (self detect: booleanBlock
ifNone: [^exceptionBlock value])`

You should combine one or more simple presentation emphases with a clickAction for it to display differently, rather than deciding that all clickActions are going to be presented a particular way. We defined a simple emphasis `#link` as a blue underlined style, to make it familiar to those with Web experience. We don’t have room for that code today, but you’ll need to make new instance creation methods for both *CharacterAttributes* and *TextAttributes*.

BUT WHERE'S THE HTML?

Once you have the foundations—a proper object model for hypertext—spitting out HTML is almost trivial. We use a distributed responsibility pattern familiar to anyone who has examined how #printString works:

Object

asHtml

"Answer a representation of myself that is suitable for use in a Web page. Subclasses should not override this method; rather, they should override htmlOn:."

```
| stream |
stream := (String new: 100) writeStream.
self htmlOn: stream.
^stream contents
```

Object

htmlOn: aStream

"Place on <aStream> a representation of myself that is suitable for use in a Web page. The default representation for objects is a #storeString representation in 'code' style. Answer <aStream>."

```
^aStream nextPutAll: '<CODE>';
store: self;
nextPutAll: '</CODE>';
yourself
```

At this point, different objects are free to render themselves into HTML as they see fit. Of course, the one we've been concentrating on is *Text*, and so we have the requisite big, ugly method. Much of this complication is because of the desire to preserve some of the presentation of lines that begin with tabs. Since HTML presentation is normally driven by emphasis rather than content, treating tabs as presentation required an awkward, double-pass treatment:

Text

htmlOn: aStream

"Place on <aStream> a representation of my contents suitable for use in a Web page. Answer <aStream>. For each emphasis found, write beginning and ending HTML tags. For each special HTML Character, write the appropriate HTML character entity. For lines beginning with tabs, write the proper indented definition list."

```
| turnOff |
turnOff := (String new: 16) writeStream.
^self class subscriptOutOfBoundsSignal
handle: [:ex |
aStream nextPutAll: turnOff contents.
ex returnWith: aStream]
do: [ | here tabLevel prevTabLevel thisEmphasis
endEmph char characterEntity |
"Handle initial tabs properly."
```

```
Tab == self first ifTrue: [^(Text with: CR), self
htmlOn: aStream.].
```

```
here := 1.
prevTabLevel := tabLevel := 0.
"Repeat the following until I have no more data."
[here >= self size ifTrue: [^aStream].
```

```
"For each emphasis, build up a proper tag and an
untag."
thisEmphasis := self emphasisAt: here.
thisEmphasis class == Array ifFalse:
[thisEmphasis := Array with: thisEmphasis].
1 to: thisEmphasis size do: [:i | | emph tags |
tags := (emph := thisEmphasis at: i)
isClickAction
ifFalse: [HtmlTags at: emph ifAbsent:
[HtmlNoTag]]
ifTrue: [emph value asHref -> '</A>'].
aStream nextPutAll: tags key.
turnOff nextPutAll: tags value].
```

```
"For lines that begin with one or more tabs, build
a proper level of indentation."
endEmph := here + (self runLengthFor: here).
[here < endEmph] whileTrue:
[char := self at: here.
char == CR ifTrue:
[tabLevel := 0.
[tabLevel := tabLevel + 1.
Tab == (self at: here + tabLevel)]
whileTrue: [].
tabLevel := tabLevel - 1.
tabLevel = prevTabLevel ifFalse:
[char := Tab. "to suppress <P>"
(prevTabLevel - tabLevel) abs
timesRepeat:
[aStream nextPutAll: (tabLevel >
prevTabLevel ifTrue: ['<DL>'] ifFalse:
['</DL>'])]].
tabLevel > 0 ifTrue: [aStream nextPutAll:
'<DD>'].
prevTabLevel := tabLevel].
```

```
"For each character with a given emphasis,
write the character or its HTML-legal
equivalent."
characterEntity := HtmlCharacterEntities at:
char ifAbsent: [].
characterEntity == nil
ifTrue: [aStream nextPut: char]
ifFalse: [aStream nextPutAll:
characterEntity].
here := here + 1].
aStream nextPutAll: turnOff contents.
turnOff reset.
here := endEmph] repeat.
aStream]
```

If you simply feed this method to Smalltalk, it will complain bitterly, because we've added new class variables to *Text*. We don't like to change the definition of classes, but luckily, you don't need to if you're only adding class variables. We have all this code in a common ENVY application called *HyperTextBytesmiths*, with a #loaded method that adds the needed class variables to *Text* on the fly, and of course, a #removing method that removes those class variables when we're done:

HyperTextBytesmiths

characterEntityTable

"Answer a Dictionary that associates non-ASCII characters with their HTML character entities."

^IdentityDictionary new

```
at: Character cr put: '<P>';
at: Character tab put: '';
at: $" put: '&quot;';
at: $& put: '&amp;';
at: $< put: '&lt;';
at: $> put: '&gt;';
at: (Character value: 160) put: '&nbsp;';
at: (Character value: 161) put: '&iquest;';
at: (Character value: 162) put: '&cent;';
at: (Character value: 163) put: '&pound;';
at: (Character value: 165) put: '&yen;';
at: (Character value: 167) put: '&sect;';
at: (Character value: 171) put: '&laquo;';
at: (Character value: 176) put: '&deg;';
at: (Character value: 177) put: '&plusmn;';
at: (Character value: 181) put: '&micro;';
at: (Character value: 182) put: '&para;';
at: (Character value: 183) put: '&middot;';
at: (Character value: 187) put: '&raquo;';
at: (Character value: 210) put: '&reg;';
at: (Character value: 211) put: '&copy;';
at: (Character value: 225) put: '&AElig;';
at: (Character value: 225) put: '&AElig;';
at: (Character value: 233) put: '&Oslash;';
at: (Character value: 241) put: '&aelig;';
at: (Character value: 249) put: '&oslash;';
at: (Character value: 251) put: '&szlig;';
yourself
```

loaded

"Add to TextConstants."

TextConstants

```
at: #HtmlNoTag put: '->';
at: #HtmlTags put: self tagTable;
at: #HtmlCharacterEntities put: self
```

characterEntityTable

removing

"Take away what I added to TextConstants."

TextConstants

```
removeKey: #HtmlNoTag ifAbsent: [];
removeKey: #HtmlTags ifAbsent: [];
removeKey: #HtmlCharacterEntities ifAbsent: []
```

tagTable

"Answer a Dictionary that associates a Text emphasis symbol with an Association of two Strings; the key is a tag used to turn on the emphasis, the value is used to turn off the emphasis."

^IdentityDictionary new

```
at: #bold put: '<B>' -> '</B>';
at: #underline put: '<EM>' -> '</EM>';
at: #Heading1 put: '<H1>' -> '</H1>';
at: #Heading2 put: '<H2>' -> '</H2>';
at: #Heading3 put: '<H3>' -> '</H3>';
at: #Heading4 put: '<H4>' -> '</H4>';
at: #Heading5 put: '<H5>' -> '</H5>';
at: #Heading6 put: '<H6>' -> '</H6>';
at: #italic put: '<I>' -> '</I>';
at: #strikeout put: '<S>' -> '</S>';
yourself
```

Now we can generate HTML from any *Text*, but it is necessarily "embeddable" HTML only suitable for the "body" part of a Web page. We have numerous ways of producing a complete page, but the most useful way works from any *ParagraphEditor*, because it is the foundation text editing class in VisualWorks. (The *Stream* implementation of #htmlFor: is left as an exercise for the reader!)

ParagraphEditor

htmlOn: aStream


"Place on aStream a representation of my contents suitable for use in a Web page."

^aStream

```
nextPutAll: '<HTML><HEAD><TITLE>';
print: sensor window label;
nextPutAll: '</TITLE></HEAD><BODY>';
htmlFor: self text;
nextPutAll: '</BODY></HTML>'; cr;
yourself
```

CONCLUSION

The Web can be a powerful communication tool, but like all tools, it can be misused. Just as a screw driver or an ice pick can kill a person, mandating inappropriate use of the Web can kill a project.

We've demonstrated some techniques for exporting off-line HTML from your "hot" Smalltalk documentation. Next month, we'll show you how to serve your hot documentation "on-line," so that a Web browser can view up-to-the-minute project documentation. This should be enough to silence any VP who comes storming in, shouting "What are you guys doing about the Web?" 



Alan Knight

Smalltalk Solutions

THIS MONTH I'M going to be discussing the Smalltalk Solutions conference that took place in New York at the beginning of March. This isn't my usual territory, since it hasn't got much to do with the Internet, so I'll break with tradition and discuss some of the topics I generally avoid: rumors, impressions, and products I haven't used. Obviously, you shouldn't be basing important decisions on my first looks at a product, or on unsubstantiated rumors. In fact, just to make things more interesting, I made up one of the rumors myself. See if you can spot it yourself before turning to the end for the solution.

In general, the thing that impressed me most about this conference was the maturing of the Smalltalk industry. The number and variety of different applications was remarkable, and they weren't confined to traditional business systems. For example, I was surprised to learn that the driver's license kiosks in Ontario (where I live) are programmed in Smalltalk.

It's no longer the case that everyone is talking about small pilot projects and introducing Smalltalk into the organization. An increasing number of organizations have delivered mission-critical systems in Smalltalk and realized significant gains from them. There's more concern with "business value" and how to make the transition to the "early majority" of users than there is with the latest cool features.

NEWS AND RUMORS

Speaking of cool features, Java is heavily in the news these days, and often cited as a threat to Smalltalk. In fact, many advocates of Java seem to believe that it instantly makes all existing languages and operating systems obsolete. I admit to knowing some people who feel that way about Smalltalk, but the Java zealots are doing a remarkable job of worrying people who really ought to know better. This leads to a couple of interesting Java rumors.

One, which appeared in comp.lang.smalltalk suggested that ParcPlace-Digitalk was in the process of making a Java VM which would be several times faster than Sun's. This is technically plausible, since current Java imple-

mentations are abysmally slow, and it shouldn't be that difficult to adapt a Smalltalk VM to run Java. On the other hand, I've seen no confirmation of this, especially not from PPD.

Microsoft is reacting to the Java hysteria, and although they have licensed it for use in their own web browser, they're also at work on a product to rival Java, leveraging their existing technology. In accordance with the emerging standards for naming conventions of such products (bad puns based on coffee) the new product will be named "au lait."

One of the strengths of Java is that implementations are quite cheap, and often free. There have been numerous complaints from the community that, in the pursuit of the corporate market, Smalltalk vendors have priced themselves out of range of individuals and small companies. Anyone who feels this way should be happy to hear the latest from Skip McGaughey (market manager for VisualAge). Responding to a question on his keynote speech, he said that we "absolutely need" a cheap Smalltalk that runs on an 8- to 16-MB machine and comes with multimedia instructional software so that users don't need expensive training. "Are we there today? No. Will we be there a year from now? We have to be."

Getting back into the factual and the present, the draft X3J20 report on the ANSI standard Smalltalk is now available for review. The initial informal review period ended April 30, but review and revisions continue. To get a copy, contact Lynn Barra at 202.626.5738 or lbarra@itic.nw.dc.us. There's also an X3 web page at <http://www.x3.org>. The ANSI committee has done some very interesting work, and obviously put a lot of thought into defining the language without overconstraining future implementations. They have decided not to define a standard for namespaces, not because they don't think they're important, but because they think standardization now would be premature. Although I'm a little disappointed that vendors won't be forced to implement namespaces, I have to agree with their reasons.

A big part of this column turned out to be about IBM and OTI. That's partly because there was interesting news on that front and partly for another reason. It's been quite a while since the merger and there's still very little infor-

Alan Knight is a Smalltalk guru with The Object People. He can be reached at 613.225.8812 or by email as knight@acm.org.

mation on PPD's future plans, which are of critical importance to anyone working in Smalltalk. I had prepared a modest diatribe on the subject and it had already been typeset when the information finally started to flow. It's still a trickle, but it's enough that I'm willing to hold back my wrath a little while, especially since the trickle contains encouraging words like "no runtime fees."

IBM BUYS OTI

When IBM became a Smalltalk vendor it marked a significant milestone in the evolution of the language, giving it legitimacy in the eyes of many major corporations. IBM had licensed their underlying Smalltalk technology from OTI, and now they have acquired that technology outright.

Given the close relationship between the two companies lately this wasn't a complete surprise, but it did worry a number of people. One worry, for fans of OTI, is that being part of IBM might destroy their unique corporate culture. The other worry, for fans of VisualWorks/Envy, is the long-term outlook for that product. Skip McGaughey tried to dispel these fears in his keynote address.

He re-affirmed that OTI would act as an independent subsidiary of IBM, and that it would continue to be run by Dave Thomas. In fact, he said that all Smalltalk activity within IBM now reports to Dave, so that "in a very real sense, Dave Thomas acquired IBM." He also emphasized the idea of both competing and collaborating. One example of this collaboration was that OTI will continue to supply VisualWorks Envy, enabling their competition, but allowing everyone to win by growing the market. That takes care of one side of the equation, but it remains to be seen how ParcPlace-Digitalk feels about having such an important system component provided by a competitor. Whether or not we see a version of Envy for VisualWave will be a very strong signal of their future direction.

One area where OTI's culture is already being affected is in the relaxation of their vows of silence. OTI staff are known for never letting any information slip before something is officially announced. Skip McGaughey instead emphasized the need for clear communications, even with competitors. For example, he said that IBM will let anyone see their plans for the next version without signing a non-disclosure agreement.

EMBEDDED/SERVER SMALLTALK

OTI's tools for doing embedded and server programming in Smalltalk are one of the most interesting bits of technology I've seen in a while. Although it's an area which is unfamiliar to most of the Smalltalk community, I believe that they will be very important to the future of Smalltalk.

Smalltalk is generally considered to be very resource-intensive. Development environments typically suggest 16 to 32 MB RAM and executables have trouble running in 4 to 8 MB. You certainly wouldn't think of using Smalltalk for a real-time system with only 512 K, would you? OTI would, and they've been doing it for quite a long time. Now they've come out with their second generation of

INFO@SIGS

SIGS Publications, Inc., 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

ARTICLE SUBMISSION

To submit articles for publication, please contact:
John Pugh & Paul White, Editors, 885 Meadowlands Dr. #509, Ottawa, Ontario, K2C 3N2 Canada; email: streport@objectpeople.on.ca

PRODUCT REVIEWS AND ANNOUNCEMENTS

To submit product reviews or product announcements, please contact the Editors at the address above.

CUSTOMER SERVICE

For customer service in the US, please contact PO Box 5050, Brentwood, TN 37024-5050; 800.361.1279; Fax: 615.370.4845; in the UK, please contact Subscriptions Department, Tower Publishing Services, Tower House, Sovereign Park, Market Harborough, Leicestershire, LE16 9EF, UK; +44.(0)1858.435302; Fax: +44.(0)1858.434958

SIGS BOOKS

For information on any SIGS book, contact: Don Jackson, Director of Books, SIGS Books, Inc., 71 West 23rd Street, New York, NY 10010; 212.242.7447; Fax: 212.242.7574; email: donald_jackson@sigs.com

SIGS CONFERENCES

For information on all SIGS Conferences, please contact: SIGS Conferences, 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7515; Fax: 212.242.7578; email: info@sigs.com

BACK ISSUES

To order back issues, please contact: Back Issue Order Department, SIGS Publications, 71 West 23rd Street, 3rd Floor, New York, NY 10010; Phone: 212.242.7447; Fax: 212.242.7574

REPRINTS

For information on ordering reprints, please contact:
Reprint Management Services, 505 East Airport Road, Box 5363, Lancaster, PA 17601; Phone: 717.560.2001; Fax: 717.560.2063

ADVERTISING

For ad information for any SIGS publication, please contact:
East Coast/Europe: Gary Portie
Central US: Elisa Marcus
Recruitment: Michael Peck
Exhibit Sales, West Coast: Kristin Viksnins
Exhibit Sales, East Coast: Sarah Olszewski
Phone: 212.242.7447; Fax: 212.242.7574; email: sales@sigs.com
West Coast: Diane Fuller
Phone: 408.255.2991; Fax: 408.255.2992; email: dhfsigs@hooked.net

INTERNATIONAL OFFICES

SIGS Conferences Ltd., Brocus House, Parkgate Road, Newdigate, Surrey RH5 5AH, United Kingdom. Phone: 011.44.1.306.631.331; Fax: 011.44.1.306.631.696; email: 100131,3500@compuserve.com.

SIGS France, 105 rue Jules Guesde, 92532 Levallois Perret Cedex, Paris, France. Phone: +33 (1) 41 06 18 00; Fax: +33 (1) 41 06 18 19; email: 100631,1050@compuserve.com.

SIGS Conferences GmbH, Odenthaler Strasses 47, D-51465 Bergisch Gladbach, Germany. Phone: 011.49(0).2202.936.810; Fax: 011.49(0).2202.936.812; email: 100634,2070@compuserve.com.

SIGS HOME PAGE AND ONLINE MAGAZINES

Access the SIGS Home Page at <http://www.sigs.com>;
Object Currents at <http://www.sigs.com/objectcurrents>; and
The X Spot at <http://www.sigs.com/xspot>.

embedded tools and they're starting to promote them more aggressively.

In getting such small footprints, they have an advantage over most of us because they're typically writing for systems that don't have screens, mice, keyboards, or disk drives. All these things need code to control them, which takes spaces. On the other hand, these absences make developing and debugging with these machines extremely difficult. OTI's toolset tries to make it more like regular development. Here's a quick summary of the features I found most interesting.

You develop on a workstation, but with a difference. In regular Smalltalk your development and execution environments are the same. You write code, then execute it in the same environment that runs your browsers, compilers, and so forth. When you're ready to deliver, you strip out what you're not using. In Envy/Embedded you create a specification of an image to run on the embedded system, and you write code to execute in that environment. The class libraries you use can be entirely different than what's on your workstation.

When you're ready to run, your code is transferred (through a serial port or network interface) to the real machine and run. You have full interactive debugging facilities, it's just that the debugger is on your workstation and the code being run is on the target system. You have remote inspectors and workspaces, single-stepping, and the ability to save code and continue.

For packaging, there are a number of very interesting features. The entire virtual machine is re-entrant, so it can be put in ROM and shared between multiple images. The same thing can be done with large parts of the image. On some real-time operating systems Smalltalk can use the operating system threads instead of the normal Smalltalk processes.

It's these last two that seem to me to have the most significant implications for desktop environments. Better separation of development and delivery environments is important, but the ability to share most of the environment in read-only mode could easily lead to truly shareable Smalltalk DLLs. This would let me run many fine-grained Smalltalk applications simultaneously without the memory overhead of starting a separate VM for each one. I don't think that the use of real operating system threads is critical if you have a non-blocking API capability, but it's something for which Smalltalk is often criticized, so it's nice to see a real implementation.

A lot of these features are very similar to IBM's forthcoming MVS Smalltalk, and this is no coincidence. Brian Barry of OTI, in presenting the embedded product, described MVS as a really, really large embedded system. Many of the characteristics of embedded systems are shared with servers, and many of the same features are important.

PROGRAMMING EPISODES

Ward Cunningham gave a talk on a model of the development process, subtitled "Finding and Exploiting Great Objects When You Barely Have Time to Think." Ward works

in the financial world, with very demanding customers and very tight deadlines. This is his model of how to develop in that environment and still produce great code. I guess it's a development method, but it's a lot looser and willing to rely on people's competency than most of the methods I've seen. That makes it appealing to me as a programmer, but it still has enough structure that I can believe it would help meet deadlines. That's quite an accomplishment.

First, a bit of background, in case you're unfamiliar with Smalltalk theology. Ward Cunningham is a very long-time Smalltalker, who worked at Tektronix in close collaboration with Kent Beck. They did a lot of cool stuff together, like designing the HotDraw graphical editing framework and inventing CRC cards. They were also among the first to look at applying patterns to software, and although this talk was not described in terms of patterns, the influence was clear.

He started with a very simple structure for software development, which was successively elaborated with more detailed ideas, applicable in particular situations. It's hard to describe, so I'll just give a bit of flavor by paraphrasing a couple of the ideas.

Spike Solution

You've got an informal labor plan and you want to move towards implementation. You need to do some preliminary coding to make sure you understand the requirement and its implications, but you don't want to get bogged down in dealing with the complexities of existing code. So, write the smallest possible code to perform that requirement, independent of the existing mechanisms.

For example, take a clean image and implement the absolute basics of that requirement, as fast as possible. This is called a "Spike Solution" because it's like driving a spike into a wall. You do it to find out how thick the wall is and where you'll come out. You stop driving the spike as soon as the tip comes out the other side. Later on you'll drive the nails for real.

Motivated Consolidation

Consolidation is important, but your consolidation will be better the longer you put it off. Also, in the normal course of things you will never consolidate, because this is an environment where you barely have time to think. Therefore, you consolidate when, and only when, it's the shortest route to getting something out the door. Fixing the code and adding the new feature will take less time than just hacking in the new feature, and you only do it for regions of the code where it's motivated (i.e., funded). One of the essential elements for consolidation is regression tests. They're incredibly liberating, because they let you change something radically and still know if it works or not.

If this looks interesting, you can find more information, including the "Episodes" pattern language on which this is based, on Ward's web site at <http://c2.com>.

The solution to the rumor puzzle is, read from right to left: "romur tial ua eht pu edam I" ☿



Jay Almarode

Multi-user canonicalization

DURING APPLICATION DEVELOPMENT, it is sometimes useful to guarantee that your code does not create two objects that are logically equivalent. Instead, you would like the attempt to create a new object to actually return an existing, equivalent object, if it should exist.

Otherwise, the new object can be created and registered in such a way so that subsequent attempts to create an equivalent object will return this one. This technique, called canonicalization, makes your code more efficient by eliminating redundant objects and allowing you to take advantage of object identity. For example, if you know that there will only be a single object to represent some logical entity, you can use identity comparisons (`==` or `~`) when scanning for the presence of the object in some collection. Identity comparisons are usually more efficient to use because they are typically in-lined by the compiler, and do not require fetching the objects to return an answer.

Most Smalltalkers are already familiar with the concept of canonicalization with the use of symbols. By definition, symbols are guaranteed to be unique, so that any symbol with the same sequence of characters will have the same identity. This means that no matter where or how a symbol is created, an identity comparison of two equivalent symbols always returns true. In fact, the implementation of `=` for class `Symbol` is the same as `==`.

The uniqueness of symbols allows them to be used in fast identity-based collections, such as a key in an identity dictionary, while preserving the semantics of equality look-up. This is one reason why method selectors are symbols rather than strings, since they are used as keys in a method dictionary.

A common usage for canonicalization is to implement a smart cache of objects whose state is derived from an external system. For example, if objects are being materialized from a relational database, then a cache typically maps a relational primary key value to its corresponding Smalltalk object. If some part of the

application needs an object with a particular key value, the cache is consulted first. If there is already an entry in the cache for that particular key, then the application can avoid having to execute time-consuming code to communicate with the relational database and perform the relational-to-object mapping, since it has already been done before (of course, if the relational data has been modified since the initial caching occurred, then the cache must somehow be updated or invalidated, but that is a different problem).

Building your own canonicalization mechanism is fairly straightforward in a single-user Smalltalk system. A typical implementation is to override the instance creation method to check for the presence of an existing, equivalent object before creating a new one. A common implementation is to maintain a dictionary in a class variable, where the keys of the dictionary are the logical values upon which equivalence is determined, and the values of the dictionary are instances of the class that have already been created. I suggest using a class variable, rather than a class instance variable, so that creating instances of subclasses consults the same dictionary. Another advantage of this implementation is that it is very easy to get all instances of a class and its subclasses.

To illustrate this technique, here is the implementation of an instance creation method for class `Employee`. In addition to having instance variables for name and social security number, `Employee` has a class variable called "CanonDictionary" that is initialized to a dictionary. In this model, social security number is the primary key upon which equivalence is based, i.e. we never want object memory to contain more than one instance of `Employee` with the same social security number. Since we always want an `Employee` to have a social security number, we override the "new" method to raise an error, and require instance creation to occur with the "name:ssn:" method listed here:

```
classmethod: Employee
name: aName ssn: aSSN
"Return an instance with the given name and ssn. If one
```

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@gemstone.com.

does not exist in the canonicalization dictionary, create a new one; otherwise, return the existing one."

```
^ CanonDictionary at: aSSN ifAbsent: [ | emp |
    emp := self basicNew name: aName; ssn: aSSN.
    CanonDictionary at: aSSN put: emp.
]
```

This technique works fine in single-user Smalltalk systems, since only one user is creating objects in this image. But in multi-user Smalltalk, there may be concurrent users who are creating objects in a shared image.

This opens the door to the possibility that users will experience concurrency conflicts on the canonicalization dictionary. In addition, since each user operates with their own transactionally consistent view of objects, there may be more than one user who thinks he or she is creating the first instance of an Employee with a particular social security number. This is because neither user will see the other's modifications until his or her transaction is committed. At the very least, one of the users could experience a concurrency conflict, but it could be worse if both users were allowed to create logically equivalent instances of Employee and the application code depended upon their uniqueness.

Fortunately, by subclassing an existing specialized multi-user class, this situation can be handled correctly. In GemStone Smalltalk, the class RcHashDictionary provides concurrency semantics that are close to what is needed (see my column in the March–April 1995 issue of the *Smalltalk Report* for a description of reduced conflict classes). This multi-user dictionary allows concurrent updaters and removers from the dictionary to perform their operations without conflict, as long as they are using different keys. For example, two concurrent users who are performing at:put: operations with non-equivalent keys will not experience concurrency conflicts. But in our example, concurrent users might try to create instances with the same social security number, so they would experience conflict. What is needed is the ability to recognize these conflicts, choose one of the instances to be the canonical Employee with that social security number, and to replace all references to the noncanonical Employee with references to the canonical Employee (allowing the noncanonical Employee to be eventually garbage collected).

To solve this problem, I created a subclass of RcHashDictionary, called RcCanonicalDictionary. This class only needs to override one method to provide the desired behavior; however, to implement this method requires an understanding of how reduced conflict behavior is achieved. When a user attempts to commit a transaction, the underlying system detects if there are physical conflicts on objects, for example, checking if this transaction wrote an object that another concurrent transaction had already written and committed. For most objects, a physical conflict means the transaction cannot succeed. However, for special reduced conflict objects, they are given a second chance to determine if the physical conflict can logically be resolved.

This involves selectively updating the view of these objects so that the committed modifications of other users are visible, and then replaying the modifications of the current transaction on the reduced conflict objects. If the modifications can be replayed without failing, then the transaction is allowed to commit successfully.

For RcHashDictionaries, the method that replays updates to the dictionary is _replayAt:put:oldValue:. This method is similar to at:put:, except that the third argument is the original value at the given key before the update occurred (this argument is nil if the entry was added for the first time).

This allows the replay method to check if the value before the update is the same during replay as it was when the operation was originally invoked during the transaction. When the operation is replayed, if the current value is not the same as the old value, then we know some concurrent user has updated the dictionary at this key and we should fail the attempt to commit the transaction.

For our new RcCanonicalDictionary, rather than fail the transaction when another user commits a new entry at the same key, we would like to forget the value we were going to insert, and use the value that another user already inserted. This involves *swizzling* all references to the value we were about to insert to the new value inserted by a concurrent user. Fortunately, this is not very hard to do, since we can get a collection of all objects that were written during the transaction, and scan them to find references to our value. This avoids having to scan all of object memory to find references, which is prohibitive for a large scale number of objects.

One thing that must be accounted for when swizzling object references is to correctly update collections where the position of an object in the collection is dependent upon the identity of the object.

In GemStone Smalltalk, Bag and its subclasses use the identity of its elements to determine their positions in the internal implementation structures. Consequently, rather than overwriting the reference to the old value in these collections, the swizzling method first removes the old value and then adds the new value to the collection. Below are the methods to replay the insertion into an RcCanonicalDictionary when a physical conflict is detected, and the methods to swizzle references in general objects and for Bags.

```
method: RcCanonicalDictionary
    _replayAt: aKey put: aValue oldValue: oldValue
    "Stores the key/value pair in the dictionary. If there is
    already a value for the given key, then this method
    swizzles references to refer to the existing value."
    | existingVal |
    "see if there is now an existing entry (added by a
    concurrent user) "
    existingVal := self at: aKey otherwise: nil.
    "if there is no existing entry, update the dictionary;
    otherwise swizzle "
    existingVal isNil
```

```

ifTrue: [ self at: aKey put: aValue ]
ifFalse: [
    " for each object written during this transaction,
    swizzle references "
    (System _hiddenSetAsArray: 9) do: [ :obj |
        obj _swizzleReferencesFrom: aValue to:
        existingVal
    ]
].
" return true to indicate that the transaction can proceed "
^ true
%

method: Object
_swizzleReferencesFrom: obj1 to: obj2
"Scan the named instance variables and indexable portion
of the receiver, looking for references to obj1. For any
that are found, replace the reference with obj2."
" first scan named inst vars "
1 to: self class instSize do: [ :j |
    obj1 == (self instVarAt: j)
ifTrue: [ self instVarAt: j put: obj2 ]
].
" scan indexable portion if necessary "
self class isIndexable
ifTrue: [
    1 to: self _basicSize do: [ :j |

```


```

        obj1 == (self _at: j) ifTrue: [ self _at: j put: obj2 ]
    ]
]
%

method: Bag
_swizzleReferencesFrom: obj1 to: obj2
"If obj1 is contained in the receiver, remove all
occurrences of it, and add the same number of
occurrences for obj2."
" invoke superclass method for named instance variables "
super _swizzleReferencesFrom: obj1 to: obj2.
(self includes: obj1)
ifTrue: [
    (self occurrencesOf: obj1) timesRepeat: [
        self remove: obj1.
        self add: obj2.
    ]
]
%

```

Canonicalization of objects is a useful technique with many applications. In a multi-user environment, canonicalization mechanisms must take into account concurrent users creating equivalent objects.

This column has demonstrated one approach for solving this problem using the power and extensibility of multi-user Smalltalk. 

Editors' Corner



John Pugh



Paul White

CERTAINLY IT HAS been demonstrated over the past few years the merits of using Smalltalk for application development in a wide variety of disciplines. It has successfully been deployed in areas as diverse as banking, insurance, telecom, and utilities, and the list goes on. With that given, it is amazing the number of times we still end up having to justify Smalltalk as a valid choice for software development projects. The misconceptions with respect to speed, difficulty of development, the proprietary nature of the software and so on still persist. It still seems impossible to avoid the “why should I even consider Smalltalk for my application development?” type of questions.

It's never quite clear why these misconceptions cannot be eradicated. Perhaps it's just the nature of the software industry, and engineering in general. Many of the often stated shortcomings of Smalltalk of course have long been addressed. Dynamic compilation, for example, provided a big boost in execution speed, and puts Smalltalk very close in execution speed to C++, when it is used as a fully object-oriented system. Garbage collection is another criticized feature even though today's implementations are so well tuned that these criticisms are for the most part unfounded.

The reason for writing is not to complain about the unfair criticisms, but instead to highlight the work we all have left to do if we're to take Smalltalk to the next level. If we really believe Smalltalk is worth sharing with the rest of the software world, then we need to do a much better sales job. Of course, many companies are working very hard to do just that. GemStone, for example, has taken a big step forward by positioning their OO database product as an application data server that uses Smalltalk as the primary development environment. Given the visibility they've been receiving of late, this is a positive step for all of us. IBM has also clearly demonstrated that Smalltalk is a major part of their future in the software arena. Their moving VisualAge and Smalltalk to their various platforms makes it a viable solution for software development shops that otherwise never would have considered it. And investors in ParcPlace/Digitalk are certainly expecting growth in the industry.

We have to realize that having Smalltalk progress on a pilot project by pilot project basis is not the scale of

growth necessary. We need to see it being adopted by organizations as their leading choice for mainstream application development. And those of us who make a living from this technology cannot afford to stand idly by. We have to take the lead in educating “the masses” in why Smalltalk is a sound business choice. We're now to the point that making arguments strictly on a technology basis will never enable us to achieve this growth. The arguments must be raised to issues of business value, not technical merit. As an example, we believe that for software development to qualitatively improve as an industry, we need to move toward the adoption of notions such as software components.

The arguments (for using Smalltalk) must be raised to issues of business value, not technical merit.

These components need to be self-contained, testable, and well specified on the basis of their behavior, constraints, input types, and output types. If we make this argument convincingly, then it becomes much simpler to convince people that Smalltalk

is probably the best choice for achieving this type of software organization. We urge you to try arguing the case for Smalltalk in this type of context, rather than “but it has garbage collection which saves you a lot of work”—we think you'll be more successful.

The other issue with respect to “selling” Smalltalk is that we cannot afford to wait much longer in succeeding. We probably need to take a type of “storm the beaches” mentality to this, since the time is ripe now for market penetration. The simple fact is that virtually every software development shop is facing the same problem, namely that the systems they are being asked to build are significantly more difficult to construct than systems of even two or three years ago. The new systems being created today must capture business rules and business logic to be considered successful. Well, let's argue for Smalltalk's use because it is the best tool for capturing this knowledge. We think it is possible to demonstrate that Smalltalk is, purely from a language point of view, better positioned for application development than languages such as PowerBuilder, VisualBasic, and even C++. Let's use this to our advantage, rather than arguing whether or not Smalltalk's window creation facilities are as slick as other products—in the longer run, business decisions will always be based on business value! We hope you'll enjoy the issue.

Editors

John Pugh and Paul White
Carleton University & The Object People

SIGS Publications Advisory Board

Tom Atwood, *Object Design*
 François Bancelhon, *O₂ Technology*
 Grady Booch, *Rational*
 George Bosworth, *ParcPlace-Digital*
 Jesse Michael Chonoles, *Lockheed Martin ACC*
 Stuart Frost, *SELECT Software*
 Adele Goldberg, *ParcPlace-Digital*
 Thomas Keffler, *Rogue Wave Software*
 R. Jordan Kriender, *IBM Consulting Group*
 Thomas Love, *Consultant*
 Bertrand Meyer, *ISE*
 Meilir Page-Jones, *Wayland Systems*
 Bjarne Stroustrup, *AT&T Bell Labs*
 Dave Thomas, *Object Technology International*

The Smalltalk Report Editorial Board

Jim Anderson, *ParcPlace-Digital*
 Adele Goldberg, *ParcPlace-Digital*
 Reed Phillips
 Mike Taylor, *ParcPlace-Digital*
 Dave Thomas, *Object Technology International*

Columnists

Jay Almarode, *GemStone Systems Inc.*
 Wayne Beaton, *The Object People*
 Kent Beck, *First Class Software*
 Dwight Deugo, *The Object People*
 Juanita Ewing, *ParcPlace-Digital*
 Bob Hinkle, *Consultant*
 Tim Howard, *FH Protocol, Inc.*
 Ralph E. Johnson, *University of Illinois*
 Alan Knight, *The Object People*
 Mark Lorenz, *Hatteras Software, Inc.*
 Jan Steinman, *Bytesmiths*
 Rebecca Wirts-Brock, *ParcPlace-Digital*
 Barbara Yates, *Bytesmiths*

SIGS Publications Group, Inc.

Richard P. Friedman, Founder, President, and CEO
 John McCormick, Editorial Director

Editorial/Production

Kathleen M. Major, Managing Editor
 Elisa Varian, Director of Manufacturing
 Seth J. Bookiey, Associate Managing Editor
 Dan Olawski, Production Editor
 Sue Mycka, Desktop Designer and Cover Design
 Margaret Conti, Manufacturing Coordinator

Circulation

Elayne Glick, Circulation Director
 Lawrence E. Hoffer, Director, New Business Development
 Byron Scarlett, Assistant Circulation Manager

Advertising/Marketing

Gary Portie, National Sales Manager
 Elisa Marcus, Advertising Manager, Central US
 Michael W. Peck, Advertising Representative
 Kristine Viksnins, West Coast Exhibit Sales
 Sarah Olszewski, East Coast Exhibit Sales
 212.242.7447 (v), 212.242.7574 (f)
 Diane Fuller & Associates, Sales Representative, West Coast
 408.255.2991 (v), 408.255.2992 (f)
 Nancy Beuschel, Promotions Manager for Magazines

Administration

Margherita R. Monck, General Manager
 David Chatterpaul, Senior Accounting Manager
 Bibi Budhrum, Accounts Payable



Publishers of JOURNAL OF
 OBJECT-ORIENTED PROGRAM-
 MING, OBJECT MAGAZINE,

C++ REPORT, THE SMALLTALK REPORT, THE X JOURNAL, JAVA
 REPORT, OBJECT CURRENTS (ONLINE), THE X SPOT (ONLINE),
 OBJECT EXPERT (UK), and OBJEKTSPKTRUM (GERMANY)



For more object
 news and analysis,
 check out SIGS
 online at
<http://www.sigs.com>

Table of Contents

July–August 1996

Vol 5 No 9

Features

How to display an object as a string: printString and displayString 4

Bobby Woolf

Objects are often identified using a string that describes the object. Bobby offers some guidelines on how to implement and use printString and displayString to produce an object's description.

Smalltalk SQA—The Public/Private Problem #2 9

Jeff McKenna

Jeff explores the public/private problem, proposes a solution, and then describes how to use the solution to define a unit, interface, and aggregate testing.

Proper use of class methods 12

Jill Nicola

Using class methods for creating new objects, managing class variables, and building test objects.

Implementing mixins in Smalltalk 14

Terry Montlick

A simpler, more pragmatic approach for working with mixins.

Columns



Managing Objects 16

Smalltalk as an Internet server
Jan Steinman & Barbara Yates

Mainframes and terminals are tired, client-server is wired. With a simple TCP-based server framework, you can concentrate on

the service you provide, without getting bogged down in server details.



Getting Real 21

Communicating between sessions

Jay Almarode

Jay discusses two kinds of client-to-client communication that can be supported by multi-user Smalltalk and shows how to use these services to implement concurrent processing algorithms.



Visual Programming 24

Reusable components

Dwight Deugo & Wayne Beaton

Deugo and Beaton describe the building blocks for constructing any application window: parts and corrections.

Departments

Editors' Corner 2

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar–Apr, July–Aug, and Nov–Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1996 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Bulk rate U.S. postage paid Lancaster, PA, permit 161. Canada Post International Publications Mail Product Sales Agreement No. 290386.

Individual Subscription rates 1 year (9 issues): domestic \$89; Mexico and Canada \$114, Foreign \$129; Institutional/Library rates: domestic \$199, Canada & Mexico \$224, Foreign \$239. To submit articles, please send electronic files on disk to the Editors at 885 Meadowlands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

POSTMASTER: Send domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sigs.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

How to display an object as a string: *printString* and *displayString*

Bobby Woolf

WHEN I TALK about how to use different sorts of objects, people often ask me what these objects look like. I draw a bunch of bubbles and arrows, underline things while I'm talking, and (hopefully) people nod knowingly. The bubbles are the objects I'm talking about, and the arrows are the pertinent relationships between them. But of course the diagram is not just circles and lines; everything has labels to identify them. The labels for the arrows are easy: The name of the method in the source that returns the target. But the labels for the bubbles are not so obvious. It's a label that somehow describes the object and tells you which one it is. We all know how to label objects in this way, but what is it that we're doing?

This is a Smalltalk programmer's first brush with a bigger issue: How do you display an object as a string? Turns out this is not a very simple issue. VisualWorks gives you four different ways to display an object as a string: *printString*, *displayString*, *TypeConverter*, and *PrintConverter*. Why does there need to be more than one way? Which option do you use when?

This article is in two parts. This month, I'll talk about *printString* and *displayString*. In September, I'll talk about *TypeConverter* and *PrintConverter*.

printString AND displayString

There are two messages you can send to an object to display it as a string:

- *printString*—Displays the object the way the developer wants to see it.
- *displayString*—Displays the object the way the user wants to see it.

printString is as old as Smalltalk itself. It was part of the original Smalltalk-80 standard and was probably in Smalltalk long before that. It is an essential part of how Inspector is implemented, an inspector being a development tool that can open a window to display any object. An inspector shows all of an object's slots (its named and indexed instance variables); when you select one, it shows that slot's value as a string by sending the slot's

value the message *printString*. The inspector also shows another slot, the pseudovisible *self*. When you select that slot, the inspector displays the object it's inspecting by sending it *printString*.

displayString was introduced in VisualWorks 1.0, more than 10 years after *printString*. *displayString* is an essential part of how *SequenceView* (VisualWorks' List widget) is implemented. The list widget displays its items by displaying a string for each item. The purpose of this display-string is very similar to that of the print-string, but the results are often different.

printString describes an object to a Smalltalk programmer. To a programmer, one of an object's most important properties is its class. Thus a print-string either names the object's class explicitly (a *VisualLauncher*, *OrderedCollection* (#a #b), etc.) or the class is implied (#*printString* is a *Symbol*, 1/2 is a *Fraction*, etc.). The user, on the other hand, couldn't care less what an object's class is. Because most users don't know OO, telling them that this is an object and what its class is would just confuse them. The user wants to know the name of the object. *displayString* describes the object to the user by printing the object's name (although what constitutes an object's "name" is open to interpretation).

STANDARD IMPLEMENTATION

The first thing to understand about *printString* is that it doesn't do much; its companion method, *printOn:*, does all of the work. This makes *printString* more efficient because it uses a stream for concatenation.¹ Here are the basic implementors in VisualWorks:

```
Object>>printString
| aStream |
aStream := WriteStream on: (String new: 16).
self printOn: aStream.
^aStream contents

Object>>printOn: aStream
| title |
title := self class name.
```



```
aStream nextPutAll: ((title at: 1) isVowel
  ifTrue: ['an '] ifFalse: ['a ']).
aStream print: self class
```

```
nextPutAll: ' ',self name]
```

displayString is not implemented as gracefully as printString. Rather than using a two-step process and a stream, displayString is a single method that returns a string. By default, that string is the object's print-string:

```
Object>>displayString
^self printString
```

Ideally, displayString should be implemented using displayOn:, but that message already has a different meaning in the VisualComponent hierarchy. However, those methods in the VisualComponent hierarchy would be better named "displayWith:," which more accurately describes what the method does. This would then free up displayOn: to be implemented to add an object's name onto a stream. Until displayString is implemented this way, subimplement displayString in your own classes.

displayString is a VisualWorks convention that the other Smalltalk dialects do not have. However, as you can see, its implementation is very simple, so you can easily add it to your VisualSmalltalk or IBM Smalltalk image if you'd like to.

YOUR IMPLEMENTORS

You should never implement printString in your own class (even though ParcPlace did in HelpPage and HelpSeeAlso). However, you will often want to enhance the string it produces; do so by subimplementing printOn:.

Your implementors of printString should always specify the object's class. Furthermore, it should tell the developer which instance of that class it is. To do this, printString (implemented in printOn:) should print out one or more of the object's identity variables. Identity variables are one of the types of instance variables I described in my previous article.² The values in an object's identity variables identify which instance it is and rarely change. They are the keys used to find that object in a dictionary or a database. By printing the identity variables, you're telling the developer which instance this is. If he wants to see its status and cache variables, he can use an inspector. If printOn: needs to print out a variable that's not a string, it should send that variable printString or displayString.

Cursor has a good example of printOn:. A Cursor has a name aspect to identify which cursor it is. Thus its printOn: method looks like this:

```
Cursor>>printOn: aStream
self name == nil
ifTrue: [...]
ifFalse: [aStream
  print: self class;
```

Basically, the cursor prints its class and its name (separated by a space). That tells the developer this is a Cursor and which one it is.

Your implementors of displayString should never specify what the object's class is, but they should specify which instance it is. displayString does this by printing one or more of the object's identity variables. Many objects don't have any identity variables. In these cases, there probably is no good way to display this object to the user. In such a case, just inherit Object>>displayString and avoid using it.

Remember that printString is how you want this object to appear in an inspector to a developer; displayString is how you want it to appear in a list widget to a user.

AN EXAMPLE

Let's say you're implementing the class Person. It has an aspect, name, which is an instance of PersonName. The classes will be subclassed from Object. This means that their print- and display-strings will be "a Person" and "a PersonName." This is of limited use in an inspector; worse, a selection-in-list for a collection of Persons will list "a Person" in every slot.

Here's how we could implement printString (via printOn:) and displayString to make them more useful:

```
Person>>printOn: aWriteStream
super printOn: aWriteStream.
aWriteStream
  nextPutAll: ' ';
  nextPutAll: self displayString
```

```
Person>>displayString
^self name displayString
```

```
PersonName>>printOn: aWriteStream
super printOn: aWriteStream.
aWriteStream
  nextPutAll: ' ';
  nextPutAll: self displayString
```

```
PersonName>>displayString
^self lastName, ' ', self firstName
```

The results for a person named "John Smith" are shown in Table 1.

Note that implementing printString to send the message displayString is somewhat unusual. However, I find it to be a simple and convenient example of reuse for many objects.

<i>Method</i>	<i>Default Output String</i>	<i>Custom Output String</i>
Person>>printString	a Person	a Person: Smith, John
Person>>displayString	a Person	Smith, John
PersonName>>printString	a PersonName	a PersonName: Smith, John
PersonName>>displayString	a PersonName	Smith, John

Table 1. The strings produced by printString and displayString.

This can have adverse consequences in ENVY since `Object>>printString` and `Object>>displayString` are defined in separate applications, Kernel and WindowSystem, respectively. Thus in ENVY, your applications that contain implementors of `printString` that use `displayString` may need to have WindowSystem—and thus Kernel as well—as prerequisites. Specifically, the implementors of `displayString` that `printString` uses must be in the prerequisites; luckily, `Object>>displayString` is usually not one of them. Setting up the prerequisites is usually not a problem for Application Model applications, but can be a problem for Domain Model applications, because they should not have WindowSystem as a prerequisite. If this is a problem for your code, the solution is to modify the OTI applications to move the necessary implementors of `displayString` from WindowSystem to Kernel. (Or you can ignore the problem because you probably won't use the image without a windowing system anyway!)

printString SHOULD NOT FAIL

Sometimes `printString` fails and issues an error notifier. This is *really* annoying. Often during development, you have an object that is not working correctly. As you inspect it to figure out why, you keep getting message-not-understood errors saying that `UndefinedObject` does not understand some message. This really limits the usefulness of the inspector!

One way to get around this problem is to have your implementors of `printOn:` check each variable before using it. Only print out a variable if it's not nil. However, checking for nil all of the time is tedious. Even if the variable is not nil, it may still be of the wrong type (which would explain why the object is not working correctly). But since the variable's value is the wrong type, it probably won't understand the messages `printOn:` sends to it, so `printOn:` will still fail.

Another tactic is to only send the variables messages that all objects understand. If you only send messages like `printString` to a variable, the message is guaranteed to work no matter what the variable's value is. However, if your implementor of `printOn:` contains a bug, it will fail and fixing the bug will be frustrating.

The universal way to prevent `printString` from failing is to have it trap errors and handle them. You can trap all errors by implementing `printString` like this:

```
Object>>printString
| aStream |
aStream := WriteStream on: (String new: 16).
Object errorSignal
handle:
[:ex |
aStream
reset;
nextPutAll: 'an invalid ';
print: self class.

ex return]
do: [self printOn: aStream].
```

^aStream contents

This way, if `printOn:` fails, the error handler will print out the name of the class and say that the instance is invalid. At this point, you can inspect the object to see why it is invalid. I think that is a lot better than getting an error notifier.

You may want to make this modification in your image. This will require modifying ParcPlace's `Object>>printString` method. You should usually avoid modifying vendor code, but in this case I think doing so is the best solution.

displayString AND asString

A common problem with using strings is that string concatenation (implemented in VisualWorks by `SequenceableCollection>>`.) is not very polymorphic (nor should it be). If the concatenation argument is nil, a `Character`, an `Exception`, or some other nonstring-like object, Smalltalk will issue an error. To avoid this problem, developers routinely send an object `printString` before concatenating it. But `printString` does a lousy job of printing the object for concatenation: strings have quotes around them, symbols have pound-signs in front of them, most objects are called "an Object," etc.

To do a better job of printing an object out so that it can be concatenated onto a string, many developers use `asString`. They implement `Object>>asString` to define the standard protocol, then implement `asString` in all kinds of classes as they find objects that don't convert "correctly." I contend that this is a haphazard way to program and overloads ParcPlace's original `asString` protocol. `asString` is a message VisualWorks uses for converting a string-like object (such as a symbol, text, or filename) into a `String`. If an object is not at all string-like, it really has no clear implementation for `asString`.

Instead, I think that `displayString` is the solution developers are looking for. Both `asString` and `displayString` return strings. Neither message puts any junk in the string to specify the object's class. The main difference is that `asString` is an "as..." message. This implies that the receiver can be (and will be) converted to a `String` equivalent. `displayString` makes no such promises of equivalency; it simply says it will display the object as a string that describes the object.

Thus I recommend implementing `displayString` for any object you need to concatenate onto a string. Implementors you might need are:

- `UndefinedObject>>displayString` should return an empty string;
- `Character>>displayString` should return a one-character string;
- `CharacterArray>>displayString` should be reimplemented as "`^self asString displayString`."

I think this policy will be more consistent and easier to reuse than random implementors of `asString`.


CONCLUSIONS

Here are the main points in this article:

- `printString` displays an object the way a developer would

describe it. It specifies the object's class and specifies which instance the object is by displaying one or more of its identity variables.

- `displayString` displays an object the way a user would describe it. It does not specify the object's class because users never do. It specifies the object's name, that being one or more of its identity variables.
- In VisualWorks, don't subimplement `printString`; subimplement `printOn:` instead. Do subimplement `displayString`.
- Consider reimplementing `Object>>printString` with an error handler so that it cannot fail.
- Do not implement `Object>>asString` or most other implementors of `asString`. Use `displayString` instead.

In the next article, I'll talk about `TypeConverter` and `PrintConverter`. 

References

1. Woolf, B. "A Sample Pattern Language: Using Streams for Concatenation," *Smalltalk Report*, Feb. 1995.
2. Woolf, B. "A Strategy for Using Instance Variables," *Smalltalk Report*, June 1996.

Bobby Woolf is a Senior Member of Technical Staff at Knowledge Systems Corp. in Cary, NC. He mentors Smalltalk developers in the use of VisualWorks, ENVY, and Design Patterns. Comments are welcome at woolf@acm.org or at <http://www.ksscary.com>.

Smalltalk SQA: The Public/Private Problem #2

Jeff McKenna

IN THE PREVIOUS article of this series, we opened up a discussion of testing issues when using Smalltalk. We discussed aspects of GUI and Model testing, and we provided a definition of a software component as code, documentation, and tests. In this article, we discuss the testing of these components in detail, including the role of regression testing.

In discussing components, it is very important that we include the ability to construct complex components as aggregations of simpler components, because this is very common and allows us to define components in a recursive manner. Our prior discussion of a software component emphasized the role of the interface in the definition of the component. The interface must be supplied for an aggregate or complex component, as well as a simple component. This brings us to a problem that I refer to as the Public/Private Problem.

In Figure 1, we show a simple diagram of three classes, A, B, and C, some numbered methods, 1 to 8, and two message sends from outside. Note that the method numbers are arbitrary.

Smalltalk currently defines the interface through the public/private "attribute" of the methods. In Figure 1, methods 7, 1, 4, 3 and 6 would be considered public if the two message sends were the only way the classes are used.

Now consider Figure 2. In this figure, we have constructed two components, I and II, from the supplied classes.

Note that from the point of view of component I, the interface is 1 and 6. From the point of view of component II, the interface is 6 and 7.

Let's look at Class A methods.

Method 2

Class A Private
Component I Private
Component II Private

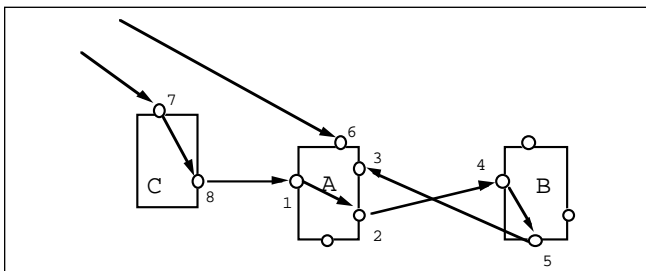


Figure 1.

Method 4

Class A Public >>A(4)
Component I Private
Component II Private

Method 1

Class A Public >>A(1)
Component I Public >>I.A(1)
Component II Private

Method 6

Class A Public >>A(6)
Component I Public >>I.A(6)
Component II Public >>II.A(6)

We are using a simple dot notation to indicate the intersection of the method and the component:

<component>.<class>(<selector>)

From this diagram it is easy to see that the public/private attribute of a method is not a useful construct in determining the interface of a component. Each component must define its own public methods (i.e., its interface). We have found that the identification of this interface is critical to the building of reusable components.

None of the currently available code control systems for Smalltalk support this view of the interface definition of a component. At most they support the public/private attribute of a method. This situation makes it difficult to adequately specify and test a component. I would like to encourage the vendors to add such support to their tools. Such support would move us a long way in the direction of being able to clearly define components.

Note that if the packaging changes, then the interfaces may change fairly dramatically. For example, if compo-

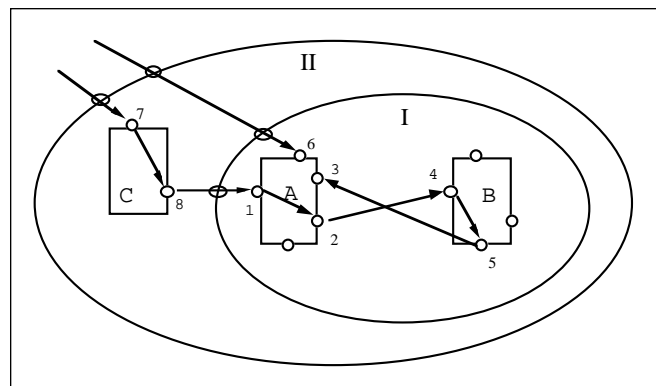


Figure 2.

nent I is changed to be classes A and C rather than classes A and B. As we would expect, the interfaces of II remain II.C(7) and II.A(6). The interface of I, however, changes noticeably. I.C(7) and I.A(4) are added to the interfaces, I.A(6) does not change its status, and I.A(1) is no longer part of the interface.

We can make one further observation regarding aggregate components. If a method defined in a component, C, is private, then it must remain private in any aggregate component that contains the component C. This restriction is often violated, as “interesting” methods are discovered deep within a component structure. One might consider such usage as behavior “leakage,” which is analogous to memory leakage. The tool support requested above would make it possible to detect such leakage.

This view of the public/private problem also solves the difficult problem of considering methods in a class hierarchy. If a component, CA, holds an abstract class, A, and another component, CS, holds a concrete subclass, S, then how do we handle methods that are defined in the class A but only used within the class S? Our view is that all such methods must be considered public to the component CA, just as any methods in S that are required in A to “complete” the abstract class should be considered public to the component CS.

All of this is a bit easier to think about if we just consider a class as a component consisting of a bunch of methods. Thinking of classes in this way also makes it easier to consider loose methods (class extensions).

TESTING COMPONENTS

With this enhanced definition of a component and its interface, it is now possible to discuss how we might test a component.

Testing only the interface, with no knowledge of the internals, is called black box testing. Most practitioners consider black box testing insufficient because it is typically impossible to test all possible states. Testing with knowledge of the internals is called white box testing. A suggestion has been made that testing the interface with a “little” knowledge of the internals should be called gray box testing!

The decision as to the type of testing (i.e., black, gray, or black box) depends on the testing job being performed. Let us consider these three major tasks performed with tests:

- Unit Testing
- Interface Testing
- Aggregation Testing

Unit testing is typically performed by the developer and should verify that the component functions as designed. This usually means white box testing or, at least, gray box testing.

Interface testing is a term I use to mean testing only the interface. It is important that interface tests are provided to support consolidation and redesign activities. SQA should ensure that interface testing completely exercise the interface of the component.

Aggregation testing is the term that I use when testing aggregations or complex components. In aggregation testing, interface testing of the subcomponents is followed by the unit testing of the component itself. In other words, first test if the pieces still work and then determine if they are working together correctly; bottom-up testing as it were.

In our example, here is the testing sequence for component II:

Unit	Class A
Unit	Class B
Interface	Class A*
Interface	Class B*
Unit	Component I*
Unit	Class C
Interface	Component I**
Interface	Class C**
Unit	Component II**
Interface	Component II

The asterisk indicates the aggregation test of component I and double asterisks indicate the aggregation test of component II.

Of course, in practice, distinctions are never this clear. However, they should be considered when considering the efficacy of testing.

The above sequence works fine when fixing bugs and when adding functions. For each version, existing tests are used as is or are expanded to test for bugs and the new functions.

This is classic regression testing, which can be automated. Automation helps keep systems “no worse” than they were in the prior build. Regression testing in Smalltalk systems appears to have more value than regression testing in classical software development. While I am not exactly sure why, I suspect that it is because inheritance and the distributed nature of Smalltalk systems make the impact of change more difficult to predict. The developer doesn’t know the whole system.

Regression testing does not work as well during consolidation or refactoring. Say component I is significantly changed so that it no longer use classes A and B, but rather uses classes X and Y. Call this new component I’. Also assume that the interface to component I’ remains unchanged from I. The test sequence for component I’ is now

Unit	Class X
Unit	Class Y
Interface	Class X
Interface	Class Y
Unit	Component I’
Interface	Component I’


The key to note is that the last test is the same as in the original testing. In other words, Interface(I) is the same test as Interface(I’), because the interface has not changed. If unit and interface testing are combined, as many folks do, then this is not true; Unit(I’) is clearly not the same as Unit(I).

In practice, this means that if the tests are not separated into the unit and interface components, tests have *no* utility in verifying if the new version can replace

the old. In practice this happens all the time. The developer changes class A, changes the class A tests to reflect the change, and then is puzzled when someone else's use of class A breaks. This is because the developer changed the interface and then just changed the tests to match.

Good design and the enforcement of interface contracts reduces the exposure to this type of trouble. Good interface tests can be used to ensure that interface contracts are kept up.

As more software is developed by top-down construction, combining existing components in new ways, the importance of interface testing becomes greater. No longer can the developer of a component use SENDERS to find all the clients of that component. In the extreme view, if the interface changes in any way it is really a new component. New and improved perhaps, but still a new component.

This article has explored the public/private, proposed a solution, and then used that solution to define unit, interface, and aggregate testing. In our next article, we will discuss roles in the testing process with a particular focus on changes in time. 

Jeff McKenna is the founder and President of MCG Software, Inc., Wilsonville, OR. MCG Software offers testing frameworks for Smalltalk. Jeff has been involved with software for more than 33 years and been involved with Smalltalk since 1982. He was chairman of OOPSLA '94. He may be reached at mckenna@acm.org.

Proper use of class methods

Jill Nicola

THE BIGGEST BLUNDER most new Smalltalk developers make is improper use of class methods. Many a new Smalltalk convert has implemented major portions of a system design using class methods. Well, why not? They are easy to access—callable from anywhere by any object.

Tempting, but not a good idea. First, consider your system design documents. There is no place on an object model for class methods. When you build an object model of your system design, the services in a class symbol map to instance methods, not class methods. For example, consider the simple class symbol for a product vendor as shown in Fig. 1:

Every product vendor has its own name, address, and list of products offered. These attributes that every product vendor possesses translate into Smalltalk instance variables defined in the ProductVendor class. Every product vendor can be asked to provide a quote for a given quantity of product. This is a behavior every product vendor provides, so it translates into an instance method defined in the ProductVendor class.

Now at some time during system execution all the product vendor objects must be loaded into memory. One approach is to write a class method, loadAll, in the ProductVendor class. This class method goes out to the database and reads all the product vendor objects. A class method is used so it can be called by a human interface screen or some other process. Great, but where does it go on the object model? To show it on the class symbol would imply it is an action every product vendor performs—an instance method (see Fig. 2).

Faced with this problem, most developers invent an extension to the notation or discover within the notation some obscure demarcation for distinguishing class

methods from instance methods. Oh boy. If you follow this approach, you have an object model with notation only you understand that is implemented using global functions. That's right, global functions.

Your design is much less object oriented with class methods. Ideally, you want *all* the functionality in your system to be implemented by *objects* that represent things in the real world, or correspond to system components. Class methods are a fluke, a side effect of the fact that most object-oriented languages need a data structure to serve as the definition for producing objects, and this data structure, typically called a class, needs to have behavior so it can produce objects. Making use of this fluke is just an acknowledgment that you have system functionality you cannot associate with any object in your design . . . global functionality.

Another problem arises when requirements change. Suppose you want to use the ProductVendor class in two different applications—one with an ORACLE database, one with a OODBMS. The loadAll method would have to be written differently for each application. In fact, every time the data representation requirements changed you would need a new version of the loadAll method. Ouch.

So to recap, the problems with class methods are: 1) Class methods require new or obscure notation on the object model. 2) Class methods make your design less object oriented. 3) Class methods decrease the flexibility of your system to changes in requirements.

What to do? What to do? First, learn to recognize that wherever class methods abound . . . objects are missing. In the example, a loadAll class method was used because there was no object in the system design responsible for loading product vendors. So add one (see Fig. 3).

Heck, add a bunch... ProductVendorOracleDBA, ProductVendorOODBMSDBA. However many you might need, just get that loading behavior out of the class method on your business domain object and get it into an instance method on a separate object. Now in all your future applications whenever product vendors need to be loaded, create a product vendor DBA of the proper class, hook it to the server, and tell it to loadAll. Gee, you could even document the loading procedure with an object interaction or a scenario diagram, something that would be really hard if you were using class methods.

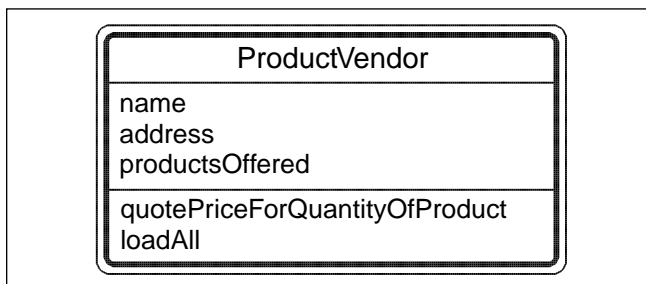


Figure 1. Simple class symbol.

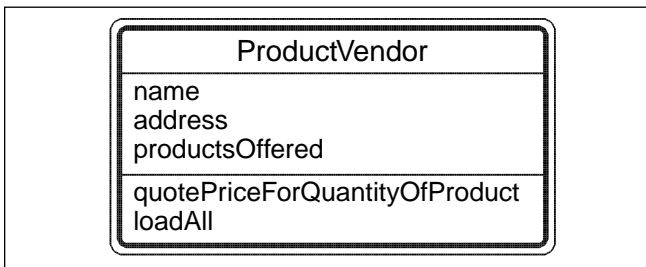


Figure 2. An Instance method.

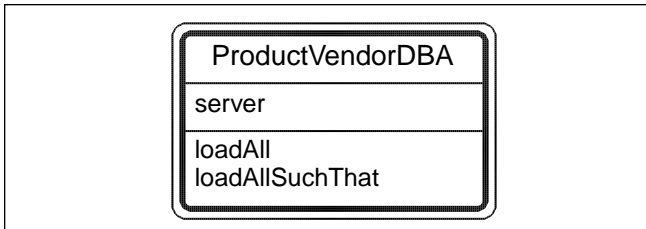


Figure 3. Object for loading product vendors.

So, system functionality in class methods, particularly in a business domain class, indicates objects are missing. Here's when you should use class methods:

1. *Creating new objects.*

```
Product new
Date today
Time now
perot
Array new: 10
```

Messages `new`, `today`, `now`, `perot`, `new:`, correspond to class methods that create new objects. The `new` class methods typically create objects with `nil` or default initial values in the instance variables. The other methods, such as `today`, and `now`, create special objects in the class that have instance variables preset to meaningful values.

- #### 2. *Managing class variables.* Use class methods to:
- Initialize, reset, flush class variables.
 - Provide read and write access into class variables.

Wow. That's a short list. Well, there is one other time you might consider class methods, but it is for development purposes not system design.


3. *Creating example or test objects.*

A test object has in its instance variables typical data values that would exist during a normal system execution. Test objects are a great help during development because they allow developers to run portions of the system without having to load data, guess at representative data values, or keep workspaces open with scripts for building objects. Here's how a typical test object method looks. Note, how the executable comment within the method makes it easy to run.

ProductVendor class methodsFor: 'examples'

```
testObject
    "ProductVendor testObject"
    | vendor |
    vendor := self new.
    vendor name: 'Vendor X'.
    vendor address: Address testObject.
    vendor addProductOffered: Product testObject.
    vendor addProductOffered: Product testObject2.
    vendor addProductOffered: Product testObject3.
    ^vendor
```

Now do not go off and implement elaborate test scenarios with a slew of class methods; you will be making the *missing object* mistake all over again. Test scenarios will vary from application to application, so build separate objects to implement your testing procedures. Test objects are essentially the unit tests from which all test procedures are built.

To conclude, guard against class methods creeping into your design. Designs using class methods are not easily represented within an object model, are less object oriented and more functional, and are brittle to changes in requirements. Where class methods abound, objects are missing. Use class methods for creating new objects, managing class variables, and building test objects. 

Jill Nicola is President of JEN Consulting, which offers Smalltalk training, consulting, and mentoring services. She specializes in architecture design and customized GUIs. She can be reached by email at nicola@jencon.com.

Implementing mixins in Smalltalk

Terry Montlick

GRADY BOOCH DEFINES a mixin as:

“A class that embodies a single, focused behavior, used to augment the behavior of some other class via inheritance; the behavior of a mixin is usually orthogonal to the behavior of the classes with which it is combined.”¹

There are always situations in which “mixing in” another class is the most straightforward thing to do. But mixins are generally thought of as requiring multiple inheritance,² which is the mechanism by which languages like CLOS supply them. Smalltalk, however, does not have multiple inheritance. Furthermore, multiple inheritance raises a host of issues, such as repeated inheritance—where the same superclass can be reached via more than one route through the parent hierarchy.

Instead, I’m going to take a simpler, more pragmatic approach. In this implementation, mixins are provided to a class by adding a message, `mixins`, which replies with a collection of mixin objects. If an object which uses mixins does not understand a particular message, then each object in the `mixins` collection is tried, in order. Technically, this is not true multiple inheritance because it provides only “interface inheritance” and not “class inheritance.”

In order for this to work, the `Object` class must be modified.* Ordinarily, I am loath to do this, but this change is very simple and foolproof. Honest.

First, change the name of the existing `doesNotUnderstand:` method in the `Object` class to `originalDoesNotUnderstand:`. Then, add the following method:

```
doesNotUnderstand: aMessage
    "If the object has mixins, see if one responds to
    aMessage."
    (self class canUnderstand: #mixins) ifTrue: [
```

* You could alternatively create a class that overrides the `doesNotUnderstand:` message selector of the `Object` class, and always subclass from this. However, this restricts the utility of mixins, since you would not then be able to “mixin” to an existing class hierarchy.

```
self mixins do: [ :mixin |
    (mixin class canUnderstand: aMessage selector)
    ifTrue: [
        ^mixin perform: aMessage selector
            withArguments:
                aMessage arguments
    ].
].
].
].
"was not handled by mixin, so pass to original
doesNotUnderstand handler"
^self originalDoesNotUnderstand:
    aMessage
```

Multiple inheritance raises a host of issues, such as repeated inheritance.

The operation of this method is very simple. If the class of the object that generated the `doesNotUnderstand:` message selector understands the `mixins` message, then this message is sent to it. The reply is some kind of `Collection`, which is sent the `do:` mes-

sage selector with a block argument. This block argument tests to see if an element of the `Collection` understands the message selector that the original object did not understand. As soon as such an element is found, it is sent the message selector, along with the original arguments.

The mixins may be put in any type of collection. A simple `OrderedCollection` may be used as the `mixins` object. A dictionary might also be used to gain named access to the individual mixin objects.

Here is a simply and highly artificial, but illustrative, example of using mixins. A class, `MixinTest`, has a single instance variable called `mixins`. It has the following accessor:

```
mixins
    ^mixins
```

and the following initialize method:

```
initialize
    mixins := OrderedCollection new.
    mixins add: (Date today).
```

The initialize method sets the mixins instance variable as an OrderedCollection with a single element, a Date, which is today's. This initialize method is sent by the instance creation method:

```
new
  ^super new initialize
```

Inspecting the following statement causes the current day of the month to be displayed:

```
MixinTest new dayOfMonth
```

Additional levels of mixins can be added. Consider a class Mixin2Test, which is a subclass of MixinTest. This class also has the instance creation method:

```
new
  ^super new initialize
```

but the initialize method is

```
initialize
  super initialize.
  mixins add: (Point x: 100 y: 50)
```

By calling super initialize, the MixinTest object creates the mixins instance variable and adds a Date object to it. The Mixin2Test initialize method then adds a Point object to mixins.

A Mixin2Test object now understands messages for two mixin objects. For example:

```
Mixin2Test new dayOfMonth
Mixin2Test new x
```

An object of any class may act as a mixin. However, an abstract mixin class is particularly powerful. An abstract mixin is a class that cannot be instantiated in isolation, because it requires another object or objects to provide methods for it.

I'll call the object that a mixin is added to the root object. A class for subclassing abstract mixin classes is called AbstractMixin. It has a single instance variable called root, which has the accessors:

```
root
  ^root

root: anObject
root := anObject
```

It also has the class-side instance creation method:

```
root: anObject
  ^self new root: anObject
```

Subclasses of AbstractMixin must declare their root object when they create a new instance. The root object is implicitly referred to by sending messages of the form:

```
self root <some message for root object>
```

Every AbstractMixin subclass should clearly document what methods it requires its root object to provide.

As an example of abstract mixins, I'll use one given by

Seidewitz.³ The abstract mixin class InterestMixin is a subclass of AbstractMixin and provides the incremental functionality of earning interest. It has a single-instance, variable, rate, with the usual accessors, plus a method for computing interest earned:

```
interestEarned: dt
  "root must provide a 'balance' method"
  ^root balance * rate * dt
```

An instance creation method sets both the rate and the root object:

```
rate: aNumber root: anObject
  ^self root: anObject rate: aNumber
```

Another mixin class, this time a concrete mixin called AccountMixin with an instance variable called balance, provides basic account functionality:

```
deposit: aNumber
  balance := balance + aNumber
```

```
withdraw: aNumber
  balance := balance - aNumber
```

An instance creation method sets the current balance:


```
balance: aNumber
  ^self new balance: aNumber
```

Now, I'll put these together in a new class called SavingsAccountImplementation, which has an instance variable mixins with an accessor of the same name. All that is necessary is add the instance initialization method:

```
balance: aBalance rate: aRate
  mixins := OrderedCollection new.
  mixins add: (AccountMixin balance: aBalance).
  mixins add: (InterestMixin rate: aRate root: self).
```

and the following class-side instance creation method,

```
balance: aBalance rate: aRate
  ^self new balance: aBalance rate: aRate
```

That's all there is to it! The AccountMixin object provides the implementation of the balance method, which is required by the InterestMixin. SavingsAccountImplementation does not have to supply anything. 

References

1. Booch, G. *Object-Oriented Analysis and Design with Applications*, 2nd ed. Benjamin/Cummings, Menlo Park, CA, 1994, p. 515.
2. Gamma, E., et al. *Design Patterns*. Addison-Wesley, Reading, MA, 1995, p 16.
3. Seidewitz, E. "Controlling Inheritance," *Journal of Object-Oriented Programming* 8(8), Jan. 1996.

Terry Montlick is the founder of Software Design Consultants, which specializes in state-of-the-art Smalltalk projects. He can be reached by email at 75260.2606@compuserve.com or at <http://www.softdesign.com/softinfo/sdc.html>.



Jan Steinman



Barbara Yates

Smalltalk as an Internet server

IN THE JUNE ISSUE, we demonstrated how to turn arbitrary *Text* objects into HTML, and in September 1995, we demonstrated how to modify VisualWorks under ENVY so you could store all your commentary in styled text. The only thing missing to have live web access to your Smalltalk project documentation is a server!

We've never been fond of specific solutions to general problems. It would be easy to hard-code a server that is dedicated to serving HTML versions of Smalltalk documentation, but there is so much code that is common to *any* server that we couldn't ignore the potential reuse.

For example, a server of any kind has these needs:

- Manage a socket name space — you can't simply pick any number for your socket.
- Initialize a socket and prepare it for use.
- Loop forever, waiting for connection requests.
- Record service requests in a log.
- Screen service requests for security reasons.
- Fork off individual service requests, so the main loop isn't unduly delayed from its primary task of waiting for connections.
- Manage unexpected problems that might occur in a service request.
- Perform the requested service, and return a result.

The following TCP server framework for VisualWorks hides all but the last step, allowing the server author to concentrate on the actual service being provided, without being distracted by the mechanics of managing sockets, processes, logs, and exceptions.

SETTING UP A SERVER INSTANCE

Our server is defined as:

```
Object subclass: #TcpServer
  instanceVariableNames: 'port socket server service
    handler requests logger logProtect'
  classVariableNames: 'CanTalkToBlock'
```

Jan Steinman and Barbara Yates are co-founders of Bytesmiths, a technical services company that has been helping companies adopt Smalltalk since 1987. Between them, they have more than 22 years of Smalltalk experience. They can be reached at Barbara@Bytesmiths.com or Jan@Bytesmiths.com, or via <http://www.bytesmiths.com>.

```
DefaultHandlers DefaultServices Portmap
PortmapControl '
poolDictionaries: "
```

Instances of *TcpServer* provide stateless services on Transmission Control Protocol (TCP) Internet domain sockets. Each instance is uniquely associated with a port number on a given machine, which must be supplied when creating an instance. Because port must be unique, we use a class *Portmap* registry to maintain this uniqueness. Because this registry will be accessed from multiple threads of control, it must be protected by a mutual exclusion mechanism in *PortmapControl*. We set all this up when initializing the class, which also sets up the default security and the *DefaultHandlers* and *DefaultServices*.

TcpServer class:

initialize

"Set up long-term state that is used for instance management."

self beSecure.

"If this is a re-initialize, be thread-safe."

PortmapControl == nil ifFalse: [self shutDown].

PortmapControl := Semaphore forMutualExclusion.

Portmap size > 0 ifTrue:

[Portmap copy do: [:server | server terminate]].

Portmap := IdentityDictionary new.

(DefaultHandlers := IdentityDictionary new) at: 0

put: self nullHandler.

(DefaultServices := IdentityDictionary new) at: 0

put: self discardService

onPort: portNumber

"Answer an active instance that provides default services for port <portNumber>."

^Portmap

at: portNumber asInteger

ifAbsent: [(self new port: portNumber asInteger) resume]

The connection security mechanism employs a block that answers a Boolean when passed an incoming sock-

et. If the block answers false, the connection is dropped immediately. This class method sets the default connection security, but once the connection security is passed, an individual server can take extra precautions or implement finer graduations of security. We also have utility methods **beFriendly**, which allows all connections, and **beLonely**, which only allows connections from the same machine, which can be useful for testing.

TcpServer class:

beSecure

"Set the default security to only accept connections from the same network."

```
CanTalkToBlock := [:socket |
    socket getPeer networkAddress = socket
    getName networkAddress]
```

The final part of class initialization declares what to do when an instance is created for a port number that does not have a default service block or exception handler. Normally, an instance has its own service and handler. If not, a default service and/or handler is fetched from the class for a given port number. If even that fails, then the "default default" service and/or handler is used. Because zero is an illegal port number, we use it to hold the "default default" service and handler.

TcpServer class:

nullHandler

"Answer a handler for when nothing is to be done with errors. (This is generally not a wise choice!!)"

```
^[:exception :stream | ]
```

discardService

"Answer a 'discard' service, which is to be used when no service can be found for a given port."

```
^[:stream | ]
```

Finally, accessing methods for the unique identifying information for an instance must take some special actions.

TcpServer:

port

"Answer the port that is listened to by this server for requests. If none is given, answer 7, for an echo server."

```
^port ? [7]
```

port: portNumber

"Initialize me with state needed for default communication on the given <portNumber>. Answer myself."

```
self port: portNumber service: nil handler: nil
    logger: System errorLog
```

The definition of the ? method was published in our

January 1996 column. It simply answers the receiver, unless it is nil, in which case the argument is evaluated and answered. Note that we also use a few ENVY utility methods in this code, which you will need to change if you are not going to use this as an ENVY documentation server.

With one more method, we will have all the essential base state needed to implement our server. This is the primary instance initialization method.

TcpServer:

```
port: portNumber service: serviceBlock
```

```
handler: exceptionBlock logger: logStream
```

"Initialize me with state needed for a particular task. Any argument can be nil, and will be defaulted suitable for an 'echo' server that logs to the Transcript.

<portNumber> is an Integer port number to listen to.

<serviceBlock> is a one-argument block that is passed a stream on a transient socket on <portNumber> when a connection arrives.

<exceptionBlock> is a two-argument block that is passed the exception and the socket stream when <serviceBlock> has an unhandled exception.

<logStream> is place to write log messages."

```
requests := WeakArray with: 0.
```

```
service := serviceBlock.
```

```
handler := exceptionBlock.
```

```
port := portNumber asInteger.
```

```
logger := logStream ? [Transcript].
```

```
(self class register: self) ifFalse:
```

```
    [self error: 'You already have a service on this
    port! You can only have one service
    per port per machine.']
```

Remember the need to keep track of port numbers? This is handled by the class, which also needs a way to "forget" about port numbers as their server instances are released. The class also manages associations between port numbers and the services (and their exception handlers) that each port provides.

TcpServer class:

register: instance

"Register the given <instance> of myself, unless one is already registered at that port. Answer success or failure."

```
^(Portmap at: instance port ifAbsentPut:
    [PortmapControl critical: [instance]]) == instance
```

unregister: instance

"Unregister the given <instance> of myself. Don't complain if I can't find it."

```
PortmapControl critical:
```

```
    [Portmap removeKey: (Portmap keyAtValue:
    instance ifAbsent: []) ifAbsent: []]
```

defaultHandlerFor: portNumber

"Answer an appropriate handler for <portNumber>, or a default default if none."

^DefaultHandlers at: portNumber ifAbsent:

[DefaultHandlers at: 0]

defaultServiceFor: portNumber

"Answer an appropriate service for <portNumber>, or a default default if none."

^DefaultServices at: portNumber ifAbsent:

[DefaultServices at: 0]

Recall that class initialization set up a **nullHandler** and a **discardService** to be used when nothing else was specified for a given instance on a given port number. That means we need a way to associate other handlers and services with ports, so that instances can be tightly cohesive with a port number, but loosely coupled with a service and handler.

The "default default" of a **discardService** with a **nullHandler** doesn't make for a very useful server!

TcpServer class:

defaultHandlerFor: portNumber **is:** twoArgBlock

"Set the exception handler for <portNumber> to <twoArgBlock>. When evaluated, the two arguments will be:
the <exception> that was the argument to the handle: block,
a read-append <stream> on the transient socket that is being serviced.
This is not thread safe, and should not be changed by some server action."

2 = twoArgBlock numArgs

ifFalse: [self error: 'Sorry, I need a two-argument clean block here!!']

ifTrue: [DefaultHandlers at: portNumber put: twoArgBlock]

defaultServiceFor: portNumber **is:** oneArgBlock

"Set the service for <portNumber> to a clean <oneArgBlock>. When evaluated, the argument will be a read-append <stream> on the transient socket that is being serviced."

1 = oneArgBlock numArgs

ifFalse: [self error: 'Sorry, I need a one-argument clean block here!!']

ifTrue: [DefaultServices at: portNumber put: oneArgBlock]

MAKING A SERVER INSTANCE USEFUL

Although you need more code for a functional server, we now have the base state needed to create and initialize a server instance. Let's put it to work by deriving the instance state needed, such as the socket connection and input process.

The basic service and handler are usually initialized from the class registry of default services and default handlers:

TcpServer: **handler**

"Answer a two-argument block that is evaluated upon service exception. It is passed the exception and a stream. Non-local returns should not be attempted. The block answer is discarded.
If no handler exists, get one suitable for my port."

^handler ? [handler := self class defaultHandlerFor: self port]

service

"Answer a one-argument clean block that is forked upon service request. It is passed a stream on the transient socket connection. Stream closing will be handled by the evaluator. The block answer is discarded.

If no service has been set, initialize it to one appropriate for my port."

^service ? [service := self class defaultServiceFor: self port]

This class-instance collaboration might not seem necessary; in fact it isn't. However, the temporality of server instances is very different from that of port-service associations, so it's useful to bind server instances tightly to a port number, but loosely to a service.

For example, an instance that is providing World Wide Web service using Hyper Text Transfer Protocol (HTTP) is created and discarded more frequently than the binding of this service to port number 80, the default HTTP port number. This reduces coupling in the time domain, which is often overlooked by designers who concentrate on reducing behavioral or implementation coupling.

Now that we have a service and a handler, the important stuff can happen. An independent thread runs the primary server loop that waits on socket connections, checks to see if the connection is legal, then services the connection's request.

TcpServer: **server**

"Answer an unscheduled Process that listens for and dispatches service requests.

This service loop should spend most of its time waiting on a socket connection, and so has a high priority. The service it implements is typically time-consuming, and so should be forked at a low priority, which immediately allows the server to resume listening for connections."

^server ? [server := [[| connection |
(CanTalkToBlock value: (connection := self socket accept))

```

ifTrue: [self serviceRequest: connection
readAppendStream]
ifFalse: [[connection close] fork]] repeat]
newProcess.
server priority: Processor userInterruptPriority - 1.
server]

```

socket

"Answer an IOSocketAccessor that listens to my port for service requests."

^socket ?

```

[socket := OSErrHolder existingReferentSignal
handle: [:ex |
logger == nil ifFalse:
[logger cr; show: 'You appear to have
another server running on port ', self
port printString, ' on this machine.']].
ex returnWith: nil]
do: [IOAccessor defaultForIPC
newTCPserverAtPort: self port]]

```

serviceRequest: stream

"A connection has been accepted on a transient copy of my socket. <stream> is a read-write stream on that socket. Log the activity and provide the requested service in a separate thread at low priority."

```

self registerRequest: ([self serviceRequestFork:
stream]
forkAt: Processor userBackgroundPriority + 1)

```

serviceRequestFork: stream

"Provide a requested service, based on the socket <stream>, which at this point has not been read at all. If there is a problem, invoke an instance-specific handler. This method is forked at a low priority."

```

Signal noHandlerSignal
handle: [:ex | self handler value: ex value: stream]
do: [self log: 'Open connection at ',
EmTimeStamp now printString from: stream.
self service value: stream.
self log: 'Close connection at ' EmTimeStamp now
printString from: stream].
OSErrHolder errorSignal handle: [:ex | ] do:
[stream close]

```

Because individual requests are forked off, it is essential to track them down and kill them if needed, so request threads are registered in requests, a WeakArray. As these requests terminate, they are collected as garbage and removed from requests.

TcpServer:

registerRequest: serviceRequestProcess

"A connection has been accepted and <serviceRequestProcess> has been forked to deal with it. Hang onto it weakly, so it can be killed when I'm killed. When it terminates, the scavenger will remove it from the registry."

INFO@SIGS

SIGS Publications, Inc., 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

ARTICLE SUBMISSION

To submit articles for publication, please contact:
John Pugh & Paul White, Editors, 885 Meadowlands Dr. #509, Ottawa, Ontario, K2C 3N2 Canada; email: streport@objectpeople.on.ca

PRODUCT REVIEWS AND ANNOUNCEMENTS

To submit product reviews or product announcements, please contact the Editors at the address above.

CUSTOMER SERVICE

For customer service in the US, please contact PO Box 5050, Brentwood, TN 37024-5050; 800.361.1279; Fax: 615.370.4845; in the UK, please contact Subscriptions Department, Tower Publishing Services, Tower House, Sovereign Park, Market Harborough, Leicestershire, LE16 9EF, UK; +44.(0)1858.435302; Fax: +44.(0)1858.434958

SIGS BOOKS

For information on any SIGS book, contact: Don Jackson, Director of Books, SIGS Books, Inc., 71 West 23rd Street, New York, NY 10010; 212.242.7447; Fax: 212.242.7574; email: donald_jackson@sigs.com

SIGS CONFERENCES

For information on all SIGS Conferences, please contact: SIGS Conferences, 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7515; Fax: 212.242.7578; email: info@sigs.com

BACK ISSUES

To order back issues, please contact: Back Issue Order Department, SIGS Publications, 71 West 23rd Street, 3rd Floor, New York, NY 10010; Phone: 212.242.7447; Fax: 212.242.7574

REPRINTS

For information on ordering reprints, please contact: Reprint Management Services, 505 East Airport Road, Box 5363, Lancaster, PA 17601; Phone: 717.560.2001; Fax: 717.560.2063

ADVERTISING

For ad information for any SIGS publication, please contact:
East Coast/Europe: Gary Portie
Central US: Elisa Marcus
Recruitment: Michael Peck
Exhibit Sales, West Coast: Kristin Viksnins
Exhibit Sales, East Coast: Sarah Olszewski
Phone: 212.242.7447; Fax: 212.242.7574; email: sales@sigs.com
West Coast: Diane Fuller
Phone: 408.255.2991; Fax: 408.255.2992; email: dhfsigs@hooked.net

INTERNATIONAL OFFICES

SIGS Conferences Ltd., Brocus House, Parkgate Road, Newdigate, Surrey RH5 5AH, United Kingdom. Phone: 011.44.1.306.631.331; Fax: 011.44.1.306.631.696; email: 100131,3500@compuserve.com.

SIGS France, 105 rue Jules Guesde, 92532 Levallois Perret Cedex, Paris, France. Phone: +33 (1) 41 06 18 00; Fax: +33 (1) 41 06 18 19; email: 100631,1050@compuserve.com.

SIGS Conferences GmbH, Odenthaler Strasses 47, D-51465 Bergisch Gladbach, Germany. Phone: 011.49(0).2202.936.810; Fax: 011.49(0).2202.936.812; email: 100634,2070@compuserve.com.

SIGS HOME PAGE AND ONLINE MAGAZINES

Access the SIGS Home Page at <http://www.sigs.com>;
Object Currents at <http://www.sigs.com/objectcurrents>; and
The X Spot at <http://www.sigs.com/xspot>.

```
(requests includes: 0) iffFalse: [requests grow
                                replaceAll: nil with: 0].

requests
  indexOf: 0
  replaceWith: serviceRequestProcess
  startingAt: 1
  stoppingAt: requests size
```

Finally, instances need to be started, stopped, and killed. If you are using ENVY, you'll want to have an application **startUp** method that relays to `TcpServer startUp` to restart your servers, and a **shutDown** method that relays to `TcpServer shutDown` to suspend them. Also remember to have a removing method that gets rid of all instances by sending `TcpServer initialize`.

TcpServer:**resume**

```
"Begin my server."

logger == nil iffFalse:
  [logger cr;
   nextPutAll: 'Resuming service on port '; print:
               self port;
   nextPutAll: ' at '; print: EmTimeStamp now; flush].
self server resume
```

suspend

```
"Suspend my server in such a way that when it
resumes, it opens a new socket. Terminate any
active requests in process."

| count |
logger == nil iffFalse:
  [logger cr;
   nextPutAll: 'Suspending service on port ';
   print: self port;
   nextPutAll: ' at '; print: EmTimeStamp now.
   count := (requests reject: [:request | request =
                               0]) size.
   count > 0 ifTrue: [logger space; print: count;
                     nextPutAll: ' active requests
                               cancelled.'].

   logger flush].
requests do: [:request | request = 0 iffFalse: [request
                                                terminate]].
server == nil iffFalse: [server terminate. server := nil].
socket == nil iffFalse: [socket close. socket := nil]
```

terminate

```
"Terminate my server and release my state."

self suspend.
self class unregister: self.
logger == nil iffFalse: [logProtect wait. logger close].
socket := server := service := handler := logger :=
logProtect := nil
```

TcpServer class:**shutDown**

```
"Suspend all my services so that the image can be
quit and re-started."
```

```
PortmapControl critical: [Portmap do: [:server |
server suspend]]
```

startUp

```
"Re-start all my services."
```

```
PortmapControl critical: [Portmap do: [:server |
server resume]]!
```

WHAT'S LEFT?

We've run out of space, but this implementation sketch should give you enough "thoughtware" to improvise. We left out the thread-safe log interface (there are problems if multiple processes write to the global Transcript), and our implementation has more extensive logging and security features and a home page facility.

We'll leave you with a handler and a service that implement a complete telnet interface to VisualWorks using this framework, hoping it might inspire your own services.

TcpServer class:**textualHandler**

```
"Answer a handler that dumps a textual stack."
"self defaultHandlerFor: 23 is: self textualHandler"
```

```
^[:exception :stream | stream
  nextPutAll: 'Unhandled exception: '; nextPutAll:
  exception errorString; cr;
  nextPutAll: exception initialContext printStack]
```

evaluationServiceLoop


```
"Answer a service block that loops over lines of input,
evaluating each and sending back the result."
"self defaultServiceFor: 23 is: self
evaluationServiceLoop"
```

```
^[:stream | stream
  next: 6; "Discard garbage characters."
  nextPutAll: 'Smalltalk evaluation service'; cr;
  nextPutAll: 'Type "self close" to end session'; cr;

  nextPutAll: 'Smalltalk> '.
[stream
  print: (Compiler evaluate: stream nextLine for:
  stream); cr;
  nextPutAll: 'Smalltalk> '] repeat]
```

CONCLUSION

It is easier to write client-server code in VisualWorks than in C, but it is still not easy enough! With some work, you can build a framework that reduces TCP servers to one or two methods.

Next issue, we'll conclude this series by tying together this month's server framework with last month's HTML interface, and you'll have your Smalltalk project documentation on your company's Intranet! 



Jay Almarode

Communicating between sessions

IN MULTIUSER ENVIRONMENTS, users often want to communicate with each other. In some cases, they want to broadcast their message to all interested parties; in other cases, they only want to carry on a dialog with one other user. In a system that supports transactions, an application sometimes wants to be notified when changes to particular objects have been committed. A client/server system already has the infrastructure to provide these services. The client and server already have predefined communication protocols, and the server has knowledge of all the clients currently logged in. This column discusses two kinds of client-to-client communications that can be supported by multiuser Smalltalk and shows how to use them to implement concurrent processing algorithms.

In a system that supports transactions, application sessions are committing changes to objects all the time. Many times an application needs to know when another concurrent session has committed a change to specific objects of interest. For example, a stock broker application may want to trigger some activity when the price of a particular stock has changed. Or, an inventory management application may need to initiate item purchases when the inventory dips below a specified threshold. A reservation system may want to be notified when a new reservable unit becomes available. In these cases, the application does not care who made the change; it just wants to be notified that a change occurred and which objects were modified.

In GemStone Smalltalk, class `System` provides protocol to receive notification when particular objects are modified. Each session that is logged in maintains its own 'notify set'. A session can register objects of interest by placing them in its notify set. This set only exists for the life of the session; that is, it is not a persistent object, but it is maintained across transaction boundaries. An application can add a single object to its notify set by executing `System addToNotifySet: anObject` or can add a collection of objects by sending `addAllToNotifySet:.` There is also protocol to access and remove objects from the notify set.

Once objects have been added to its notify set, there

are two ways in which the session can receive notification. One way is to poll for the objects that have changed; the other is to install an exception handler. If the application installs an exception handler, it must first enable the ability to receive this error (it is not really an error, but the underlying implementation uses the error mechanism to interrupt execution). This error is enabled by sending `System enableSignaledObjectsError`. Whether polling or handling an exception, to find out which objects have been modified, the application sends `System signaledObjects`. This message returns a set of objects that have had changes committed to them and clears the signaled objects set for the next use.

The following section of code illustrates how to install an exception handler and get the changed objects:

```
"first enable the ability to be notified when
  changed objects are committed "
System enableSignaledObjectsError.

" now install an exception handler to catch the notification "
Exception
category: GemStoneError
number: (ErrorSymbols at: #rtErrSignalCommit)
do: [ :ex :cat :num :args | | changedObjects |
    " get the objects that have had changes to them
      committed "
    changedObjects := System signaledObjects.
  ].
```

When adding objects to the notify set, an application must consider what part of an object will actually be written, so that the session will be notified when a change to that object is committed. For example, an `RcCounter` object (described in an earlier column) is actually implemented as a composite object, composed of multiple subobjects that each encapsulates a numerical value. It is the sum of all values in the subcomponents that actually make up the `RcCounter`'s value. When an `RcCounter` is incremented or decremented, it is one of the subcomponents that is actually written. Consequently, to receive notification when a change to an `RcCounter` is committed, a session must place the root object and all of its subcomponents in the notify set.

Another kind of useful client-to-client communication

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@gemstone.com.

is when two sessions want to talk to one another directly and immediately. In GemStone Smalltalk, this is possible by sending a signal to another session currently logged in. A signal consists of a SmallInteger, whose meaning is agreed on by the participants in the dialog, and a sequence of bytes (a maximum of 1,023). As with the changed object notification mechanism just described, signaling is implemented using the underlying error mechanism. Consequently, a session must enable the reception of these signals by sending `System enableSignaledGemStoneSession-Error`. A session can receive signals from multiple senders, and the signals are queued in the order received.

For a session to send a signal to another session, it must identify the other session by its unique session identifier, a SmallInteger. There are a couple of ways that a user can find out about other sessions currently logged into the system. To get the session IDs of all users currently logged in, you can send the message `System currentSessions`, which returns an array of SmallIntegers. For each session ID, you can send `System descriptionOfSession: sessionId` to get back an array of more detailed information. Among the pieces of information returned by this message is the name of the host machine on which that user is logged in, and the `UserProfile` object for that user. Getting information about other users is a privileged operation, so you must have 'session access' privilege to send these messages.

Once you have the identifier of the session to which you would like to send a signal, you can send `System sendSignal: aSignalNumber to: aSessionId withMessage: aString`. The receiver of the signal executes `System signalFromGemStoneSession` to receive an array of signal information. The array is empty if no signal has been sent. If a signal has been sent, the array consists of three elements: the session id of the sender, the signal number, and the bytes.

Because signaling uses the underlying error mechanism, a receiver can install an exception handler to be triggered when a signal is sent to it, or the receiver can poll for signals.

Signals are a simple mechanism that can be used to build complex behaviors that involve more than one concurrent session. One use of signals is to coordinate sessions for implementing concurrent algorithms. Implementing concurrent algorithms with individual sessions means that you are allocating tasks among multiple processes, each with its own dedicated Smalltalk interpreter and view of the object repository. Some care must be taken to make sure that each session's transaction point of view is reasonably up to date with the others. Usually this means that a session begins a new transaction as the first step in performing its part of the concurrent algorithm.

The remainder of this column describes a simple pair of classes that can be used for implementing concurrent processing. The implementation consists of one class, called `WorkerBee`, whose responsibility is to receive commands to do work, and another class, called `QueenBee`, which sends commands to multiple `WorkerBee` objects and accumulates the results of their work. A `QueenBee` uses signals to send commands to each `WorkerBee` and

uses changed object notification to learn when each `WorkerBee` has committed its work.

The implementation of `WorkerBee` is fairly simple. Its main task is to execute a service loop, waiting for instructions from any `QueenBee`. Class `WorkerBee` defines a single instance variable, called `amountToSleep` that holds the number of seconds to delay each time through its service loop. This allows the responsiveness of each `WorkerBee` to be tuned. Note that the `WorkerBee`'s OS process and resources used can be further controlled using configuration parameters as described in an earlier column on tuning. Each time, through its service loop, a `WorkerBee` checks if a signal was received. If so, it initiates some work based on the signal number. Because the meaning of signal numbers must be agreed upon by the sender and receiver, I use a pool dictionary shared by `WorkerBee` and `QueenBee` to provide symbolic names for different signal numbers.

The pool dictionary has entries with the following meanings:

#handshake	initiate an agreement to work for a QueenBee
#freeWorker	end the agreement to work for a QueenBee
#executeString	execute the given string for a QueenBee
#commit	commit the current transaction
#abort	abort the current transaction
#terminate	terminate the service loop of the WorkerBee

A `WorkerBee` must synchronize with a `QueenBee` before it does any work. In this simple example, a `WorkerBee` and a `QueenBee` perform a handshake in the following way: A `QueenBee` sends a signal initiating the handshake. Included in this initial signal is the `QueenBee`'s name. This must be a name that the `WorkerBee` can resolve to get the `QueenBee` instance that sent the signal. If the `WorkerBee` is not already servicing another `QueenBee`, it returns a signal indicating its availability; otherwise it indicates it is busy. At this point, the `WorkerBee` is dedicated to a single `QueenBee`, waiting for commands. The implementation of the `WorkerBee`'s `serviceLoop` is as follows (simple portions of this method have been omitted for brevity):

```
method: WorkerBee
serviceLoop
```

```
"Start up a loop, waiting for instructions."
```

```
| continue queen queenSessId
| continue := true.
" worker bee loop "
[ continue ] whileTrue: [ | signalArray |
    signalArray := System signalFromGemStoneSession.
    " if no signal was sent, sleep for awhile "
    signalArray isEmpty
        ifTrue: [ System sleep: self amountToSleep ]
        ifFalse: [
            | signalNumber signalSender signalBytes |
```

```

signalSender := signalArray at: 1.
signalNumber := signalArray at: 2.
signalBytes := signalArray at: 3.

" command to execute the given string "
signalNumber = executeString
  ifTrue: [
    " only accept commands from one queen "
    (queen notNil and: [ signalSender =
      queenSessId ])
      ifTrue: [ queen addToHive: signalBytes
        _execute ]
    ].
" receive a request to work for a queen "
signalNumber = handshake
  ifTrue: [
    queenSessId isNil
      ifTrue: [
        queenSessId := signalSender.
        " resolve the QueenBee's name to an
          instance "
        queen := System myUserProfile
          objectNamed: signalBytes.
        System sendSignal: handshake
          to: signalSender
          withMessage: 'Available'
        ]
      ifFalse: [ " signal that the WorkerBee is
        unavailable "
        System sendSignal: handshake
          to: signalSender
          withMessage: 'Unavailable'
        ]
      ].
  ].
continue := signalNumber ~= terminate.
]
].

```

The implementation of the QueenBee is also fairly simple. Class QueenBee defines three instance variables: its name, an array containing the session ID of each of its worker bees, and a bag in which each WorkerBee can place the result of its work. This last instance variable, cutely named hive, will be concurrently updated by multiple WorkerBees. To avoid concurrency conflicts, this variable contains an instance of RcBag. You may recall from an earlier column, an RcBag has concurrency semantics such that concurrent adds to the bag will not conflict.

In addition, the QueenBee wants to be notified when each WorkerBee has committed the result of its work to the RcBag. To accommodate this, the QueenBee places the RcBag and its subcomponents into its notify set. Once a QueenBee has issued the command for each of its workers to do some work, it can wait for notification of changes to the RcBag to gather results. The following code listing shows the methods to add the RcBag to the notify set, and to find all WorkerBees and perform the handshake with them:

```

method: QueenBee
addToNotifySet

" put the RcBag and all of its subcomponents in the notify
set "
System addToNotifySet: hive.
hive _doSessionBags: [
: addBag : removeBag |
  System addToNotifySet: addBag.
  System addToNotifySet: removeBag.
]

method: QueenBee
getWorkerBees

"Find possible worker bees, then perform a handshake to
see if they are available for work. Set the array of worker
bee's session id's with those that are available."

| possibleWorkers |
" find all users logged in as WorkerBee "
possibleWorkers
:= System currentSessions
  select: [ :sessId |
    ((System descriptionOfSession: sessId) at: 1)
    userId = 'WorkerBee'
  ].

" initiate the handshake "
possibleWorkers do: [ :sessId |
  System sendSignal: handshake
    to: sessId
    withMessage: name.
].

workers := Array new.
possibleWorkers size timesRepeat:
[ | signalArray |
  signalArray := System signalFromGemStoneSession.
  signalArray isEmpty
    ifFalse: [
      | signalSender signalNumber signalBytes |
        signalSender := signalArray at: 1.
        signalNumber := signalArray at: 2.
        signalBytes := signalArray at: 3.

        " check if WorkerBee was available "
        (signalNumber = handshake and:
          [ signalBytes = 'Available' ])
          ifTrue: [ workers add: signalSender ]
        ]
      ]
]
]

```

The mechanisms described in this column let one session find out about and communicate with other sessions. These mechanisms provide the infrastructure to build complex applications in a multiuser environment. ■



Dwight Deugo



Wayne Beaton

Reusable components

USING A GRAPHICAL user interface builder like VisualWorks' Canvas tool and ObjectShare's Window Builder Pro, a developer avoids the tedious task of hand coding an application window's layout by using a tool that generates the code after the window is constructed visually. Using such GUI builders, developers view a window's appearance when opened and how it appears when resized before ever executing a line of the application window's "real" code. These tools permit a developer to identify what application methods should execute when specific events, such as the clicking of a button, occur in the application window. Good ones will write method stubs that developers subsequently complete to perform the required reaction to given events. Reaction methods often send messages to business objects to set or retrieve information about them and may force the window to update its views. Although these tools no longer require that developers hand code an application window's layout, they still require them to implement the interaction between the views and the underlying business domain.

The latest generation of GUI builders include tools such as VisualSmalltalk's PARTS Workbench and IBM's VisualAge Composition Editor. Both GUI builders are examples of the new Construction from Parts technology called *Visual Programming*. They enable developers to create windows and other components by assembling and connecting reusable components, also known as *parts*. Rather than hand-coding the interactions between the parts, developers make visual connections between a source part's events and another part's actions. The GUI builders still write Smalltalk code to construct the window for the developer, but, in addition to layout code, they also write code connecting events on parts to the execution of methods on others. Provided that there are good parts, developers do not need to write any Smalltalk code to develop their application windows. They just assemble them from existing parts and say "go". However, without guidelines, it is now possible to paint spaghetti instead of just writing it!

Dwight Deugo and Wayne Beaton are senior members of the development educational staff at The Object People, in Ottawa, ON, Canada. Dwight (dwight@objectpeople.on.ca) has immersed himself in objects for more than 10 years and has helped clients with their object immersions as a project mentor and as a course instructor. Wayne (wayne@objectpeople.on.ca) is the coordinator of course construction and a software developer.

The success of visual programming depends on how organizations use it and on the availability of a rich library of reusable generic and domain-specific parts. This column will focus on visual programming tips and techniques to help you become a more effective visual programmer. Future columns will cover how to manage the number of connections in your window and describe visual debugging techniques. We will also provide many examples of reusable components developed using visual programming parts and techniques, such as an advanced factory part, a broker, a marquee, and web parts. Initially, we will use examples derived from IBM's VisualAge for Smalltalk environment, but we will include examples from ParcPlace-Digitalk's next product release.

This column describes the building blocks for constructing any application window: parts and connections. As an example, we build an Action List Window with IBM's VisualAge using only those building blocks—no Smalltalk code. The window's requirements are to let a user enter any number of actions into a To-Do list and then move them to a Completed list. Our goal is to demonstrate that, when given the building blocks and good reusable components, you can do a substantial amount "programming" without writing a single line of code. Be warned that visual programming rarely, if ever, provides a complete solution.

REUSABLE COMPONENTS (PARTS)

Before one can do any visual programming, one must have access to, or must create, a number of reusable components (parts). There are two different types of parts: *Visual* and *Nonvisual*. Visual parts have visual representations and appear in a runtime application window, for example, buttons, lists, input fields, and labels. Nonvisual parts have no visual representation, such as a Printer, CD player, Ordered Collection, Variable, and any domain-specific business parts. Nonvisual parts implement objects that provide logic, storage, and resource access for your application windows. Visual and Nonvisual parts are simply assemblies of visual and nonvisual parts.

In VisualSmalltalk's PARTS, all parts in the Workbench are instances of Smalltalk classes. The part's default interface includes all the messages the Smalltalk object understands and all of the events it can trigger. In IBM's VisualAge, all parts in the Composition Editor are Smalltalk classes. The part's default interface is empty until the

developer decides what portion of the part's Smalltalk class' interface to make public.

A part's interface includes attributes, events, and actions. Attributes represent properties of a part, such as the name of an employee, that other parts access. An attribute can be any Smalltalk object, including other visual and nonvisual parts. One can initialize a part's attributes using a GUI builder's property or settings tool at development time or can access them dynamically at runtime. Actions are an operation that a part executes when events on other parts trigger them. For example, a button click event (generated when the user clicks on the button in the application window) could trigger a window's close action. Actions correspond to Smalltalk methods or code fragments. Events are signals that one part can send to another to notify it that something has occurred.

CONNECTIONS

A developer specifies relationships between parts by making connections between them. The first type of connection is an event-to-action connection. This link connects an event of one part with the execution of another part's action. When the event triggers, the action executes. The second type of connection is an attribute-to-attribute connection, which can be viewed as a two-way event-to-action connection. The change of one part's attribute (the event) triggers the setting of the second part's attribute to the same value (the action), and vice-versa.

A link, also called connection, is a type of part. Therefore, it has attributes and events. The attributes of a connection correspond to the parameters that the action at the end of the connection requires and the action's result. If an action requires no parameters, the connection has only one attribute: a result. Since actions just execute Smalltalk code, the connection stores the result object as an attribute. Since the setting of an attribute is equivalent to an event, it is possible to make a connection between the result event and other actions. One can trigger an action on another part when a previous action finishes and returns a result.

Events may or may not generate parameter values. For example, the clicking of a button only triggers a click event. On the other hand, the selection of an item in a list generates a selection event and provides the selected object as an argument for a connection to use as one of its parameters. Of course, one can change the value by making a connection to the link.

VISUAL PROGRAMMING EXAMPLE

Visual programming permits developers to quickly construct application windows provided the appropriate parts are available. Using VisualAge for Smalltalk version 3.0, we quickly constructed the "ActionListWindow" shown in Figure 1. This window allows the user to construct a list of actions to do for the current day. From that list, completed actions can be moved to a completed action list. At the end of the day, the user should have all of his or her actions in the completed actions list (ha ha)!

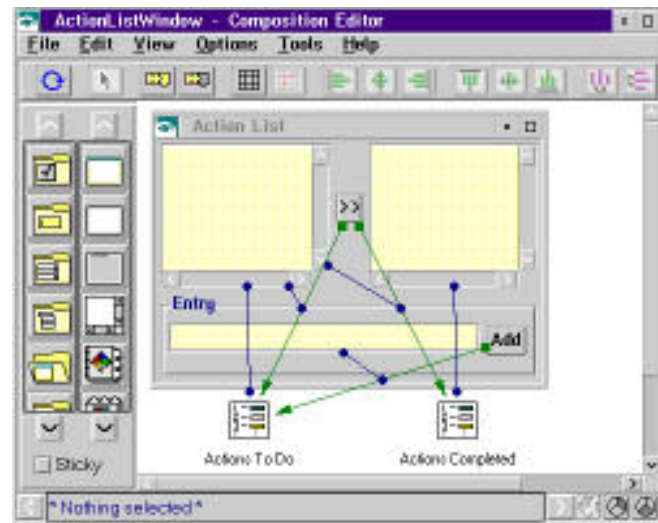


Figure 1. ActionListWindow in the Composition Editor.

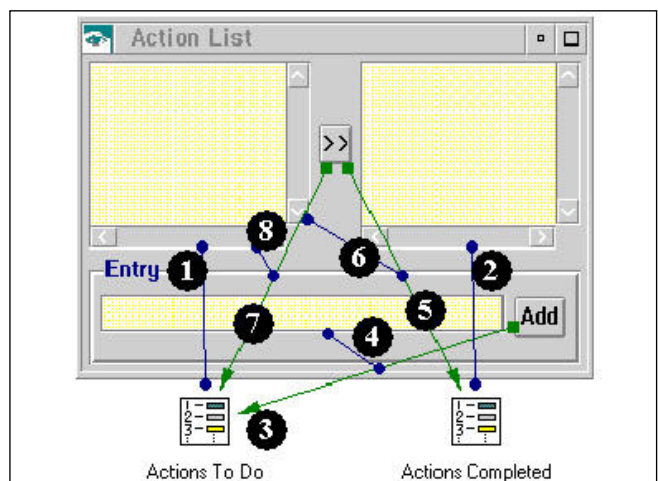


Figure 2. ActionListWindow Connections.

This first-pass of the ActionListWindow contains several visual parts, two nonvisual parts, and a few connections. The ordered collection* part, "Actions To Do," is connected to the left-most list with an attribute-to-attribute connection. The collection's "self" attribute[†] is connected to the list's "items" attribute. This connection specifies that the ordered collection stores the items to display—if the ordered collection changes in any way, the change is automatically reflected in the list box. A similar connection links the "self" attribute of the ordered collection titled "Actions Completed" to the "items" attribute of the right-most list.

The "clicked" event of the push button labeled "Add" is connected to the "add:" action of the "Actions To Do" ordered collection through an event-to-action connection. The "add:" action requires a parameter. We specify the connection parameter with an attribute-to-attribute connection from the "anObject"

continued on page 28

* An ordered collection holds any number of Smalltalk objects in the order in which they are added.

† In VisualAge, all parts have a "self" attribute. This attribute represents the whole part.

```
Link # PartName.attribute/event/action →
PartName.attribute/event/action

1. ActionList.items → 'Actions To Do'.self
2. ActionsCompletedList.items → 'Actions Completed'.self
3. AddButton.clicked → 'Actions To Do'.add:
4. inputField.object → connection3.anObject
5. >>Button.clicked → 'Actions Completed'.add:
6. ActionsCompletedList.selectedItem → connection5.anObject
7. button.clicked → 'Actions To Do'.remove:
8. ActionsCompletedList.selectedItem → connection7.anObject
```

Figure 3. ActionListWindow Legend.

continued from page 25 attribute of the original connection to the “object” attribute of the entry field. These two connections provide the ability to add objects to the ordered collection. Any objects added are automatically displayed by the connected list.

Clicking the “move” button, labeled “>>,” moves the selected item from the left-most list to the right-most one. Objects removed from the “Actions To Do” ordered collection are added to the “Actions Completed” ordered collection. The order of the following connections is important.[‡] The “clicked” event of the “move” button is connected to the “remove:” action of the “Actions Completed” ordered col-

[‡] Once you remove an object from an “Action To Do” ordered collection, it is no longer in the left-most list. Therefore, it can no longer be the selected item and cannot be moved to the “Actions Completed” ordered collection.


lection. This event-to-action connection requires an object (the object to be removed) that is supplied by connecting the “anObject” attribute of the event-to-action connection to the “selectedItem” attribute of the left-most list box. The “move” button’s “clicked” event is also connected to the “remove:” action of the “Actions To Do” ordered collection, with the “anObject” parameter supplied again by the “selectedItem” attribute of the left-most list box.

Clearly, we require a better way of describing the connections—textual descriptions are too long. A concise connection representation is both desirable and necessary. Figure 2 shows our Action List Window again, but this time we have added line labels (unfortunately VisualAge does not provide this facility for us) and Figure 3 shows the legend.

IN THE FUTURE

To keep this example small, we have avoided certain issues. The push button is not disabled when its does not apply. The “move” button should be enabled only when there is a valid selection in the left-most list box. The “Add” button should be enabled only when the user has entered data in the entry field. Perhaps some ability to remove items from one or both lists might prove useful. Ultimately, the information needs to persist in some way. These are issues we intend to address in future columns.

THE CODE

The code used in this column is available on the World WideWeb. Our URL is <http://www.objectpeople.on.ca.> 

Editors' Corner



John Pugh



Paul White

PERSISTENCE. THIS ONE word causes more grief and costs more time in terms of system development with objects than any other single issue. In theory, the problem is a very simple one. Is there a way for an application to store and retrieve the data contained within its objects?

Will the persistence mechanism also maintain the relationships between the objects in addition to the raw data? Will it support object identity properly? And how fast (slow) will it execute?

Providing a suitable answer for all of these questions is extremely difficult. Persistence has always been an issue with object-oriented languages, Smalltalk being no exception. The first persistence mechanism designed for Smalltalk used the storeOn: method to ask an object to generate a string that, when evaluated, would return the object itself. Clever idea, but not very practical. The next idea was to use a Loader/Dumper style implementation, where an entire object (and all its parts) can be stored in a file, and later retrieved. This worked well as far as it went. The shortcoming was that an object being retrieved had no notion of the relationships it used to have with other objects, and vice versa. And it certainly did not scale in terms of size or maintenance.

The correct solution, of course, is to use a true database of some flavor. Be it IMS, Oracle, Sybase, Gemstone, Versant, etc., the idea is to store and retrieve objects using a facility designed to do just that.

So, what's so hard about this? As most of you know, the difficult part is not the "data" stored with the object. It is simple to store and retrieve data such as a name, phone number and date of birth of a customer object. The difficulty comes in maintaining the relationships between the objects as designed in our systems. For example, our customer object may have a reference to an account object (or many) as well as relationships with other customers (a bank would keep track of our relationship with our spouses). The challenge is how keep track of these types of relationships? And how can we support the dynamic nature of these relationships?

The quick answer, of course, is to use an object persistence storage mechanism. Tools such as GemStone, Versant, and others handle these complex and dynamic relationships with next to no effort. There are other issues to consider when deciding whether or not to employ these new persistence mechanisms, but ease

of managing these relationships isn't one of them. However, most of us aren't in the position to entertain such a choice (and maybe we shouldn't anyway). Our organizations have invested significantly in other technologies that serve the overall organization quite well. Furthermore, since the data used by our Smalltalk applications is often shared with others, a more traditional route may be the only practical choice.

Using a relational mechanism poses some significant challenges. For example, determining how to map the relationships found in our object model to tables is nontrivial. The greater the difference between the object model to the data model, the more difficult the mappings can become. For example, many-to-many relationships (customers to accounts) require an intermediate table in the relational world, but in the object world, they're really a nonissue. Storing and retrieving objects from non-relational

mechanisms is proving to be even more difficult. Many relationships that exist in objects are virtually impossible to duplicate in, for example, a CICS transaction.

Having stated the difficulties, many have been able to successfully bridge the worlds. Most of these solutions are "home-grown." Our concern with home-grown solutions is that even once we describe how to overcome the hurdles and discover the mappings required, we're still faced with two significant challenges. First, the execution speed can often be painful (especially with writes) if not very careful about the database commands generated (e.g., SQL statements or stored procedures). Second, and more important, is the effort that is required to maintain the mappings.

This point concerning maintenance is being overlooked by far too many shops who are building their own interfaces. Many are saying "I can build it myself" and they can. But the impact on the elegance of the implementation is going to lead to systems that are tougher to extend and difficult to understand. What many seem to fail to grasp is, even if they can understand it while they're writing the code, will the person coming behind them to maintain it be able to understand? And what about the person behind them? The real costs of our solutions will be seen down the road. We strongly believe it is our job as software engineers to design systems that are easy to maintain—if we fail, we haven't done our organizations any favors.

Enjoy the issue.

*It is our job to design
systems that are easy
to maintain.*

Editors

John Pugh and Paul White
Carleton University & The Object People

SIGS Publications Advisory Board

Tom Atwood, *Object Design*
François Bancelon, *Object Technology*
Grady Booch, *Rational*
George Bosworth, *ParcPlace-Digital*
Jesse Michael Chonoles, *Lockheed Martin ACC*
Stuart Frost, *SELECT Software*
Adele Goldberg, *ParcPlace-Digital*
Thomas Kaffer, *Rogue Wave Software*
R. Jordan Kriender, *IBM Consulting Group*
Thomas Love, *Consultant*
Bertrand Meyer, *ISE*
Meilir Page-Jones, *Wayland Systems*
Bjarne Stroustrup, *AT&T Bell Labs*
Dave Thomas, *Object Technology International*

The Smalltalk Report Editorial Board

Jim Anderson, *ParcPlace-Digital*
Adele Goldberg, *ParcPlace-Digital*
Reed Phillips
Mike Taylor, *ParcPlace-Digital*
Dave Thomas, *Object Technology International*

Columnists

Jay Almarode, *GemStone Systems Inc.*
Wayne Beaton, *The Object People*
Kent Beck, *First Class Software*
Dwight Deugo, *The Object People*
Juanita Ewing, *ParcPlace-Digital*
Bob Hinkle, *Consultant*
Tim Howard, *FH Protocol, Inc.*
Ralph E. Johnson, *University of Illinois*
Alan Knight, *The Object People*
Mark Lorenz, *Hatteras Software, Inc.*
Jan Steinman, *Bytesmiths*
Rebecca Wirts-Brock, *ParcPlace-Digital*
Barbara Yates, *Bytesmiths*

SIGS Publications Group, Inc.

Richard P. Friedman, Founder, President, and CEO
Chris Keating, Publishing Director—US Magazines
John McCormick, Editorial Director

Editorial/Production

Kristina Joukadar, Senior Managing Editor
Elisa Varian, Director of Manufacturing
Jan Foster, Cover Design
Douglas Finlay, Associate/Managing Editor
Serena Tesler, Production Editor
Erika Romero, Desktop Designer
Margaret Conti, Manufacturing Coordinator

Circulation

Elayne Glick, Circulation Director
Byron Scarlett, Assistant Circulation Manager

Advertising/Marketing

Gary Portie, National Sales Manager
Elisa Marcus, Advertising Manager, Central US
Michael W. Peck, Advertising Representative
Kristine Viksnins, West Coast Exhibit Sales
Sarah Olszewski, East Coast Exhibit Sales
212.242.7447 (v), 212.242.7574 (f)
Nancy Beuschel, Promotions Manager for Magazines

Administration

Margherita R. Monck, General Manager
David Chatterpaul, Senior Accounting Manager
Bibi Budhrum, Accounts Payable



SIGS PUBLICATIONS
Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, REPORT, THE X JOURNAL, JAVA REPORT, OBJECT CURRENTS and THE X SPOT (ONLINE), OBJECT EXPERT (UK), JAVASPECTRUM and OBJEKTSPETTRUM (GERMANY)



For more object news and analysis, check out SIGS online at <http://www.sigs.com>

Table of Contents

September 1996

Vol 6 No 1

Features:

Externalizing Business-Object Behavior

4

Paul Davidowitz

One of the frameworks explored by the advanced development group at Andersen Consulting is the externalization of business-object behavior, which allows for tailoring by the end user. The behavior is represented as rule bases, which are specified in Smalltalk.

Using events for constraint solving

11

Annick Fron

Constraint-solving techniques have been commercially implemented in C++ and used for such industrial applications as train and plane scheduling. Smalltalk events allow an elegant representation of these techniques.

How to display an object as a string: TypeConverter and PrintConverter

16

Bobby Woolf

Producing an object's description is not enough. When the user types in another string, the new one needs to be converted back into an object. Here are some guidelines on how to use the TypeConverter and PrintConverter classes to display objects and read new ones from the user.

Columns



Visual programming: Managing connection complexity

19

Wayne Beaton and Dwight Deugo

What approach is one to take to minimize the connection complexity of visual parts? There must be a way to understand easily what is being built now and what has been built in the past.



Managing Objects: SmallDoc Web serving

22

Jan Steinman and Barbara Yates

Now that you know how to turn SmallDoc into HTML and how to build a generic TCP/IP server, you can now begin serving up Smalltalk project documentation to anyone with a Web server.



Getting Real: Fault tolerance

25

Jay Almarode

Here are some mechanisms to achieve fault tolerance, and how to recover when bad things happen.



Smalltalk Idioms: Object-oriented recursion

27

Kent Beck

While coming to understand recursion as a rite of passage for developers, it must be managed somewhat differently to make effective use of it.

Departments

Editors' Corner

2

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar–Apr, July–Aug, and Nov–Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1996 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Bulk rate U.S. postage paid Lancaster, PA, permit 161. Canada Post International Publications Mail Product Sales Agreement No. 290386.

Individual Subscription rates 1 year (9 issues): domestic \$89; Mexico and Canada \$114; Foreign \$129; Institutional/Library rates: domestic \$199, Canada & Mexico \$224; Foreign \$239. To submit articles, please send electronic files on disk to the Editors at 885 Meadowlands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

POSTMASTER: Send domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sigs.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

Externalizing Business-Object Behavior: A Point-and Click Rule Editor

Paul Davidowitz

EAGLE is an integrated set of tools, architectures, processes, patterns, and reusable components that Andersen Consulting brings to its clients to enable the design and development of mission critical business systems. One of the frameworks explored in the area of tailorability was the externalization of business-object behavior, which makes conventional black-box business-object behavior available for tailoring by an end user. This externalized behavior is represented as rule-bases, which are specified in Smalltalk. This paper describes a tool that can be used to create and edit the rules of the rule-bases. The tool guarantees that the source code it produces will always compile and, at run time, will never generate the doesNotUnderstand: message. The tool is a context-sensitive editor that makes use of typing information and handles a wide range of expressions.

The basic technique is to transform a source string via manipulation of its abstract syntax tree. The abstract syntax parse tree, or more accurately ProgramNode tree in Visual Works, is produced as an intermediate step during code

compilation, but is useful in its own right as an intermediate representation between source code and compiled code. The tree representation is convenient for dealing with syntactic issues. This representation can be transformed via decompilation into a string representation, with which most users prefer to interact. One representa-

tion can be transformed into the other as is convenient. Valid syntax and valid message-selectors are achieved by constraining the user to choose from valid manipulations of the ProgramNode tree. Valid message arguments are achieved by having the user choose from manipulations that satisfy type requirements. Here is an example of a rule-base consisting of two rules. The

rule-base derives the attribute isReceiving for a Warehouse business object.

Rule1 premise: [:aWarehouse | aWarehouse isTakingInventory]

Rule1 action: [:aWarehouse | aWarehouse isReceiving: false]

Rule2 premise: [:aWarehouse | aWarehouse isTakingInventory not]

Rule2 action: [:aWarehouse | aWarehouse isReceiving: (aWarehouse isOpen and:[aWarehouse isAvailableStorage])]

As the example shows, a rule contains a single-argument block; the argument being the instance of the business object. The nature of Eagle rules is that they tend to be short and simple. The tool in turn was designed to perform well for shallow-nested blocks and a small number of assignments and temps.

PPD VisualWorks was the development environment; concepts may or may not be applicable to other Smalltalk environments.

The standard input to the tool is a string specifying a single argument block and the class name of the business

*The tool guarantees
that the source code it produces
will always compile and,
at run time, will never generate
the doesNotUnderstand: message.*

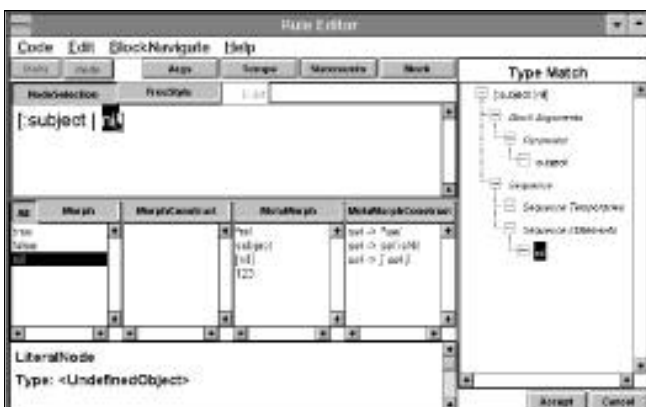


Figure 1. Starts with a generic seed-block of [:subject | nil].

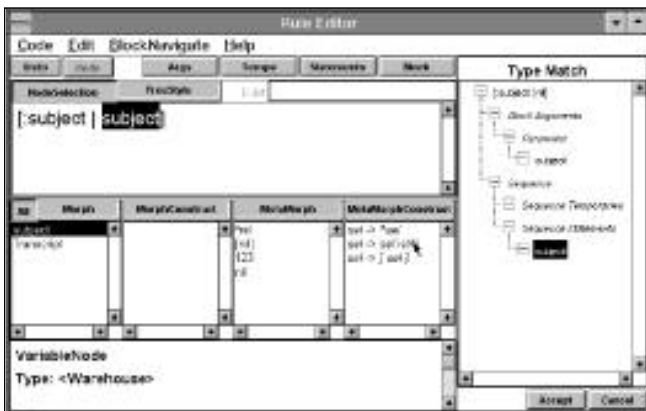


Figure 2. DEMO 2.

object. The output is a similar string. Let's create the action block of Rule2, starting with a generic seed-block of `[:subject | nil]` as shown in Figure 1.

The view in the upper right is the ProgramNode tree. The user has the ability to select a node either by clicking on it in the tree view, or by clicking appropriately on code in the NodeSelection text view (shown). For example, to select a MessageNode in the text view, the user clicks on its selector. We click on `nil`, which is a LiteralNode.

We proceed to replace the `nil` statement with the argument of the block. We have replacement options available from four lists: Morph, MorphConstruct, MetaMorph, and MetaMorphConstruct. These options (collectively referred to as morphs) produce valid replacement nodes for the current-selection. Morph proper and MorphConstruct produce replacement nodes of the *same class* as that of the current selection; MetaMorph and MetaMorphConstruct produce those of a *different* class. The *construct* suffix means that the replacement node, instead of being a fixed prototype, is rather constructed from the current selection. For example, `construct sel -> sel isNil` (sel denoting current selection) means: replace the current selection with a MessageNode of selector `isNil`, and use the current selection as the receiver.

Let's metamorph. We select `subject` in the MetaMorph list. This replaces the LiteralNode with a VariableNode and, we get Figure 2.

The Morph list presents the option of using global Transcript; it is possible to have other globals as well. We select MetaMorphConstruct `sel -> sel isNil`. This replaces the VariableNode with a MessageNode, whose receiver is the VariableNode, and then we get Figure 3.

We have expanded the MorphConstruct list and see choices for different selectors. Each of these choices shows the return type (depicted by the up-arrow), as well as the required types for the arguments (if any). (The vertical bar appearing in a type specification is read as *or*.) We pick `sel -> sel rcvr isReceiving:`. This replaces the current MessageNode with another MessageNode of the same receiver, but different selector, that of `isReceiving:`.

Let's proceed by showing remaining steps with descriptive text, where bold emphasis indicates current selection.

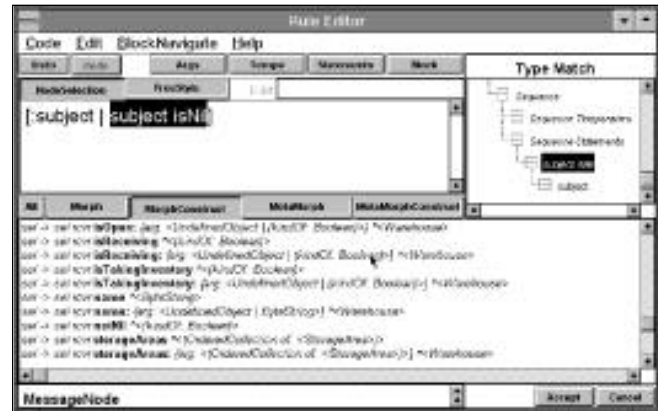


Figure 3. DEMO 3.

- 4) `[:subject | subject isReceiving: nil]`
- 5) `[:subject | subject isReceiving: nil]`
- 6) `[:subject | subject isReceiving: subject]`

At this point, as shown in Figure 4, the label at the top of the tree view has turned red and now reads: Type Mismatch instead of Type Match. The current selection appears (inverted) red in the tree view, and the Accept button at the lower right is disabled. The red color for the node indicates that for that particular node, there is a type mismatch. The type status in the lower left shows that the required-type is `<UndefinedObject | (kindOf: Boolean)>`, but the actual type is `<Warehouse>`. Warehouse is neither an UndefinedObject nor a kindOf: Boolean, so the required type is therefore not satisfied. Unless all nodes have their required type satisfied, the tool will not permit the code to be accepted.

Whereas each allowed manipulation of the ProgramNode tree will result in correct syntax, it will not necessarily satisfy required type for all ProgramNodes. As far as the tool is concerned, an unsatisfied required type is the sole cause for the generation of the `doesNotUnderstand:` message. The tool guarantees prevention of the `doesNotUnderstand:` message, by requiring that all required types are satisfied. The user is alerted to type mismatches, and it is the user's responsibility to satisfy them. Type mismatch usually occurs with message arguments.

We continue to morph, aware that the type mismatch was due to an intermediate morph step.

- 7) `[:subject | subject isReceiving: subject isNil]`
- 8) `[:subject | subject isReceiving: subject isOpen]`
- 9) `[:subject | subject isReceiving: subject isOpen isNil]`
- 10) `[:subject | subject isReceiving: (subject isOpen and: [nil])]`
- 11) `[:subject | subject isReceiving: (subject isOpen and: [nil])]`
- 12) `[:subject | subject isReceiving: (subject isOpen and: [subject])]` (At step 12 there is another type mismatch, again with the argument of `isReceiving:`.)
- 13) `[:subject | subject isReceiving: (subject isOpen and: [subject isNil])]`
- 14) `[:subject | subject isReceiving: (subject isOpen and:`

The wrapper hierarchy (somewhat simplified).

Added state allows a `ProgramNode` to know its parent, as well as keep track of its required and current type. `OrderedCollectionWrapper` wraps collections pointed to by `ProgramNodes`; this includes `SequenceNode` statements and `MessageNode` arguments. *Manipulation* is defined as either wrapper replacement or addition/deletion of an `OrderedCollectionWrapper` child.

Not having the option of modifying the `ProgramNode` classes can be tricky. It was necessary for example, to deep-copy a `ProgramNode`, an ability which it lacks. The technique is to regenerate it by compiling its decompiled string.

THE USER-ROOT

The user-root *parent* hierarchy is designated as follows:

```
nil
  BlockNodeWrapper
    SequenceNodeWrapper
      ParentOfUserRoot
```

The `ParentOfUserRoot` is an `OrderedCollectionWrapper` on sequence statements and is constrained to always have one statement– the user root. Wrapper replacement is forbidden for any wrappers above the user root. Indeed, the user is aware of the user-root tree only. The user-root must have a parent because the user root needs to be replaceable, and this requires a parent wrapper. As a kind of `AbstractStatementNodeWrapper`, the user root has flexibility in being replaceable with wrappers of other classes.

CONSTRUCTS

Constructs are obtained from the soft-typing information of `ProgramNode` instance variables (as shown in Figure 6.). For example, take a `MessageNode`, which is a statement. This node can be replaced with any of the subclasses of `StatementNode` such as `ValueNode`. Since the receiver of a `MessageNode` is itself a `ValueNode`, it follows that it is permissible to replace this `MessageNode` with its receiver, as shown in Figure 8.

Replacing a statement `MessageNode` with its receiver.

Construct Return message receiver (123 isNil → 123).

The following constructs are supported:

- Return message receiver (123 isNil \rightarrow 123)
- Be message receiver (123 \rightarrow 123 isNil)
- Change message selector and arguments only (123 + 456 \rightarrow 123 * nil)
- Change message selector only (123 + 456 \rightarrow 123 * 456)

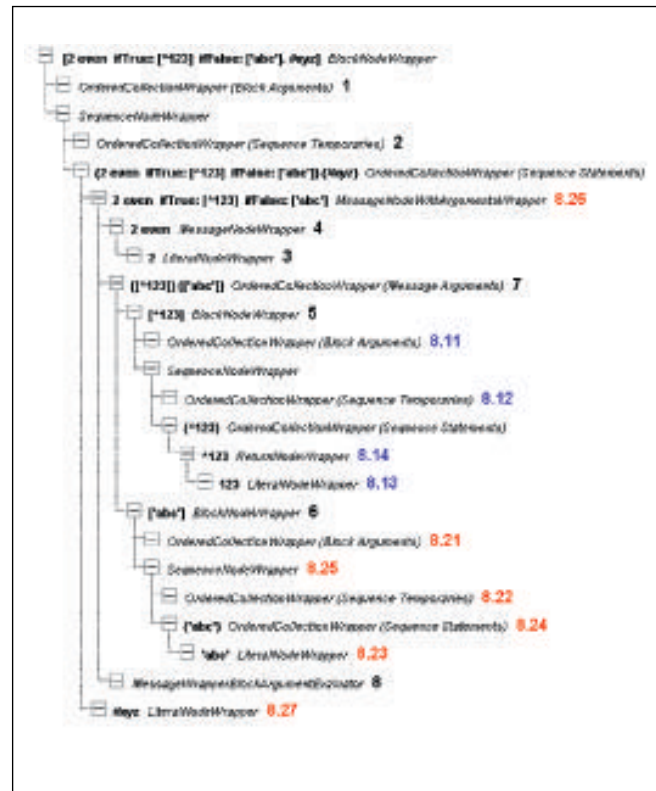


Figure 6. Branch.



Figure 7. Loop.

- Return block's first and only statement ($[123] \rightarrow 123$)
- Enclose statement in block ($123 \rightarrow [123]$)
- Return assignment value ($t1 := 123 \rightarrow 123$)
- Be assignment value ($123 \rightarrow t1 := 123$)



Figure 8. Construction.

- Remove up-arrowT> (^123 → 123)
- Add up-arrow (123 → ^123)

MANIPULATION VALIDATION


Each potential manipulation must be validated to be made available to the user. A manipulation is invalid if it hasn't been implemented for example, like the replacement of a message receiver. A manipulation is invalid if it is syntactically incorrect. For example, the definition of a temporary variable may not be deleted if the variable is currently referenced.

We also check the ramifications for message receivers if the type of a temporary variable were to change. For a message-receiver whose type is determined by a temporary, we ensure that the receiver's required type is satisfied. For example, `t1 isEmpty`, where the receiver's required-type is `<kindOf: Collection>`. If the type of `t1` were to change from `<ByteString>` to `<SmallInteger>`, for example, the check would fail, because `<kindOf: Collection>` would not be satisfied with `<SmallInteger>`. This check is accomplished by simulating the manipulation on a parallel test tree to preview the results.

CORRECT SOURCE POSITION

Wrapper source-position information enables the user to select the wrapper by clicking on it in a text view. A pristine `ProgramNode` tree will have correct source-position information stored in its nodes. Once the tree is manipulated, however, this information is no longer guaranteed to be in synch with the decompiled string. The technique is to create a parallel `ProgramNode` tree after every manipulation, by compiling the decompiled string of the modified wrapper tree. A wrapper seeks its correct source position from its parallel node counterpart.

Instead of directly replacing a node in the tree, why not do so indirectly by replacing the corresponding string component in the overall string? For a `BlockNode`, for example, replace the string `'[nil]'` in the correct spot in the overall string. Then, from this overall string create a fresh wrapper tree that would then have the correct source position. This approach, however, also has a drawback; The state information in the pre-manipulation wrapper tree must always be transferred to the post-manipulation wrapper tree.

In the next article, we plan to conclude by looking at typing, traversal of the wrapper tree, treatment of blocks, and creation of the wrapper-tree. 

Reference

1. Gamma, E. *et al. Design Patterns: ELEMENTS of REUSABLE OBJECT-ORIENTED SOFTWARE*, Addison-Wesley, Reading, MA, 1995.

Paul Davidowitz is a senior developer at Andersen Consulting. He can be reached at paul.davidowitz@ac.com.

Using Events for constraint solving

Annick Fron

Smalltalk has evolved and the good old MVC scheme has now grown up into events. Events are implemented in Visual Smalltalk and Visual Age, but the code will be presented here in the Visual Smalltalk environment.

The MVC implementation in Smalltalk relies on a couple of changed/update messages sent back and forth between dependents.

Figure 1 shows how the dependency mechanism is triggered: Each time object A is modified and calls method "changed", or one of its variants, all of its dependents are informed and execute an "update" method.

A more sophisticated scheme allows an aspect to be passed as an argument to an update, in order to refine the monolithic link between dependent variables. The problem is that the update method has to decode the argument to decide on its behavior. For example,

```
anObject changed: #color
anotherObject >>update: anArgument
  anArgument = #color ifTrue: [...]
  anArgument = #size ifTrue: ...
```

Event programming can be seen as a refinement of the *changed/update* pairs. Namely, instead of maintaining a list of dependents, the system maintains a dictionary with events as entries. This allows for a much more efficient and finely tuned scheme for maintaining dependencies.

Event-based programming has been very successful for building graphical interfaces and for visual programming. In Visual Age and Visual Smalltalk, events are the backbone of links between objects in order to build a visual application.

Events are also popular at the operating and windowing system levels, but this is not the topic here.

CONSTRAINT-SOLVING TECHNIQUES

Constraint-solving techniques allow the user to tackle such combinatorial problems as scheduling a meeting or dispatching resources according to some criteria.

One popular technique, called "Finite domain tech-

niques," is used when variables can only take a finite number of values. For instance, scheduling a project when the time unit is a day can be modeled using this technique.

The idea is to represent the variable not as bounded to a value, but with a range of potential values called its domain. The solver will then ensure consistency between these domains through the constraints.

For instance, if two variables are constrained to be equal, their domains should be the same. Hence, each time one domain is modified the other one should be as well.

The easiest way to represent domains is through intervals. One easy way to detect interval modification is through its bounds. On intervals, it is thus possible to define two modification events, on each bound. We will call them: #min and #max.

In Figure 2, two constrained variables and one constraint are shown. When the domain of x is modified by some other constraints or external event, it informs all connected constraints, in this case here the equality constraint. This constraint has only one connected variable, y. Because it is equal, it must tell y to modify its lower bound as well, thus triggering a #min event for y. y in turn will inform its own constraints.

Consistency on finite domain has been proven to be efficient for solving constraints on integers, which is usually exponentially complex. The only restriction to get a fixed-point solution is to always shrink the domain; never

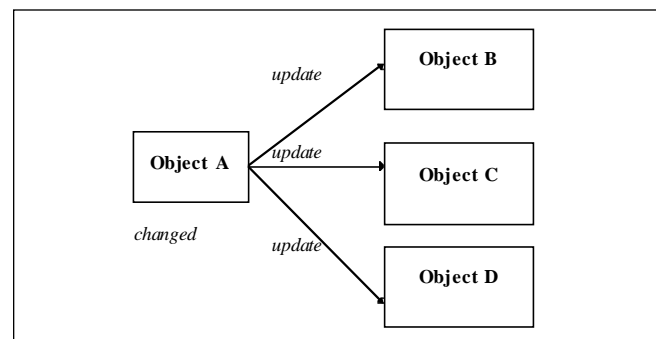


Figure 1. A changed message in object A triggers update messages on all its dependents.

increase it. The triggering order of events on domains has been proven not to affect the result. Yet, this algorithm is not complete and needs an enumeration phase in order to find all of the solutions. This will be omitted here.

A SMALLTALK IMPLEMENTATION OF CONSTRAINTS

The Smalltalk implementation of a constrained variable is very simple, and events will help a lot in the job. The first question to ask is: "Do the constraints represent objects, or are they included into other objects?" Having constraints as true objects is very helpful. Because it allows the implementation of a constraint hierarchy to refine new constraints, it provides a handle to dynamically inhibit or activate a constraint.

Here, we take a simple assumption that constrained variables need not be any Smalltalk object; they can derive from a specific root under object. Therefore, we get two object hierarchy roots, one for constraints, one for variables.

THE VARIABLES

Variables are defined by their domain and by the events they are able to respond to. Here we get only finite domain variables (the model can be extended to other kinds of variables, but for reasons of simplicity, it will be omitted here).

In Visual Smalltalk, it is possible to define any semantic event at the class level, through the method `constructEventsTriggered`. The second step is to couple every domain modification (instance variable accessor) with an event using the `triggerEvent: message`. This should be compared to adding changed to instance variable accessors displayed on a view.

The operator `*` and `=@` are only syntactic sugar for the final example. `=@` means equals in the constraint sense, which should not be confused with variable assignation (variables that have no value yet).

For esthetic reasons, variables get a name to be printed.

Object

```
subclass: #ConstrainedVariable
instanceVariableNames: 'name domain'
classVariableNames: ''
poolDictionaries: ''
```

!ConstrainedVariable class methods!

constructEventsTriggered

```
"Private - answer the set of events that instances of the
receiver can trigger."
^#(#min #max) asSet
```

from: aMin to: aMax

```
^super new from: aMin to: aMax
```

!ConstrainedVariable methods!

```
* y
```

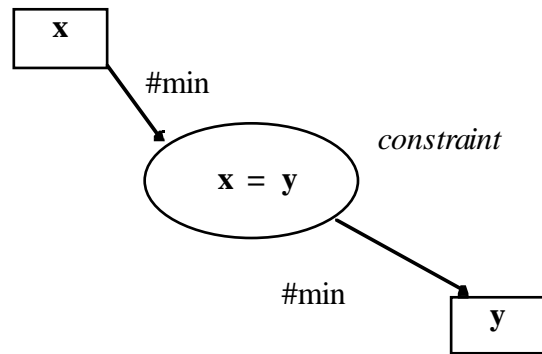


Figure 2. An event on a variable will propagate to all related constraints. The constraints in turn will trigger new events on the connected variables.

```
| c |
c := ActTimes x: self y: y.
^c result "a constrained variable"
```

```
+ y
| c |
c := ActAdd x: self y: y.
^c result "a constrained variable"
```

```
@= y
```

```
| c |
c := ActEquals x: self y: y.
^c result "a constrained variable"
```

domain

```
^domain!
```

domain: anObject

```
domain := anObject!
```

max

```
^domain last!
```

max: aValue

```
domain newMax: aValue.
self triggerEvent: #max!
```

min

```
^domain first!
```

min: aValue

```
domain newMin: aValue.
self triggerEvent: #min!
```

printOn: aStream

```
aStream nextPutAll: name, ' ', domain printString! 
```

from: aMin to: aMax

```
domain := Interval from: aMin to: aMax.
```

THE CONSTRAINTS

Here we need two types of constraints: constraints on comparators, which will have two arguments, `x` and `y`, as pointers to the variables they involve; and constraints on operators when we need to introduce a new constrained

variable, the result, which is also stored as an instance variable and called *r*.

When a constraint is created in the program, the method *post* will define the event dispatch mechanism according to the constraint semantic. It uses the *when:send:to:* message that links events to actions on several objects.

The following table shows the links between events and domain updating for the equality constraint *x @= y*.

Description	Event	Update
x minimum increased	#min on x	y domain takes min(x) as new lower bound
x maximum decreased	#max on x	y domain takes max(x) as new upper bound

Symmetrically, we can define the same for *y*. Methods have to be defined in the constraint class to compute the updates, such as *xmin* or *xmax*. The constraint is called *ActEquals*. *ActEquals* also needs an *init* method in order to ensure domain consistency prior to any computation.

In the example, the two operators introduced are *ActAdd* (addition) and *ActTimes* (multiplication by a constant factor). These classes inherit from *ActConstraint*, an abstract class which does not have any behavior here.

```
ActConstraint subclass: #ActEquals
  instanceVariableNames: 'x y'
  classVariableNames: ''
  poolDictionaries: ''!
```

!ActEquals class methods !

```
x: var1 y: var2
  ^super new x: var1 y: var2.
```

!ActEquals methods !

```
x: var1 y: var2
  x := var1 . y := var2.
  self post; init.
```

```
init
  | m s |
  m := x min max: y min.
  s := x max min: y max.
  x min: m; max: s.
  y min: m; max: s
```

```
post
  x when: #min send: #xmin to: self.
  x when: #max send: #xmax to: self.
  y when: #min send: #ymin to: self.
  y when: #max send: #ymax to: self
```

```
xmax
  y max: x max
xmin
```

```
y min: x min
```

```
ymax
  x max: y max
```

```
ymin
  x min: y min
```

=====

```
ActConstraint subclass: #ActAdd
  instanceVariableNames: 'x y r'
  classVariableNames: ''
  poolDictionaries: '' !
```

!ActAdd class methods !

```
x: var1 y: var2
  ^super new x: var1 y: var2
```

!ActAdd methods !

```
x: aVar1 y: aVar2
  x := aVar1
  y := aVar2.
  r := ConstrainedVariable from: (x min + y min) to: (x max + y max).
  self post
```

```
post
  x when: #min send: #xmin to: self.
  x when: #max send: #xmax to: self.
  y when: #min send: #ymin to: self.
  y when: #max send: #ymax to: self.
  r when: #min send: #rmin to: self.
  r when: #max send: #rmax to: self.
```

```
rmax
  x max: (r max - y min).
  y max: (r max - x min).!
```

```
rmin
  x min: r min - y max.
  y min: r min - x max!
```

```
xmax
  r max: x max + y max.
  y min: r min - x max
```

```
xmin
  y max: r max - x min.
  r min: y min + x min
```

```
ymax
  r max: x max + y max.
  x min: r min - y max
```

```
ymin
  x max: r max - y min.
```

CONSTRAINT SOLVING

r min: x min + y min

ifTrue: [r max: y * x max]

ifFalse: [r min: y * x min]

=====

ActConstraint subclass: **#ActTimes**

instanceVariableNames: 'x y r'

classVariableNames: "

poolDictionaries: " !

!ActTimes class methods ! !

x: var1 y: anInteger

^super new x: var1 y: anInteger

!ActTimes methods !

x: aVar1 y: anInteger

x := var1. y := anInteger.

anInteger >= 0

ifTrue: [r := ConstrainedVariable from: anInteger * x
min to: anInteger * x max]

ifFalse: [r := ConstrainedVariable from: anInteger *
x max to: anInteger * x min].

self post.

post

x when: #min send: #xmin to: self.

x when: #max send: #xmax to: self.

r when: #min send: #rmin to: self.

r when: #max send: #rmax to: self.

rmax

y >= 0

ifTrue: [x max: (r max / y) floor]ifFalse

: [x min: (r max / y) ceiling]

rmin

y >= 0

ifTrue: [x min: (r min / y) ceiling]

ifFalse: [x max: (r min / y) floor]

xmax

y >= 0

xmin

y >= 0

ifTrue: [r min: y * x min]

ifFalse: [r max: y * x min]

=====

EXAMPLE

The simple example is used to test the code and show how some partial solving can be achieved. It defines the domains of the variables, and sets the only constraint: $x + 3y + 4z = 2t + c$.

ConstrainedVariable class>>example

"ConstrainedVariable example"

| x y z t c |

x := ConstrainedVariable from: 0 to: 3. x name: 'x'.

y := ConstrainedVariable from: 0 to: 1. y name: 'y'.

z := ConstrainedVariable from: 2 to: 5. z name: 'z'.

t := ConstrainedVariable from: 0 to: 3. t name: 't'.

c := ConstrainedVariable on: #(5). c name: 'c'.

((x + (y * 3)) + (z * 4))@=((t * 2) + c).

x printOn: Transcript.


y printOn: Transcript.

z printOn: Transcript.

t printOn: Transcript.

Transcript cr.

COMMERCIAL IMPLEMENTATIONS

This article has given a brief insight into constraint solving techniques. These techniques have been commercially implemented in C++, and used on such industrial applications as train and plane scheduling. Smalltalk events allow a very elegant presentation of the consistency scheme. 

Annick Fron can be reached at 100342.3301@compuserve.com.

How to display an object as a string:

TypeConverter and PrintConverter

Bobby Woolf

IN PART 1 of this article, I described the need for each object to generate a short string that identifies itself. VisualWorks provides two messages to do this: `printString` and `displayString`. `printString` displays the object to a developer, so it specifies the object's class. `displayString` displays the object to a user, so it should not specify the object's class. In Part 2, I'll talk about two classes for converting an object to a String: `TypeConverter` and `PrintConverter`.

TYPECONVERTER

`TypeConverter` is a class that was introduced in VisualWorks 1.0. It is a kind of `ValueModel`, a model that contains a single aspect called "value." The converter's subject is itself a `ValueModel`, so the converter enhances its subject by adding conversion behavior while preserving its value-model behavior.¹ This is an example of the Decorator pattern.²

The conversion behavior that a `TypeConverter` adds is the ability to convert its subject's value from one type to another, and back again. This assumes that the two types are convertible and the conversion is bi-directional.

The primary type that `TypeConverters` convert to is `Text`, which is just a fancy string. `ParcPlace` has already implemented the algorithms `numberToText`, `dateToText`, `objectToText`, and so on.

`TypeConverter` has two advantages over `printString` and `displayString`:

- A `TypeConverter` does not just display the object as a string; it converts the object to its String (actually `Text`) equivalent.
- A `TypeConverter` can convert the String equivalent back into the original type.

`TypeConverter` was introduced to support `InputFieldView`. An input field's value might be any type of object. The field needs to display the object as a string, a task that `displayString` can do. However, if the user types in a new

string, the field needs to convert that string back into its value's type. The standard message for converting a string into an object is `readFromString`.

A `TypeConverter` encapsulates `displayString` and `readFromString`: together so that a source type can be converted into a String and back again. In the process, it remembers the target's type—whether the new object should be a Number, a Date, etc. It also checks for exceptional conditions, such as the original object being nil or times, when empty-string from the user should be converted into nil. Since `TypeConverter` encapsulates all of this behavior into a single object, it can easily be reused any time this conversion is needed.

AN EXAMPLE

Here's an example of how a `TypeConverter` can be used. Let's say you've stored somebody's age, and that it's accessible through a `ValueModel` called `ageHolder`. Age is a Number, but you need to display it in a field. If the user types in a new age, you need to convert it back into a Number. This code shows the two value-models you'll need:

```
| ageHolder ageAsStringHolder |
ageHolder:=10as Value.
ageAsStringHolder :=
    TypeConverter onNumberValue: ageHolder.
Transcript cr; show: 'age's type is ',
    ageHolder value class displayString.
Transcript cr; show: 'ageAsString's type is ',
    ageAsStringHolder value class displayString.
```

The transcript shows that `ageHolder` contains a `SmallInteger`, and `ageAsStringHolder` contains a `Text`. The input field's model would be `ageAsStringHolder`. Any code needing to access the age in its unconverted form would go through `ageHolder`.

PRINTCONVERTER

The problem with `TypeConverter`, `displayString`, and `printString` is that they all assume that there's only one way to show a particular object as a string. This "one-size-fits-all" approach is often insufficient. A Date can be printed

many different ways: December 25, 1990; 25-DEC-90; Christmas Day. A Time has several choices: 4:00 P.M.; 16:00:00; etc. A Number can be printed with leading and/or trailing zeros: 1; 1.00; and so on. TypeConverter can't handle these formatting choices. As long as it converts to the right type, it's done. Subtleties about what exactly the resulting type should look like have to be handled somewhere else.

PrintConverter is essentially a TypeConverter that has been optimized to display objects to the user as strings. Whereas, a TypeConverter can convert from any type to any other type and back again, and PrintConverter only converts to strings. Like `displayString`, PrintConverter doesn't even convert the object to its string equivalent; it just displays the object as a string. Like a TypeConverter, a PrintConverter can convert an input string back into the original object's type.

The major advantage PrintConverter has over both TypeConverter and `displayString` is that it can format the string it displays. You do this by specifying the type of source object to be converted, but also by specifying the format of the resulting string. For example,

```
PrintConverter for: #date
```

will create a PrintConverter that will display a Date using the default format. On the other hand,

```
PrintConverter  
for: #date  
withFormatString: 'd-mmm-yy'
```

will create a PrintConverter that will display a Date using the format specified.

The other advantage of the way PrintConverter works is that a single converter can be used to display a number of objects of the same type with the same format. To display a list of twenty Dates with the format 'd-mmm-yy,' you only need one PrintConverter for the whole list. As the list prints each Date, it runs the Date through the PrintConverter, which returns the formatted string for that Date. To perform a similar conversion using TypeConverters, you would need twenty TypeConverters, one for each Date.

Ironically, the only widgets that use PrintConverters are input fields, combo boxes (which, of course, contain input fields), and those data sets that contain input fields and/or combo boxes. `SequenceView` (the List widget) doesn't use PrintConverter. So if you develop a PrintConverter that formats Dates in a special way that you like, you can use that format to display a list of Dates, in a `DataSetView` but not a `SequenceView`. To use that format in a List widget, you have to implement a method, such as `Date>>displayStringSpecialWay`, and set the `SequenceView`'s `displayStringSelector` to `displayStringSpecialWay`.³ So now you have the same format implemented in a special instance of PrintConverter for Dates and a special method in Date. I would prefer to only implement this code in one place, not two.

WHERE PRINTCONVERTER IS USED

You're already using PrintConverters, even if you don't realize it. In the Painter, when you specify the properties for an Input Field, two of the properties on the Basics page are Type and Format. What you're specifying is the source object's type (String, Symbol, Text, Number, etc.) and its format ((@@@) @@@-@@@@, 0.00, etc.). This is all of the information needed to set up a PrintConverter. When you open the window, as the Builder creates the Input Field, it also creates a PrintConverter with the properties you have specified. Combo Box and Data Set have similar properties that specify the PrintConverter to use.

As you create your own objects that need to be displayed as strings, I suggest you create new PrintConverters to display them. Let's say you have a Money class.

You want to be able to display a

Money object in an Input Field and get a new one from the user by having him type it within the field.

1. You would need to implement `PrintConverter>>initForMoney`. Use the corresponding methods for Date, Number, and String as examples of how to implement your methods.
2. Modify `PrintConverter class>>for:` to add `#money` onto that big, long case statement.

To make your new Money PrintConverter accessible from the Properties Tool:

3. Modify `InputFieldSpec class>>typeMenu` to add 'Money' -> `#money`.

Now the Properties Tool will allow you to specify the type of a value for an Input Field as Money. This will also be available for Combo Box and Data Set widgets.

Modifying the list of formats for one of `ParcPlace`'s types is also simple. See the methods in `InputFieldSpec class>formats`. For example, to add another string format, modify the method `InputFieldSpec class>>defaultStringFormats`.

Unfortunately, specifying special formats for your new Money PrintConverter is not very easy. You would need to implement `MoneyPrintPolicy` as a subclass of `PrintPolicy`. That ultimately involves implementing `MoneyPrintPolicy class>>nextTokenOn:` and `MoneyPrintPolicy>>print:on:policy:`, a task which is not for the faint of heart.

READFROMSTRING:

Earlier I mentioned that TypeConverter (and PrintConverter) use `readFromString:`. If you don't know what this method does, you'll need to learn so you can implement your own converters.

`Object class>>readFromString:` is sort of the opposite of `Object>>printString`. The implementor in `Object` really only works if the string contains a literal or a store-string (see `Object>>storeString`).⁴ However, implementors in more

specialized classes work well because they can assume that the string represents an instance of that class. For example, Object doesn't know what to make of "April 5, 1982," nor do most classes, but Date is able to make it into a Date.

Just like printString uses printOn: to do most of the work, readFromString: lets readFrom: do everything. Thus you'll never subimplement readFromString:, but you should implement readFrom: in your own classes. Ideally, readFromString: should reverse the process of printString and displayString and should also recognize any formatting that a PrintConverter might throw in. For example, if the printString for a particular Person is "John Smith," you will need to implement Person class>>readFrom: to interpret that string as a Person with that name.

GLOBALIZATION

Globalization (internationalization), a feature added in VisualWorks 2.5, adds a whole new twist to displaying an object as a string. In order to know what string to display and how to interpret a new string, you have to know what language the user speaks and what formatting conventions he uses. This has not been an issue prior to this upgrade, because we always assumed (often inaccurately) that the user speaks American English.

First, ignore globalization when implementing printString (printOn:), storeString and readFromString:. These methods are for developers. Since Smalltalk is written in American English, the methods can assume that they should use that language. In addition I think these methods need to be simple and highly reliable; globalization is an unnecessary complication.

Second, ignore globalization when implementing displayString as well. Just as printString should be simple, so should displayString. Use displayString as a quick-and-simple way to display an object to the user. When this becomes too complicated, such as when globalization is necessary, use a PrintConverter instead.

Third, TypeConverters don't need globalization either. A TypeConverter is only responsible for converting an object's type to or from a string. As long as the object is a String, any effort to format it or translate it into another language is unnecessary. These are responsibilities better fulfilled by a PrintConverter.

Finally, a PrintConverter should use globalization when performing its conversion. Look at the ones that ParcPlace has already implemented for Date, Time, and Timestamp. Their toPrint and toFormat blocks use "Locale current ..." and TimestampPrintPolicy to display a string appropriate for the current location. Similarly, the toRead blocks use "Local, current ..." and TimestampPrintPolicy to read a string from the user. As I mentioned earlier, implementing your own subclass of PrintPolicy is difficult.

A PrintConverter that uses globalization does not use

readFromString: to convert a string back into an object. readFromString: does not use globalization, so it is not appropriate for this purpose. Instead, a globalized PrinterConverter uses a LocaleSensitiveDataReader to convert a location-specific string back into an object. Just as PrintPolicy displays an object in a location-specific way, LocaleSensitiveDataReader does the opposite. If your own subclass of PrintPolicy displays an object in a location-specific way, you'll need to implement a subclass of LocaleSensitiveDataReader to read it back.

CONCLUSIONS

Here are the main points in this article:

- Unlike printString and displayString, a TypeConverter can convert a string back into an object again.
- A TypeConverter is useful for converting an object to a String, but doesn't display it as a string very well.
- PrintConverter is designed to display an object to the user as a string. It makes displayString into a first class object.
- PrintConverter is able to format the display string the way the user prefers.
- A PrintConverter uses readFromString: to convert a string back into an object.
- All classes should implement readFromString: (via readFrom:) to convert their print-strings and display-strings back into an instance.
- A PrintConverter should use the globalization framework to format the string for the current location.

I hope you now see that displaying an object as a string is often not a trivial matter. VisualWorks provides several protocols and frameworks to help you. If you learn how to use them well, I think you'll find your system a lot easier to use. ■

References

1. Coplien, James O. and Schmidt, Douglas C., Editors. PATTERN LANGUAGES of PROGRAM DESIGN. Addison-Wesley, 1995. "Understanding and Using the ValueModel Framework in VisualWorks Smalltalk" by Bobby Woolf.
2. Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. DESIGN PATTERNS: ELEMENTS of REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley 1995.
3. Kohl, William and Howard, Tim "VisualWorks List Components" *The Smalltalk Report*, June 1994.
4. LaLonde, Wilf R. and Pugh, John R. INSIDE SMALLTALK, Vol. 1., Prentice-Hall, 1990, Section 6.2.9, "Read/Write Operations: PrintStrings and StoreStrings."

Bobby Woolf is a senior member of technical staff at Knowledge Systems Corp. in Cary, North Carolina. He mentors Smalltalk developers in the use of VisualWorks, ENVY, and Design Patterns. Comments are welcome at woolf@acm.org. or at <http://www.ksccary.com>.



Dwight Deugo



Wayne Beaton

Managing Connection Complexity

You're given your first visual programming assignment. You spend the day working on it and are proud that you completed it without writing one line of Smalltalk. A week later, after being asked to add a new feature, you return to your picture (the visual equivalent of code) and realize that you can't remember the semantics of all of those connections. It takes the rest of the day to understand what you developed last week and add the new connections to support the additional feature. Does this sound familiar?

Often we are asked "Even though visual programming enables developers to create windows quickly, what approach should one use in order to minimize the connection complexity of visual parts?" The approach should enable developers to quickly and easily understand what they're building now and what they've built in the past.

One source of the problem developers are experiencing is a result of working in a new paradigm (the construction from parts paradigm) with little or no training. When the jump was made from assembler to structured programming, many developers wrote spaghetti code until they were educated in structured programming techniques. Now, for the same reason, many developers are painting spaghetti visual parts. The one difference today is that you can see the mess you've created for yourself. However, with a little care, this need not be so. Visual programming environments, such as IBM's VisualAge, provide an assortment of parts and facilities for decreasing the complexity of visually programmed systems.

In this column, we examine one technique for managing the connection complexity of visual parts. This technique is called factoring. Too often we see a window, a Client Profile Editor for example, containing every connection to support editing of a client's name, address, phone numbers, credit history, and more, only to see other windows provide the same support for viewing or editing the identical information. One problem is that the Client Profile Editor has too many connections to be

understandable, especially when you add those to support menu interaction. Another problem is that the same connections are used in every window that supports the editing of the client's address.

The simple and easy solution to these problems is to encapsulate the connections and parts that support the editing and viewing of a business object into one reusable part. The composite part can be placed in any visual part, and connect to its required business and supporting objects. One can significantly decrease the number connections in any visual part that uses reusable parts, because the reusable part manages its own connections, and they are not visible to surrounding visual parts. Also, changes to the reusable part's behavior or visual appearance are immediately reflected in any part that uses it, avoiding any potential maintenance problems that might occur when having the same functionality implemented in two or more places.

Here, we make use of two reusable forms to minimize the connection complexity of a client profile editor. The window supports editing of a client profile, including a client's name, age, and address. The reusable forms are views of two business objects: a ClientProfile and a CanadianAddress. Like the editor, they support the cancellation of edit changes.

THE CLIENT PROFILE EDITOR APPLICATION

The Client Profile Editor lets one edit and, if desired, cancel any changes to an existing ClientProfile. For this example, we do not show how the ClientProfile is loaded or saved. Our goals are to minimize the number of visual connections needed to meet the editor's requirements, and to permit Smalltalk code only when the operation can't be done visually. The rule-of-thumb, "less is better," is true when it comes to visual connections. Fewer connections make a window's implementation easier to understand, and easier to maintain.

A ClientProfile contains a CanadianAddress, and it is from this object that we begin our exercise. Although our requirements have the address displayed from only a Client Profile Editor, we decided to create a reusable CanadianAddressForm to display it. We realize that addresses are often modified, or simply displayed in many different windows, and we want all future windows to display them in the same format. Encapsulating the logic for viewing

Dwight Deugo and Wayne Beaton are senior members of the development and educational staff at The Object People, in Ottawa, Ontario, Canada. Dwight (dwight@objectpeople.on.ca) has immersed himself in objects for more than 10 years and has helped clients with their object immersions as a project mentor and as a course instructor. Wayne (wayne@objectpeople.on.ca) is the coordinator of course construction and a software developer.

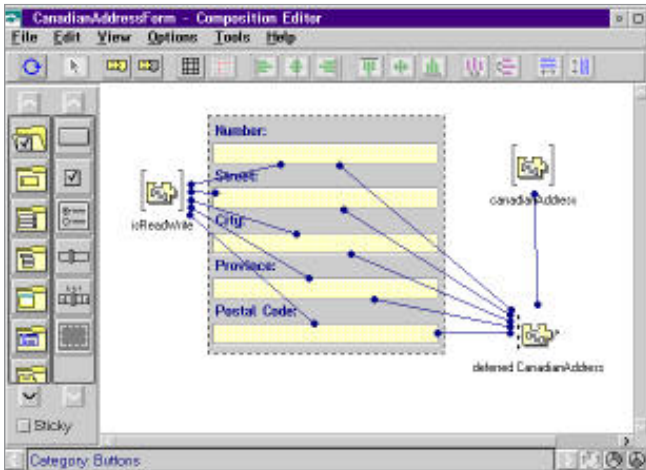


Figure 1. Canadian Address Form.

and modifying an address in the CanadianAddressForm, where it belongs, will also help later to decrease the connections in the Client Profile Editor.

A CanadianAddress is a business object with the five parts: a street number, a street name, a city name, a province name, and a postal code. A CanadianAddressForm, shown in Figure 1, is an editable view for the address. Since a CanadianAddressForm requires an address to edit, one variable part, called 'canadianAddress', is required to reference the address. Also, since the form can be used for both viewing and editing, another variable part, called isReadWrite, is required to store a Boolean, which indicates whether the form is for editing or strictly for viewing. To enable or disable editing in the form's Text parts, the 'self' attribute of the isReadWrite variable part is connected to the 'enabled' attribute of every Text part. To support the undoing of address changes, a Deferred Update part—a VisualAge supplied part—is created from the canadianAddress variable part. A connection between its 'target' attribute and the valueHolder attribute of the variable links the two.

One can view a Deferred Update part as a copy of an original that maintains a stack of changes, which can be applied backwards and forwards to the original part. The Deferred part's interface is similar to the original, having the same attributes, and connects to other parts in an identical manner. In our CanadianAddressForm, we connected the CanadianAddress's attributes to the 'object' attribute on the corresponding Text parts. To permit parts using the CanadianAddressForm to role back or apply the changes from the Deferred Update part to the canadianAddress, its apply and cancel actions are promoted as applyChanges and cancelChanges actions of the CanadianAddressForm.

The isReadWrite and canadianAddress variables are assigned objects by other parts using the CanadianAddressForm. However, to those using the CanadianAddressForm, the isReadWrite and canadianAddress variables appear only as attributes of the form and not as variables, and the connections between these variables and other parts in the form are invisible. To achieve this effect in VisualAge, the vari-

ables' self attributes are promoted as isReadWrite and canadianAddress, respectively.

The part is now complete. It has two public attributes: isReadWrite and canadianAddress, and supports two actions: applyChanges and cancelChanges. Anyone using the part must provide a Boolean value for the isReadWrite attribute and a CanadianAddress object for the canadianAddress attribute. To commit the edit changes to the CanadianAddress object one can invoke the applyChanges actions, and to undo any edit changes one can invoke the cancelChanges action.

The exercise is repeated again, but this time for a ClientProfileForm, and for the same reasons: to provide a single format for the display of a ClientProfile, to encapsulate the logic for viewing and modifying it, and to decrease the number of connections in the Client Profile Editor.

A ClientProfile is a business object with three parts: name, age, and address. A ClientProfileForm, shown in Figure 2, is an editable view of the profile. Like the CanadianAddressForm, the ClientProfileForm requires two variable parts: isReadWrite and clientProfile. The purpose of the isReadWrite variable is identical to the one in the CanadianAddressForm, and it has similar connections. The purpose of the clientProfile variable is to provide a reference to the form's business object. To support the undoing of client profile changes, a Deferred Update part is again used, this time created from the clientProfile variable part.

The Client profile form includes a CanadianAddressForm. To use the CanadianAddressForm, the isReadWrite variable part's self attribute is connected to the CanadianAddressForm's isReadWrite attribute, and the clientProfile part's address attribute is connected to its canadianAddress attribute. These connections provide the CanadianAddressForm with the objects it requires to function—in only two connections!

Finally, we want the ClientProfileForm to support the acceptance or cancellation of edit changes. Unlike the CanadianAddressForm, the files have not been touched at all. Where we promoted its deferred part's corresponding actions, this time we have to write two methods: applyChanges and cancelChanges, and add them to the form's public interface. This means that to accept or cancel the changes on this form is to have both its deferred part and the CanadianAddressForm accept or cancel the changes, which can't be done visually. The code for the scripts are as follows:

```
applyChanges
(self subpartNamed: 'deferred ClientProfile') apply.
(self subpartNamed: 'Canadian Address Form')
performActionNamed: #applyChanges.
```

```
cancelChanges
(self subpartNamed: 'deferred ClientProfile') cancel.
(self subpartNamed: 'Canadian Address Form')
performActionNamed: #cancelChanges
```

The part is now complete. It has two public attributes:

isReadWrite and clientProfile, and supports two actions: applyChanges and cancelChanges. Anyone using the part must provide a Boolean value for the isReadWrite attribute and a ClientProfile object for the clientProfile attribute. To commit the edit changes to the ClientProfile object, one can invoke the applyChanges actions. To undo any edit changes one can invoke the cancelChanges action.

We now have the parts required to build a Client Profile Editor, shown in Figure 3: a ClientProfile, a ClientProfileForm, and two buttons to invoke the ClientProfileForm's apply and cancel changes actions. In a finished application, these buttons would be replaced with menu items. However, in this article, we wanted to keep it simple and did not get into a discussion on menus. The Editor's connections are as follows:

```
ClientProfileForm.clientProfile → clientProfile.self
Read/WriteToggleButton.selection →
ClientProfileForm.isReadWrite
AcceptButton.clicked → ClientProfileForm.acceptChanges
CancelButton.clicked → ClientProfileForm.cancelChanges
```

How many connections are required to edit a ClientProfile? There are four: one to instruct the ClientProfileForm which ClientProfile to work with; one to identify whether one is viewing or editing the profile; and two to apply or cancel the end-user's changes to the existing ClientProfile. One could argue that we would have the same number of connections if we implemented all viewing and modification operations in the ClientProfileEditor itself. We would accept that argument.

However, a more important question to ask here is: "Have we gained anything by layering those connections in our ClientProfileForm and a CanadianAddressForm?" The answer is an overwhelming YES! Our two forms and one editor are easy to understand and maintain. We can reuse our forms in any other windows that need to display or modify CanadianAddresses or ClientProfiles. We have a framework for canceling user changes to business objects.

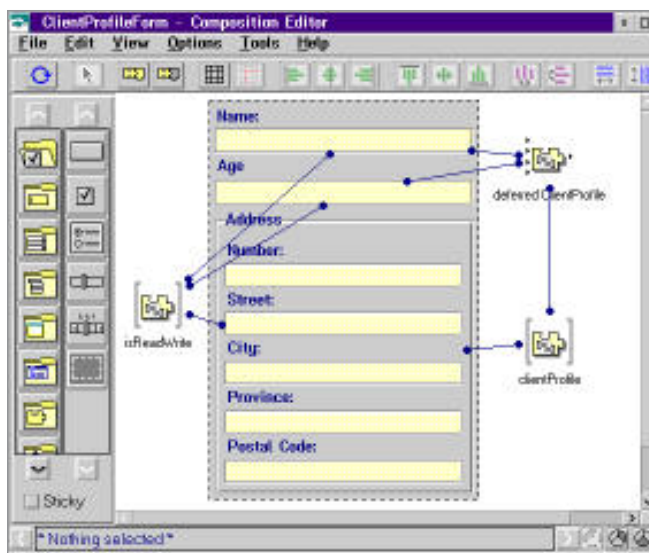


Figure 2. Client Profile Form.

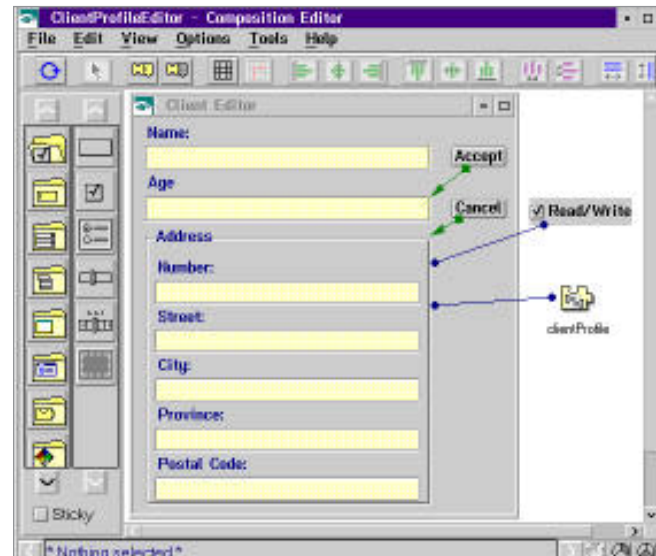


Figure 3. Client Profile Editor.

Finally, we have achieved our original goal of minimizing the connection complexity of our windows.


DO THE RIGHT THING

Minimizing connections and planning for reuse takes some thought. Often, one is required to build a window that manipulates a number of business objects. The temptation is to have the display and modification logic in one window, rather than factor the window into a number of reusable components and use them to construct the window. As seen, factoring your windows into a number of components decreases the number of connections and the complexity of each component, as well as the final window. This makes your components easier to understand and maintain. Even though you may not need the components for any other window yet, you or someone else will! So why not do it the right way to begin with?

Remember, many objects that do little, is better than few objects that do too much. Therefore, a window that is composed of many simple, reusable components, is better than a window that does everything itself. We strongly suggest that every business object have a form built for its display and editing.

Factoring is of course not a new idea. Good GUI developers have been doing it for years with tools that all GUI builders provide. For example, in Visual Works reusable forms are called "subcanvases." In ObjectShare's WindowBuilder they are called "composite panes," and in Digitalk's Parts they are called "nested parts." Whether visually programming or using one of the layout-type GUI builders, building and using reusable forms is not just a good idea, it's great object-oriented programming.

THE CODE

The code presented in this column and in future columns is available on the World Wide Web. Our URL is <http://www.objectpeople.on.ca>. 



Jan Steinman



Barbara Yates

SmallDoc Web Serving

In our June and September 1995 columns, we introduced a hyper-literate programming system we call "SmallDoc." In the last two issues, we sketched out how to turn SmallDoc into HTML, and how to build a generic TCP/IP server framework. This issue ties it all together so you can begin serving your Smalltalk project documentation to anyone with a Web browser.

The generic TCP/IP server described earlier needs only one or two blocks of Smalltalk code in order to implement a complete server. A message is sent to the class that associates a "service" block with a port number, and a second, optional (but strongly encouraged!) message is sent to the class to associate an "exception" block with the same port number.

For example, a simple hypertext service can be implemented by adding the following method to the *TcpServer* class that we presented last month:

TcpServer class:

initializeForHttpd

"Set up a service and exception handler suitable for servicing World Wide Web requests."

```
selfdefaultHandlerFor: 80 is: [:exception :stream |
    stream httpChattyHandle: exception];
defaultServiceFor: 80 is: [:stream |
    stream htmlForSmallDocRequest]
```

httpd

"Answer the default hypertext transport protocol server."

^self onPort: 80

Now, to start up a Hypertext Transfer Protocol server all you need to do is evaluate "TcpServer httpd." Of course, if you do that right now it will crash, because we haven't really written the handler or service blocks yet.

If you are on a UNIX machine, remember that port 80 is privileged — you will have to run Smalltalk as root to

run this service. If that is not possible, choose some other port above 1024, such as 8080.

When an error is encountered in the server, it should alert the client so that things can be fixed. Our `httpChattyHandle:for:` method assumes the person using the Web browser might know something about Smalltalk, and so it sends contextual information back to the client. For non-developers, you might want to simplify this by reporting only that an error occurred.

PositionableStream

httpChattyHandle: exception

"Upon trouble with the request, attempt to send back a contextual information from <exception> in HTML format on <stream>."

```
| ctx |
self
```

```
cr; nextPutAll: 'HTTP/1.0 500'; cr;
nextPutAll: 'Server: '; nextPutAll: (TcpServer
signatureIn: TcpServer controller); cr; cr;
nextPutAll: '<HTML><HEAD><TITLE>Unhandled
exception!</TITLE></HEAD><BODY><H1>';
nextPutAll: exception errorString;
nextPutAll: '</H1><P>Your request had a
problem. Please copy the following stack and mail it to
the<A HREF="mailto:'
nextPutAll: (EmUser called:'Supervisor')
networkName;
nextPutAll: "'>ENVY Library
Supervisor</A>.</P>'.
ctx := exception thisContext.
5 timesRepeat:
    [ctx == nil ifFalse:
    [self print: ctx;cr]].
self nextPutAll: '</BODY></HTML>'; cr; flush
```

Also, this example relies on ENVY repository information to report the server version information, and to obtain the email address of the repository supervisor. You should use suitable substitutes if you use a different code management system.

SERVICING REQUESTS

Now that we can handle failed requests, we should think

Jan Steinman and Barbara Yates are co-founders of Bytesmiths, a technical services company that has been helping companies adopt Smalltalk since 1987. Between them, they have over 22 years Smalltalk experience. They can be reached at Barbara@Bytesmiths.com or Jan@Bytesmiths.com, or at their website at <http://www.bytesmiths.com>.

about servicing *real* requests! When the *TcpServer* sends *htmlForSmallDocRequest* to the socket stream, the stream may contain "GET" a space, and the URL the user entered or clicked.

There is much more information in the typical HTTP request, and this method can easily become complex. If you want to process more of the request information, be sure to factor this method into smaller methods that handle particular request information.

In particular, this method only attempts to deal with "GET" requests, which is the normal way a Web browser passively requests a Web page. This method does not handle "POST" requests, which is how a Web browser passes information entered by the remote user.

PositionableStream

htmlForSmallDocRequest

"Assume I am a bi-directional stream on a socket that is connected to a web browser. Process an incoming SmallDoc GET request."

```
| line path |
[(line := self nextLine) size > 0]whileTrue:
  [('GET ' occursIn: line at: 1) ifTrue:
   [path := line copyFrom: 5 to: line size]].

self nextPutAll: 'HTTP/1.0 200';
nextPutAll: 'Server: ';nextPutAll: (TcpServer
signatureIn: TcpServer controller); cr; cr;
(path size = 0 or: [path = '/'])
ifTrue: [self httpHomePage]
ifFalse: [self htmlSmallDocGet: path]
```

This method looks for a line in the socket stream that begins with GET, and saves the rest of the line as the URL to fetch. If the URL is empty or if it is a single slash, then some form of home page information should be sent back to the Web browser, followed by closing the stream, which lets the Web browser know the request is complete.

PositionableStream

httpHomePage

"An empty GET request is received,so give a hearty welcome."

```
self
nextPutAll: 'This is an exercise for the reader.
Put some literal HTML here (or some Text asHtml!)that
explains how to navigate through your Smalltalk
documentation repository.';
close
```

If the URL is *not* empty, there's more work to do. We follow ENVY's existing structure for navigation; if you are using some other source code management system, you will have to implement a navigation strategy for your repository.

We expect the first component of the URL to be a *naming root* that serves as a dispatcher for the remainder of the URL.

PositionableStream

htmlSmallDocGet: pathString

"Place on myself valid top-level HTML for the given <pathString>, which must begin with a slash (/), and therefore must have a size greater than zero, and must consist of URLized path from some naming root, separated by slashes.

Valid roots are:

- 1) Smalltalk,
- 2) EmUser,
- 3) EmConfigurationMap,
- 4) Application, or
- 5) SubApplication.

The second component of the path is always one of the names that a root knows about. What follows is dependent on processing by the root, which is sent the rest of the path to play with.

A new root must either be handled by this method, or it must be a global, and it must supply the methods #htmlAsRootOn:, and #htmlForPath:on:.

This does minimal error checking—it assumes a handler will catch exceptions."

```
| path root |
"Parse the path, keeping the result."
path := pathRequest splitOn: $/.
root := Smalltalk at: path first asSymbol.
1 = path size
ifTrue:
  [self nextPutAll: 'Pragma: no-cache'; cr; cr.
  root htmlAsRootOn: self]
ifFalse:
  [root htmlForPath: path on: self]
```

Now we have a "naming root" that can be used for navigation, and all that is left to do is implement *htmlAsRootOn:* and *htmlForPath:on:* so that any global can serve Web information. For example, a simple inspector can be implemented by making "Smalltalk" a naming root by implementing *htmlAsRootOn:*.

SystemDictionary

htmlAsRootOn: stream

"Place on the given <stream> HTML links for my distinguished instances."

```
stream htmlTitleAndH1: 'Smalltalk Globals'.
^(self keys asSortedCollection
inject: stream
into: [:stream :globalName | |global |
  global := Smalltalk at: global Name.
  stream
  nextPutAll: '<A HREF="/Smalltalk/';
nextPutAll: globalName; nextPutAll: ">";
nextPutAll: globalName; nextPutAll: '</A> '.
global class isMeta
ifTrue:
  [stream nextPutAll: '(a class'.
```



```
(global class instSize > Object class
instSize or: [global classPool size > 0])ifTrue:
    [stream nextPutAll: ' with state'].
    stream nextPutAll: '<BR>']
    ifFalse: [stream nextPutAll: '(an instance
of ' ; print: global class; nextPutAll: '<BR>'].
    stream]) htmlCloseBody
```

Now if `htmlForPath:on:` is implemented in *Object*, you can inspect arbitrary objects from a Web browser. This method uses a number of stream utility methods that make the task easier, by providing pre-assembled snippets of commonly used HTML.

PositionableStream

htmlBody: anObject

"Place on myself the proper HTML to make <anObject> appear as body text. This must be preceded by a 'title' statement. Answer myself."

```
self nextPutAll: '<BODY>'; htmlFor:an Object;
htmlCloseBody
```

htmlCloseBody

"Place on myself the proper HTML to close off a 'body' statement. Answer myself."

```
self nextPutAll: '</BODY></HTML>'
```

htmlTitle: string

"Place on myself the proper HTML to make <string> a title. This must be followed by a 'body' statement. Answer myself."

```
self nextPutAll: '<HTML><HEAD><TITLE>';
nextPutAll: string; nextPutAll: '</TITLE></HEAD>'
```

htmlTitleAndH1: string

"Place on myself the proper HTML to make <string> a title, followed by a 'body' statement and <string> as a top-level heading. Answer myself."

```
self
htmlTitle: string;
nextPutAll: '<BODY><H1>';
nextPutAll: string;
nextPutAll: '</H1>'
```

It would be easy to slip into gratuitous serving of all sorts of objects over the Web at this point, but we'd neglect our primary purpose: to serve Smalltalk project documentation over the Web. To do this, we need to allow *SubApplication* to function as a naming root. (Since *Application* is a subclass of *SubApplication*, this also allows *Application* to serve as a naming root.)

SubApplication class

htmlAsRootOn: stream

"Place on the given <stream> HTML links for all

subapps or apps."

```
stream htmlTitleAndH1: self name, 's'.
^(self allNames asSortedCollection
inject: stream
into: [:stream :appName |
stream
nextPutAll: '<A HREF="/'; print: self;
nextPut: $/;
nextPutAll: appName; nextPutAll: ">";
nextPutAll: appName; nextPutAll:
'</A><BR>'.
stream]) htmlCloseBody
```

Now when a URL with a naming root, such as `<http://yourhost/Application>`, is entered into a Web browser, a page is returned that lists all *Applications* in the repository, together with links that have the next path component filled in. When one of the listed *Applications* is clicked in the Web browser, the following method is sent in the SmallDoc server:

SubApplication class

htmlForPath: path on: stream

"Place HTML for my components described by <path> on the <stream>."

```
| component |
2 = path size ifTrue:
    ["This is dynamic information — do not cache it
in the client."
stream nextPutAll: 'Pragma:no-cache'; cr; cr.
self htmlEditionsForName: path last on: stream]
ifFalse:
    [(path last conform: [:ch | ch isDigit]) ifTrue:
    [path at: path size put: (Integer readFrom: path
last readStream)].
    component := (Smalltalk classAt: path first)
hrefToLibraryComponentFor: path.
    component isVersion ifFalse:[stream nextPutAll:
'Pragma: no-cache'; cr].
    stream cr.
    3 = path size ifTrue:
        [stream htmlBody:
(component commentOrTemplateIn: component)] ifFalse:
        [4 = path size ifTrue:
        [stream htmlBody: (component
commentOrTemplateIn: component application)] ifFalse:
        [5 = path size ifTrue:
        [stream htmlBody: component comment] ifFalse:
        "path size > 5 ifTrue:"
        [stream error: 'bad URL']]]]
```

This method is a case statement. Only one of the "path size" cases will be evaluated in any given invocation. Also note that if the last component of the path consists of digits, it is converted to an *Integer*.

This sends two methods that we're going to have to

continued on page 36

MANAGING OBJECTS

continued from page 24

leave you to implement yourself, due to space constraints. The *SubApplication* method `htmlEditionsForName:on:` needs to obtain all the editions for the *SubApplication* named by the second part of the path, and render them into the proper HTML so they will appear as links in the Web browser. For example, if the Web browser user typed or clicked `<http://yourhost/Application/Kernel>`, the server should place links for each edition of *Kernel* on the socket stream.

The more interesting method to complete is `hrefToLibraryComponentFor:`, which takes a collection of component parts and fetches the proper component out of the repository. For example, the URL `<http://yourhost/Kernel/Object/at:put:/3016057369>` should cause the comment for the *Object* method `at:put:` with the edition time stamp of July 29, 1996 1:42:49 am to be placed on the socket stream.

As hinted by the code, our treatment of the URL depends on its number of path parts. For an individual repository component, such as an app, subapp, class, class extension, or method, the last part of the URL path is always a second count from the component's time stamp, thus allowing you to browse version history from a Web browser. These integers are meant to be “opaque references” — the user should never have to type them in; rather, they should be part of anchors that were generated from lists of editions.

Following the example of *SubApplication*, you can now

easily add `htmlAsRootOn:` and `htmlForPath:on:` to *EmUser* and *EmConfigurationMap*, as well as any other global that you want to use as a “naming root” for serving arbitrary information from Smalltalk over the Web.


SATISFYING WITHOUT COMPROMISING

This completes our series on putting your Smalltalk project documentation on the Web. This series enables our principles of hyper-literate programming by ensuring that:

- 1) the documentation for a thing is on the same conceptual level as that thing;
- 2) the documentation for a thing constantly and accurately describes that thing;
- 3) the documentation for a thing is accessible by creators, their peers, re-users, reviewers, end-user documenters, and the merely curious; and
- 4) the documentation for a thing is measurable, quantitatively and especially qualitatively.

In addition, we hope we've shown you a few useful things about developing frameworks and automatically generating HTML.

Maintaining your documentation in your Smalltalk repository while exporting it “live” to Web browsers will satisfy the needs of external parties without compromising the efficiency of your development team.

Next month, we'll explore a topic close to our hearts — the use and abuse of Smalltalk mentors. 



Jay Almarode

Fault Tolerance

Production applications need to be protected against the possibility of catastrophic failure. Disks fill up...hardware fails...operating systems crash... networks go down...but with proper foresight these situations do not have to lead to a loss of objects. This column describes mechanisms to achieve fault tolerance and how to recover when the bad things happen.

There are at least two ways in which systems achieve fault tolerance. One is to prevent the system from going down in the first place; the other is to bring the system back to a consistent state if it does go down. The typical way to avoid a system from going down is to duplicate, or mirror, the state of the object repository on different hardware, so that if the primary piece fails, the system will automatically switch over to the duplicate. To bring the system back up when it goes down, most transaction-based systems employ backup files and transaction logs to help the system recover to a consistent state. These same approaches apply to Smalltalk applications.

In multi-user Smalltalk, the object repository is manifested by one or more files (or possibly raw disk partitions) called extents. These are where the state of objects ultimately reside. For fault tolerance, as well as performance reasons, information about objects may first be written to other files, called transaction logs. Transaction logs contain information to re-do transactions that have been committed to the repository.

When a transaction is committed, all that's necessary is to completely write the transaction log records to consider the transaction complete. The extent files do not have to be updated with new or changed objects immediately, which can improve overall system performance and transaction throughput.

To avoid a multi-user Smalltalk system from going down, the system administrator can specify that the extent files are to be replicated. In addition to allocating extent files across multiple disk drives on different machines for performance and clustering reasons, the system administrator can allocate the replicated extents

on multiple disk drives as well. While the system is running, if a client or server process should encounter a read error on a primary extent file, the corresponding replicated extent file is automatically used instead.

In GemStone, the system administrator creates replicated extents in two ways. One way is to specify them in the configuration file used by the server process at startup time. Another way is to dynamically create new replicates at runtime by sending the message `SystemRepository createReplicateOf: extentFilename named: replicateFilename`. In both cases, you are mapping a primary extent file to a corresponding replicated extent. The replicated extent should be located on a different disk spindle to reduce IO contention, as well as to provide fault tolerance.

Even if the object repository is replicated for automatic switchover, it is still good practice to plan for recovery if the system goes down entirely. This planning involves deciding how often to back up the system, and how quickly the system must be back online. For 7 x 24 production applications, it is imperative that backups be performed while the system is online and other users are logged in. Since backups may require considerable resources for large object repositories, it is desirable to limit the IO rate of the process performing the backup to reduce its interference with other sessions.

To plan for backups and recovery, it is necessary to understand how transaction logging works. As mentioned earlier, transaction logs contain the information to re-do transactions that have been committed. Transaction logs are used to recover from an unexpected shutdown or to roll forward from a backup file. When configuring a system, an administrator supplies multiple locations where transaction logs are to be written. Therefore, if one disk becomes full, the system can automatically switch over to the next location. It is also possible to configure the maximum size of each transaction log file to balance the utilization of the disk resources. Transaction logs can be replicated to provide the same benefits as replicated extents.

Recall that objects may not be written immediately to extent files. To force the information in transaction logs to be written to the extent files, an administrator performs a checkpoint. Performing a checkpoint reduces the number of transaction logs that have to be applied when the sys-

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a senior software engineer at GemStone Systems Inc., and can be reached at almarode@gemstone.com.

tem recovers from a crash, where the extent files are not damaged.

Transaction logging can be set up to handle two kinds of recovery situations. In the first situation, the system has unexpectedly shut down, but the extent files are not corrupt. To recover the object repository to the last committed state, only transaction log records that were written since the last checkpoint are applied. This mode of transaction logging is called partial logging, since not all transaction logs are needed to recover. To free up space, an administrator can remove any log files written prior to the most recent checkpoint, usually leaving the current log and the one immediately before.

In partial logging mode, the frequency of performing a checkpoint helps control how long it takes to recover the system. In GemStone, the system can be set up to automatically perform checkpoints at specific intervals by setting a configuration parameter; or, a checkpoint can be performed explicitly by sending `System checkpoint`. When the system is in partial logging mode, a checkpoint is also triggered when any transaction writes a log record whose size is greater than some configurable threshold.

The second kind of recovery situation occurs when the system crashes and the extent files are corrupt. In this case, the object repository must be recovered from backup files. To recover from this situation, all transaction logs that were written since the backup are needed. This type of recovery is supported by configuring the system to be in full logging mode. Full transaction logging should be used for production applications, to guarantee recoverability in the face of media failure.

One factor determining the time to recover from a backup is the frequency of backups performed. To perform a backup of the object repository in GemStone, a user performs the message: `SystemRepository fullBackupTo: aFileOrDevice Mbytes: aByteLimit`. The first argument specifies the file, raw partition, or device where the backup is to be created. The second argument specifies a byte limit so that you can create multiple backup files by limiting the size of each part.

When the first backup file is finally written, you continue writing the next part of the backup with the message `SystemRepository continueFullBackupTo: aFileOrDevice Mbytes: aByteLimit`. Since the backup procedure may consume system resources, a user can control the IO rate of the current backup session by sending `System configurationAt: #GemIOLimit put: 10`. This example allows a maximum of 10 IOs per second.

To restore the object repository, a system administrator first starts a server process on a new object repository. Then the restore operation is performed by sending `SystemRepository restoreFromBackup: backupFilename`. At this

point, the state of the repository is the same as when the backup file was created. Now the administrator can apply transaction logs to roll forward from the state of the backup to the state of the last committed transaction.

To find out the first transaction log file needed, the administrator sends `SystemRepository restoreStatus` to get the file id of the log file. When transaction log files are created, they are given a filename that includes an increasing numerical file id so that the sequence of file creation is evident. This helps in determining which transaction log files to archive (i.e. move somewhere else), and which are needed for restoration. If the needed transaction log files have been archived, the administrator sends `SystemRepository restoreFromLog: aTranLogFilename` to explicitly specify their location. If the remaining log files are located in their original location, then the administrator performs `SystemRepository restoreFromCurrentLogs`. The administrator sends the message `SystemRepository`

`commitRestore` to finish the restoration and allow other users to log in. It is also possible to restore to a specific point in time, by sending `SystemRepository timeToRestoreTo: aDateTime`, before restoring from transaction logs.

Using transaction logs, a 'warm' backup system can be built with the mechanisms described above. A 'warm' backup system is a duplicated object repository not kept in sync with the primary repository in

real time by the underlying system. Instead, the duplicated object repository is explicitly synchronized with the primary repository at specific time intervals. The advantage of a warm backup is that it places no burden on the primary system to perform IO to multiple locations; the disadvantage is that the warm backup is only up-to-date based on the last time it was explicitly synchronized with the primary system.

To build a warm backup system, a server process is started up on a copy of the primary object repository (or it could be started up on a new repository, then restored from a backup file of the primary repository). This is the warm backup server. Next, a process is spawned that continually looks for new transaction logs being created by the primary server.

When a new transaction log file is created, this process can copy the previous log file to the backup site and perform `SystemRepository restoreFromLog: aTranLogFilename`. If the primary repository goes down, the warm backup site performs `SystemRepository commitRestore`, and it is ready for duty.

Fault tolerance is a necessary consideration for production applications. System administrators need to plan for disaster and have the mechanisms in place to recover. Duplicated object repositories and transaction logging are two mechanisms that provide the functionality needed for 7 x 24 applications. ■

“One factor determining the time to recover from a backup is the frequency of backups performed..”



Kent Beck

Object-Oriented Recursion

Understanding recursion is a watershed in the life of most software developers. The idea that you define a computation, not in terms of other computations, but in terms of itself, is a mind bender for most people. I can remember carefully drawing stack frames with their own local storage and program counter and painstakingly following the progress of factorial and depth-first binary tree traversal. It was only when I found an obscure little book in the science library that explained how to transform recursion into iteration and vice versa, that I really felt I understood recursion. Even then, it was months before I could reliably use it as a programming technique.

Such a powerful technique must be an important part of programming objects, right? Well, yes and no. Combining recursion with objects is powerful, more powerful than its procedural counterpart, but you have to manage it differently to make effective use of it.

TAKE ONE

Rule 1: Send the recursive message to different objects. Procedural recursion is defined as a procedure that calls itself with different parameters. At some point, you have to reach the base or degenerate case of the recursion, at which time you do not call the procedure further (not if you want the program to terminate, anyway). Factorial implemented with procedural-style recursion looks like this:

```
Object>>factorial: aNumber
^aNumber = 1
ifTrue: [1]
ifFalse: [aNumber * (self factorial: aNumber - 1)]
```

In this version of #factorial: the receiver of the message plays no particular role. The existence of the receiver of a message as the implicit first parameter motivates the first change in the use of recursion with objects. Rather than invoke what is in essence a subroutine over and over on the same object with different parameters, object-oriented recursion invokes the same routine, but with different objects as the receiver. The object-oriented version of factorial doesn't need an additional parameter. The receiver of the message is the number to be "factorialed."

```
Number>>factorial
^self = 1
ifTrue: [self]
ifFalse: [self * (self - 1) factorial]
```

The resulting code is simpler by one argument, but otherwise looks much like the procedural version.

WE PAUSE FOR A BIT OF MATHEMATICS

To illustrate the other difference between procedural and object-oriented styles of recursion, we will turn to Peano's Axioms of Arithmetic. Zero is represented as "zero." Other positive numbers are defined as nested invocations of the function "succ" (for "successor"). For example, three is represented as:

```
succ(succ(succ(zero)))
```

Given this definition of numbers, we can now define addition recursively. The base case of the recursion is adding any number to zero, equals that number:

Case 1: $\text{add}(X, \text{zero}) = X$

Thus, adding zero and three results in three:

```
add(succ(succ(succ(zero))), zero) = succ(succ(succ(zero)))
```

The recursive case of the definition says that adding X to a number which is the successor of Y is the same as adding X to Y , then getting the successor of the sum.

Case 2: $\text{add}(X, \text{succ}(Y)) = \text{succ}(\text{add}(X, Y))$

Algebraically, this is the same as saying:

$$X + (1 + Y) = 1 + (X + Y)$$

Adding two to one results in the following invocations:

```
add(succ(zero), succ(succ(zero))) = succ(add(succ(zero), succ(zero))) by case 2
succ(add(succ(zero), succ(zero))) = succ(succ(add(succ(zero), zero))) by case 2
succ(succ(add(succ(zero), zero))) = succ(succ(succ(zero))) by case 1
```

Lo and behold, $2 + 1 = 3!$

AXIOMS TO OBJECTS

We can turn Peano's Axioms into objects by making a successor object, which is linked to its predecessor. A

linked list of three successors represents the number three. The end of the list will be represented by nil.

```
Class: Succ
  superclass: Object
  instance variables: pred
```

We can provide a Constructor Method for Succ that returns the Peano version of an Integer:

```
Succ class>>fromInteger: anInteger
  ^anInteger = 0
    ifTrue: [nil]
    ifFalse: [self of: (self fromInteger: anInteger - 1)]
```

We create the successor of a Peano number by creating a new instance of Succ and setting its predecessor to the number.

```
Succ class>>of: aPeanoNumber
  ^self new setPred:aPeanoNumber
Succ>>setPred: aPeanoNumber
  pred := aPeanoNumber
```

We compute the predecessor of a Peano number by simply returning the value of the instance variable "pred."

```
pred
  ^pred
```

We compute the successor by tacking on another successor object:

```
succ
  ^Succ of: self
```

For debugging purposes, we can define a printing method that shows us the receiver in Peano format.

```
Succ>> printOn: aStream
  aStream nextPutAll: 'succ('
  self pred isNil
    ifTrue: [aStream nextPutAll: 'zero']
    ifFalse: [self pred printOn: aStream].
  aStream nextPutAll: ')'
```

Three now prints as three nested invocations of "succ":

```
(Succ fromInteger: 2) + (Succ fromInteger: 1) succ(succ(succ(zero)))
```

Given the definitions of #pred and #succ, we can turn the axioms of arithmetic into a method. Because we are explicitly checking for the base case of the recursion, the code is not quite a direct translation of the original axioms.

```
+ aPeanoNumber
  |subTotal|
  subTotal:= self pred isNil
    ifTrue: [aPeanoNumber]
    ifFalse: [self pred+ aPeanoNumber].
  ^ subTotal succ
```

Adding two and one result in our now famous three:

```
(Succ fromInteger: 2) + (Succ fromInteger: 1) succ(succ(succ(zero)))
```

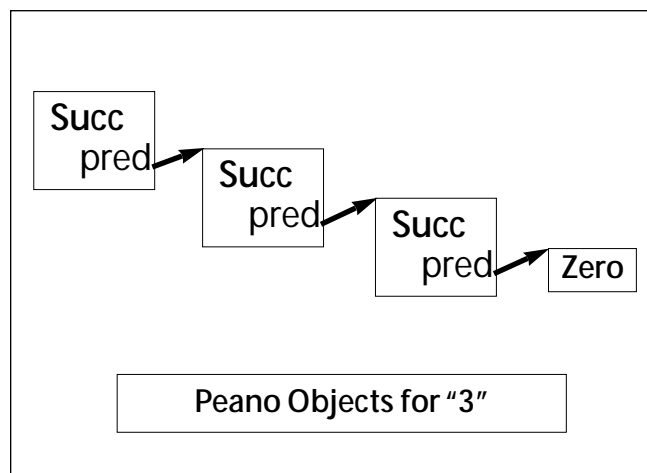


Figure 1. The number three, represented as objects.

Take Two

Rule 2: Represent the base case of the recursion by a distinct object. Now we are finally ready to examine the second difference between procedural and object-oriented recursion. Procedural recursion relies on explicit checks for the base case of the recursion. The previous code shows this style in the #+method, where an explicit conditional checks for a nil argument.

Every time I use recursion in the beginning, and distressingly often thereafter, I forget to check the base case. You can use objects and messages to make such errors less likely, and to simplify the code at the same time.

The key is not to rely on the generic undefined object to stop the recursion. Instead, you create your own "undefined object," then make sure it responds to the same messages as the object representing the recursive case.

To apply this principle here, we have to first replace nil with a new object, Zero.

```
Class: Zero
  superclass: Object
  instance variables: <none>
```

Rather than returning a nil when we want to represent a zero, we return an instance of our new object instead:

```
Succ class>>fromInteger:anInteger
  ^anInteger=0
    ifTrue: [Zero new]
    ifFalse: [self of: (self fromInteger: anInteger - 1)]
```

Adding a Zero and any number results in that number:

```
Zero>>+ aPeanoNumber
  ^aPeanoNumber
```

Adding a successor to a number now need not check for nil:

```
Succ>>+ aPeanoNumber
  ^(self pred + aPeanoNumber) succ
```

Once again, we can add two and one to get three:

```
(Succ fromInteger: 2) + (Succ fromInteger: 1) succ(succ(succ(a Zero)))
```

Notice that the zero prints out a little differently than before. We can easily fix that:

```
Zero>>printOn: aStream  
aStream nextPutAll: 'zero'
```

And we can get rid of the code in Succ that checks for nil. I love bug fixes that involve removing code!

```
Succ>>printOn: aStream  
aStream nextPutAll: 'succ(';  
self pred printOn: aStream.  
aStream nextPutAll: ')'
```

Now two plus one prints correctly again:

```
(Succ fromInteger: 2) + (Succ fromInteger: 1) succ(succ(succ(zero)))
```

The addition code looks much more like the original mathematics (taking postfix notation into account):

```
Case 1: add(zero, X) = X  
Case 2: add(succ(X), Y) = Y = succ(add(X, Y))
```

```
Zero>>+ aPeanoNumber  
^aPeanoNumber  
Succ>>+ aPeanoNumber  
^(self pred + aPeanoNumber) succ
```

We have been able to use polymorphism to write code that communicates more clearly, because it translates more directly from the original source. The code is more like a specification and less like a computer program.

The Two Ways

We have seen two ways in which object-oriented recursion differs from procedural recursion. First, rather than invoke the same procedure with different arguments, object-oriented recursion represents the invocations themselves as objects, sending the same message to different objects.

In our example, this corresponded to creating a new object to represent one invocation of the successor function. Most recursive routines don't require this (somewhat


unnatural) step. If the recursive routine is operating over a recursive data structure (trees or lists, for example), the objects are likely to be there already.

The second difference between procedural and

“object-oriented recursion represents the invocations themselves as objects, sending the same message to different objects.”

object-oriented recursion is in the use of a special-purpose object to represent the base case of the recursion. Polymorphism's ability to capture decision making in what would otherwise be a simple procedure call comes to the fore in this technique. The resulting code communicates, clearly even in the absence of explicit conditionals.

You might ask, “Why don't you use Smalltalk's built-in special object, nil, to represent the base case of the recursion?” After all, in the example above, we could implement #+ in UndefinedObject just as we did in Zero and the code would work fine. The problem is that all developers share the same UndefinedObject. If everyone added a handful of methods to it, the result would be thousands of methods on UndefinedObject, in other words, chaos. The chances of such code communicating clearly, are slim, even if there weren't accidental disagreements about what UndefinedObject>>+ should do.

If you'd like to play around with recursion, you may want to extend the code above. Try implementing #- or #*. I found implementing negative numbers (hint, you need a Pred class) to be quite challenging. 

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

Editors' Corner



John Pugh



Paul White

WE WRITE THIS as we watch the Olympic Games draw to a close. It is absolutely amazing to witness the accomplishments of the athletes in attendance, and to try to comprehend the dedication they each have to their sport. (We're also still reveling over the Canadian gold medal in the 4x100 relay!) The other amazing feat, from our perspective, is the advances in technology that were on display during the Games. Of course, the problems experienced were well documented and many criticized the technology as being a failure (sound familiar to conversations with your users?), but in many ways the criticisms are unfair, given what has been attempted. It's another case of unrealistic expectations coupled with an unwillingness and/or lack of opportunity to manage those expectations.

We have been told that the software applications built by IBM using VisualAge fared fairly well at the Games, but we have no confirmation of this fact. In any event, I think if the public stopped to realize just how far engineering has advanced in such a short time, they would begin to see that things are heading in the right direction. One might even draw the conclusion that we must be getting closer to building useful technology, considering people have much stronger opinions as to what is being built. In any event, hats off to everyone involved!

Last month we touched on the problems faced by many of us in managing the persistence aspect of our applications. While for most, persistence is undoubtedly the number one enemy in today's applications, there are certainly other common problems facing a large majority of us. One of the items heading this list is report generation.

Report generation has always been, and will no doubt remain, an extremely difficult problem. Deciding how to best utilize an 8-1/2 x 11 sheet of paper is an almost impossible task. Its limitations are obvious—there is a fixed-size region in which to print, and the items we wish to print do not have such limits. Features such as proportional fonts and text formats cause even more grief than what was faced by those true warriors of report generation over the past decades—the RPG programmers.

There are two aspects to the problem. The first is the technology for laying out reports, specifying the

desired behavior and the final 'look-and-feel' of the ultimate report. On this front, the news continues to get better. Both ParcPlace-Digitalk and IBM provide some sort of facilities for specifying reports, as do a handful of third-party vendors. Each of these has both strong and weak points, but the features being introduced are allowing report designers to work at a better level of abstraction for describing reports.

The second aspect of the problem comes from the application itself. Actually attempting to describe what is the desired behavior, from an application point of view, is a monumental task. Each report, of course, needs a header and a body (and perhaps a footer), but the amount of space required may vary,

depending upon the type of report and the contents contained within it. Deciding how to split a report that must run multiple pages is a real challenge. What's more, if the report contains cross references to other items within the report (as is

commonly found in insurance claims forms, for example), an automated process for describing this is extremely elusive.

In the end, one must remember that the problem being addressed is in many ways not solvable, at least not easily. The proof of this is simple: Just consider describing how to draft a report, describing how to handle every single possible configuration for the next five years, guaranteeing you haven't missed any case. If you can't solve it in English, you can't solve it using any modern computer language either. Having said this, what we need to do is provide much better tools to allow users to describe their solutions using abstractions, which make sense in terms of a report layout. We believe these tools will continue to improve.

Finally, we should note this issue represents an accomplishment we never envisioned when we began. When we started *The Smalltalk Report* in September 1991, we did it with the hope that it would help people like you to utilize Smalltalk more effectively, to serve as a place to share ideas, and hopefully, in some small way, help the Smalltalk industry grow and mature. After a full five years of it, we think we can stand up and say we've achieved this goal. And we certainly look forward to writing an editorial after our 10th year of the *Report!*

Enjoy the issue.

*Deciding how to split
a report that must run multiple
pages is a real challenge.*

Editors

John Pugh and Paul White
Carleton University & The Object People

SIGS Publications Advisory Board

Tom Atwood, *Object Design*
François Bancelhon, *O₂ Technology*
Grady Booch, *Rational*
George Bosworth, *ParcPlace-Digital*
Jesse Michael Chonoles, *Lockheed Martin ACC*
Stuart Frost, *SELECT Software*
Adele Goldberg, *ParcPlace-Digital*
Thomas Keffler, *Rogue Wave Software*
R. Jordan Kriendler, *IBM Consulting Group*
Thomas Love, *Consultant*
Bertrand Meyer, *ISE*
Meilir Page-Jones, *Wayland Systems*
Bjarne Stroustrup, *AT&T Bell Labs*
Dave Thomas, *Object Technology International*

The Smalltalk Report Editorial Board

Jim Anderson, *ParcPlace-Digital*
Adele Goldberg, *ParcPlace-Digital*
Reed Phillips
Mike Taylor, *ParcPlace-Digital*
Dave Thomas, *Object Technology International*

Columnists

Jay Almarode, *GemStone Systems Inc.*
Wayne Beaton, *The Object People*
Kent Beck, *First Class Software*
Dwight Deugo, *The Object People*
Juanita Ewing, *ParcPlace-Digital*
Bob Hinkle, *Consultant*
Tim Howard, *FH Protocol, Inc.*
Ralph E. Johnson, *University of Illinois*
Alan Knight, *The Object People*
Mark Lorenz, *Hatteras Software, Inc.*
Jan Steinman, *Bytesmiths*
Rebecca Wirs-Brock, *ParcPlace-Digital*
Barbara Yates, *Bytesmiths*

SIGS Publications Group, Inc.

Richard P. Friedman, Founder, President, and CEO
Chris Keating, Publishing Director—US Magazines
John McCormick, Editorial Director
Margherita R. Monck, General Manager

Editorial/Production

Kristina Joukhadar, Senior Managing Editor
Elisa Varian, Director of Manufacturing
Jan Foster, Cover Design
Douglas Finlay, Associate Managing Editor
Serena Tesler, Production Editor
Erika Romero, Desktop Designer
Margaret Conti, Manufacturing Coordinator

Circulation

Elayne Glick, Circulation Director
Byron Scarlett, Assistant Circulation Manager

Advertising/Marketing

Gary Portie, National Sales Manager
Elisa Marcus, Advertising Manager, Central US
Michael W. Peck, Advertising Representative
Kristine Viksnins, West Coast Exhibit Sales
Sarah Olszewski, East Coast Exhibit Sales
212.242.7447 (v), 212.242.7574 (f)
Nancy Beuschel, Promotions Manager for Magazines

Administration

Margherita R. Monck, General Manager
David Chatterpaul, Senior Accounting Manager
Bibi Budhram, Accounts Payable



SIGS PUBLICATIONS
Publishers of JOURNAL OF
OBJECT-ORIENTED PROGRAM-
MING, OBJECT MAGAZINE,
C++ REPORT, THE SMALLTALK

REPORT, THE X JOURNAL, JAVA REPORT, OBJECT CURRENTS and
THE X SPOT (ONLINE), OBJECT EXPERT (UK), JAVASPEKTRUM and
OBJEKT SPEKTRUM (GERMANY)



For more object
news and analysis,
check out SIGS
online at
<http://www.sigs.com>

Features:

Unblocking the Debugger

3

Joe Winchester and Mark Jones

Here is an enhancement to the IBM Smalltalk/Visual Age debugger that allows code within blocks to be debugged more easily.

Externalizing Business-Object Behavior: More On A Point-and-Click Rule Editor

8

Paul Davidowitz

In this issue the author continues to investigate the inner workings of a point-and-click rule editor, which can be tailored by the end-user.

Just Cloning Around Subclass: # Cloning Extensions

13

Keith Piraino

All applications wrestle with the problem of copying objects at some point. Although Smalltalk does not provide a way to distinguish between aggregation relationships and simple references, the author offers a solution.

Columns



Visual programming: Visual Programming and Reusable Parts: The Marquee Part

18

Wayne Beaton and Dwight Deugo

The authors create a reusable marquee part, which is responsible for scrolling text that is too large for static display.



The Best of comp.lang.smalltalk Two New Smalltalks

22

Alan Knight

Prerelease looks at Object Connect's Smalltalk MT and Intuitive Systems' Dolphin Smalltalk.



Managing Objects: Mentoring

24

Jan Steinman and Barbara Yates

Not all Smalltalk gurus are cut out to be mentors. And not every team member wants to be mentored. Here are some ideas about how to make the most of mentors.

Departments

Editors' Corner

2

Product News:

30

Smalltalk Broker, IBM connections to Notes and the Web, Dynamic Links for code navigation, GUI components.

Recruitment

32

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar-Apr, July-Aug, and Nov-Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1996 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Bulk rate U.S. postage paid Lancaster, PA, permit 161. Canada Post International Publications Mail Product Sales Agreement No. 290386.

Individual Subscription rates 1 year (9 issues): domestic \$89; Mexico and Canada \$114; Foreign \$129; Institutional/Library rates: domestic \$199; Canada & Mexico \$224; Foreign \$239. To submit articles, please send electronic files on disk to the Editors at 885 Meadows Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

POSTMASTER: Send domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sigs.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

Unblocking the Debugger

Joe Winchester and Mark Jones

Debuggers are evolving from tools that trap and help diagnose errors into an integral part of the development environment. Their most useful task is arguably the ability to step through code, line by line, allowing the programmer to follow message flow. The debugger supplied with IBM's Smalltalk/Visual Age falls short of this purpose where the code is embedded within blocks. To locate this code often requires traversing the internals of seemingly irrelevant methods. In this article we describe an enhancement to the IBM Smalltalk/Visual Age debugger that allows code within blocks to be debugged more easily.

Our approach to solving the problem first involved understanding the mechanics of the existing debugger, and then observing and automating how a programmer manually debugs his or her way through methods that contain code as blocks.

THE WAY THINGS ARE

The supplied debugger comes with four buttons: *into*, *over*, *return*, and *resume*. Pressing *into* allows the next message about to be sent to be debugged, and pressing *over* allows the message to be skipped past. When the message contains a block as an argument or receiver, pressing *over* skips past the message and past the code contained within the block. There are some exceptions to this, for example: `Integer>>to: anInteger do: []`, and `Boolean>>ifTrue: []`. Pressing *over* on these messages takes the programmer to the code within the block. This happens because the compiler inlines the code to implement messages directly in the sender for optimization purposes, and no actual message sent occurs at runtime. With all other messages that contain blocks as receivers or arguments, debugging the code within the block requires pressing *into* once, *over* seven times, and a final *into* to reach the block. If the message being debugged is a loop, e.g. `Collection>>detect: [] ifNone: []`, repeated debug-

ging of the method internals is required to visit the code within the block for each iteration of the loop. The problem here is that you should be concentrating on your code, instead of worrying about reading and understanding the internals of methods that are basically language constructs.

THE WAY THINGS SHOULD BE

We found ourselves wishing for behavior that would act the way *over* does for the inlined messages, i.e. it would always take the programmer straight to the code within the block. This functionality we decided to name *through* and implement with a new button on the debugger. Figure 1 shows how *over* takes the programmer past the code within the `Dictionary>>at: aKey ifAbsent: []` block. Pressing *through* would go into the code within the block and back to the method body without having to traverse irrelevant method internals.

MIMIC REAL LIFE

As with any good OO solution, the answer lies in observing the real world and mimicking it in code. Therefore, let us examine the scenario "How does a programmer debug the method `Dictionary>>at: aKey ifAbsent: []`." If the pro-

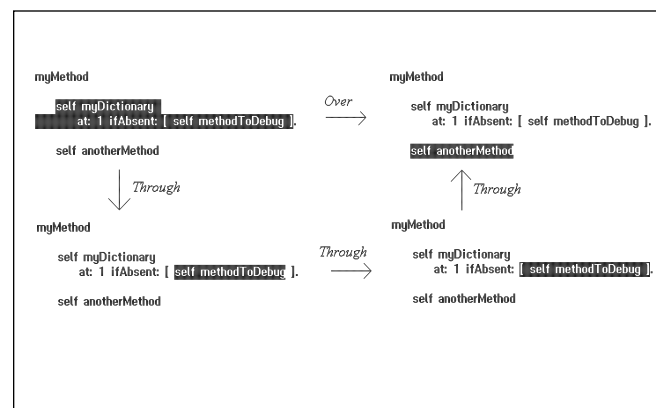


Figure 1. Depicting how *over* takes the programmer past the code within the `Dictionary>>at: aKey ifAbsent: []` block.

grammer wishes to reach the code within the block, they step into the method. An inexperienced programmer might then continue to step into each successive message until the block's embedded code is reached. A more experienced programmer knows to only examine messages that are likely candidates to evaluate the block. In both cases, the programmer succeeds once the original method that contains their source becomes active again.

We shall attempt to automate the behavior of the rookie Smalltalker. The key to our solution will come from an understanding of the mechanics behind the debugger and attempting to leverage this to implement the request "Step into a message with a block and continue to step into each successive message send until the code within the block is about to be executed."

DEBUGGER BASICS

Source code written in Smalltalk is compiled into instances of `CompiledMethod`, each of which contains a stream of bytecodes. Each bytecode is an intermediate representation of an instruction that is interpreted by the virtual machine. Instances of class `Process` execute code, and the distinguished instance of `ProcessorScheduler` is responsible for controlling process execution.^{1,2,3}

The supplied debugger runs in its own process separate from the method being debugged. It uses the method,

```
ProcessorScheduler>>debugResume: aProcess when:
aConditionBlock do: aContinueBlock,
```

to cause the process being debugged to execute until *aConditionBlock* evaluates true. Once this has occurred *aContinueBlock* is evaluated. While *aConditionBlock* is being tested the debugger must wait until *aContinueBlock* is evaluated. This is achieved with a semaphore, which the process that the debugger executes waits on, and is signaled by the *ContinueBlock*. To ensure that *aConditionBlock* is tested between each message send, the messages `breakEveryByteCode: true` and `useByteCodeMask: true` to the process being debugged. Each process has a number of frames that can be thought of as a stack of `CompiledMethod` instances.

THE ENHANCED DEBUGGER

To begin adding our new behavior we will subclass the supplied debugger and provide a new button "Through."

```
EtDebugger subclass: EnhancedDebugger
instanceVariableNames: 'throughButtonWidget'
classVariableNames: ''
poolDictionaries: ''
```

To automate our rookie's behavior we will use `ProcessorScheduler>>debugResume: aProcess when: aConditionBlock do: aContinueBlock` and construct *aConditionBlock* that evaluates true when the method being debugged is active once more. A process can derive its active method using its start frame:

```
Process>>#activeMethod
```

```
^self methodAtFrame: self startFrame
```

It is equally important to check that it is still possible to return to the method in which *through* is being processed. This is because, during execution, a return may have been processed, causing the method to be exited.

The test can be accomplished by determining if the method is still in the frame stack.

```
Process>>#cannotReturnTo: aMethod
```

```
"Loop around the frames and get the method at
each. Return false
if any of these are the method argument"
```

```
self startFrame to: self numberOfFrames - 1 do: [ :index |
    ( self methodAtFrame: index ) == aMethod
    ifTrue: [ ^false ] ].
^true
```

We are ready to implement a method on the enhanced debugger to process *through*.

```
EnhancedDebugger>>#processStepThrough
```

```
| sem currentProcess currentMethod |
```

```
"don't continue if selected process is not resumable"
self isSelectedProcessResumable ifFalse: [ ^self ].
```

```
"don't continue if a source change has been specified"
self changeRequest ifFalse: [ ^self ].
```

```
"ensure there is a selected method"
self selectTopFrameIfNone.
currentProcess := self selectedProcess.
currentMethod := self selectedMethod.
```

```
"Break every byte code of the process and resume execution
until the method is active once more or is not in the method
stack"
```

```
currentProcess
    breakEveryBytecode: true ;
    useBytecodeMask: true.
sem := Semaphore new.
Processor
    debugResume: currentProcess
    when: [
        currentProcess activeMethod == currentMethod or: [
            currentProcess cannotReturnTo: currentMethod ] ]
    do: [ :hasStackOverflowOccured |
        hasStackOverflowOccured
            ifTrue: [ self removeProcess: currentProcess ].
        sem signal ].
sem wait.
```

```
"refresh debugger"
self isOneProcessSelected ifTrue: [ self refreshAfterStep ].
```

THROUGH VERSUS OVER

This implementation automates how the rookie program-

mer debugs methods with *source* as block arguments. If *through* is used instead of *over* to debug a method containing a block with inlined *source*, the debugger takes the programmer straight into the code within the block. Once inside the block *over* can be used to step past each message send. However, it becomes easy to break out and into the method internals we are attempting to bypass, by pressing *over* after the last message inside the block where *through* should have been pressed. With methods that loop such as `Collection>>detect: [] ifNone: []`, switching between *through* and *over* to debug each iteration became an unnecessary distraction, the very thing we are trying to remove. The *through* button is needed to perform only our new processing when it is required, and to process *over* otherwise. This way the programmer can repeatedly press *through* to visit every message send within their source, with the *over* button being the exception required to genuinely skip past the source within a block.

A BETTER SOLUTION

We shall improve our solution by repeating the earlier principle of observing real life and mimicking it in code. Understanding as to whether *through* or *over* is required (when using the debugger) comes from looking at the highlighted portion of the source.

The three situations when *through*, rather than *over*, is required are:

- Sending a message in which the receiver is a block containing inline source;
- Sending a message in which any one of the arguments is a block containing inline source; and
- Leaving a block on completion of the inline source.

A segment of highlighted source is equivalent to a parse node. Different classes of parse node represent different code constructs. Thus, we will ask each parse node `#isThroughRequired` and specialize to recognize the three situations.

The superclass of all parse nodes is `EsParseNode`. By default *through* is not required.

```
EsParseNode>>#isThroughRequired
```

```
^false
```

The node representing the first two of our code constructs is `EsMessageExpression`. The decision is deferred to the receiver node to determine if it is a block and to the message pattern node to determine if any argument is a block.

```
EsMessageExpression>>#isThroughRequired
```

```
^self receiver isThroughRequired
or: [ self messagePattern isThroughRequired ]
```

The block node is `EsBlock`.

```
EsBlock>>#isThroughRequired
```

```
^true
```

The message pattern node for a keyword message is `EsKeywordPattern`. The decision is deferred to each argument node to determine if any are blocks.

```
EsKeywordPattern>>#isThroughRequired
```

```
self arguments
detect: [ :anArgument |
    anArgument isThroughRequired ]
ifNone: [ ^false ].
^true
```

The third of our situations is already covered by the block node `EsBlock`.

BACK TO THE ENHANCED DEBUGGER

The next step is for the debugger to determine the parse node for the next message to be sent within the currently selected method.

```
EnhancedDebugger>>#currentParseNode
```

```
^self parseTree == nil iffFalse: [
    self parseTree
    nodeWhichContainsPC: (self currentPC: self
selectedFrameIndex )
hasDropped: self selectedProcess hasDropped ]
```

Finally, we are ready to implement a method on the debugger that will perform *through* processing when required and *over* at all other times. This is the method that will be called by the *through* button, allowing the user to have a single button that permits them to visit every message sent within their source.

```
EnhancedDebugger>>#processThrough
```

```
| aNode |

aNode := self currentParseNode.

"process as through if selected method is the active method"
(self selectedFrameIndex ==
self selectedProcess startFrame

"and, if node requires through"
and: [ aNode notNil and: [ aNode isThroughRequired ]])
ifTrue: [ self processStepThrough ]
ifFalse: [ self processStepOver ]
```

CHANGING THE USER INTERFACE

The new '*through*' button is placed between the existing '*into*' and '*over*' buttons. To achieve this the method `#createWorkRegion` is specialized as follows:

```
EnhancedDebugger>>#createWorkRegion
```

```
super createWorkRegion.
```

```
throughButtonWidget := self
```



Figure 2. Represents a method that shows the effectiveness of the *through* button.

```
newButtonWidget: 'Through'
selected: #processThrough.

self stepIntoButtonWidget setValuesBlock: [ :w |
w rightWidget: throughButtonWidget ].

throughButtonWidget setValuesBlock: [ :w |
w
topAttachment: XmATTACHOPPOSITEWIDGET;
topWidget: self stepToReturnButtonWidget;
leftAttachment: XmATTACHPOSITION;
leftPosition: 11;
rightAttachment: XmATTACHWIDGET;
rightWidget: self stepOverButtonWidget;
bottomAttachment: XmATTACHWIDGET;
bottomWidget: self textWidget parent].

self stepOverButtonWidget setValuesBlock: [ :w |
w leftPosition: 22 ].

self stepToReturnButtonWidget setValuesBlock:
[ :w | w leftPosition: 33 ].

self resumeButtonWidget setValuesBlock: [ :w |
w leftPosition: 44 ].
```

To switch over to the enhanced debugger, the following method must be evaluated on the Transcript.

```
System startUpClass debuggerClass: EnhancedDebugger
```

Evaluating with EtDebugger will reset to the original supplied debugger.

LIMITATIONS

The implementation now combines the way the rookie programmer debugs methods containing source inlined as blocks, with the decision process made by the more experienced programmer who knows when such debugging is required.

However, there are two situations to be aware of:

- Exceptions raised while *through* processing in the receiving block of methods such as `Block>>#when:do:` do not lead to stepping *through* the exception handling block code. This situation occurs because *through* processing is not required to debug all code constructs within a block. In these cases the exception is raised during the supplied *over* processing. Therefore, the exception handling block is not stepped *through*. Due to a feature of the supplied implementation of *over* if an exception is raised, the debugger reactivates prematurely resulting in a mismatch between the method being displayed in the debugger and the active method.
- A performance problem exists if many messages are sent between the point when *through* processing starts and the block that caused *through* processing to be required is evaluated. This occurs because *through* processing involves executing the intervening code one bytecode at a time, and this has a performance overhead. Methods where this is noticeable are rare, and are generally outside the standard block messages included with the core IBM Smalltalk. A hypothetical method of the type that would be affected is `DataBaseFile>>readAt: aKey ifNoRecordFound: []`. Here, many messages will potentially be sent before the block is evaluated.

CONCLUSIONS

The enhanced debugger fulfills all the initial goals, namely to enable the developer to focus on tracing the execution of the code in which they are interested. To see how effective it can be try to debug the method shown in Figure 2, with and without the use of the *through* button. All of the code required to implement the debugger is included in this article, and can also be downloaded from the Visual Age CompuServe member-supplied forum, and from ftp://ftp.smalltalk.com/pub/ibm.smalltalk/win_debug.zip. Having used the *through* button for some time we now find it to be an invaluable aid to debugging. We hope you will find it as useful, and we welcome all feedback.

The authors would like to thank Tim Morrison of Unity Software and David Cotton for their input in preparing this article, as well as Doug Shaker of The Smalltalk Store for putting the source code on his ftp server. **S**

References

1. Goldberg, Adele, and Robson, David, Smalltalk-80: The Language, Addison Wesley ISBN 0-201-13688-0.
2. IBM Smalltalk Programmer's Reference, SC34-4493-02.
3. Budd, Timothy, A LITTLE SMALLTALK, Addison-Wesley ISBN 0-201-10698-1, 1987.

Joe Winchester and Mark Jones are consultants working for Computec International, Costa Mesa, CA. They are currently working on building application frameworks using Visual Age for a healthcare company. They can be contacted at 103276, 233@Compuserve.com.

Externalizing Business-Object Behavior: More on a Point-and Click Rule Editor

Paul Davidowitz

In the previous article (*Smalltalk Report*, September, p. 4), we introduced a point-and-click rule editor that manipulates the ProgramNode tree. We continue to investigate how it works.

TYPING

The tool's type system is based on the *types as sets of classes* approach used by Graver and Johnson.^{1,2} A ProgramNodeType is defined as a set of zero or more ProgramNodeTypeOptions. A ProgramNodeTypeOption has a class name and a *taxonomy*. *Taxonomy* indicates whether an exact match with the class is required (isMember) or whether a subclass can also be used (isKind). For example, a type option for Boolean or Number has the isKind taxonomy because these are abstract classes. The ProgramNodeType is depicted with angle brackets, with a vertical bar between options, as in <ByteSymbol | (kindOf: Number)>; the vertical bar is read as or. If kindOf is notated, the taxonomy is isKind; otherwise, it is isMember.

A type can be said to satisfy another type. For example, <kindOf: Number> is satisfied by <Integer>; and <ByteSymbol | ByteString> is satisfied by <ByteSymbol>. Two types can be merged by adding their unique type options.

There are two special types, <'Anything'> and <'Nothing'>. <'Anything'> is shorthand for <kindOf: Object>. <'Nothing'> has zero options and is used exclusively for either an out-of-scope condition, or for an unknown type inside a nonevaluated block. To illustrate:

```
[ : aBoolean | | t1 |
  t1 := 'abc'.
  t1 "<'Nothing'> out-of-scope" := aBoolean
  ifTrue: [^123]
  ifFalse: [^#xyz]].
[ | t1 |
  t1 := 'abc'.
  [t1 "<'Nothing'> non-evaluated-block"]]
```

Selector Information. One of the MorphConstructs a MessageNodeWrapper knows how to perform is *Change mes-*

sage selector and arguments only (123 + 456 → 123 * nil). The user is usually presented with a set of this construct, with each element of the set having selector information that includes the selector, argument types (if any), and the return type. Where does this set of selector information come from?

For a given type option of the receiver, if the type-option's class is not a business object, the information comes from hardcoded information on the base classes; otherwise it is generated from the business-object's logical-schema attribute/type specification. The set of selector information presented to the user is the intersection of sets from all the type options; therefore, for distant options this may mean selector information from Object only.

Let's say the selection is a MessageNodeWrapper of receiver SmallInteger, for example, 123 isNil. The receiver knows its type (<SmallInteger>), so we ask the class, whose name we get from the single type option, for selector information. This information will come from SmallInteger class and its superclasses.

This is an example of selector information from the class side of Number:

```
^TypedSelector
  singleArgumentSelector: #<
  receiverRequiredType: ProgramNodeType number
  argumentType: ProgramNodeType number
  returnType: ProgramNodeType boolean
```

The selector, required type for receiver, required type for argument, and return type are all specified (via an instance of class TypedSelector). Selector information is inherited and can be overridden. It is specified for operators of those base classes which are used as business-object attribute types (e.g. Boolean, ByteString, Number).

OTHER TYPE OPTIONS

ProgramNodeTypeOption lives in a hierarchy:

AbstractProgramNodeTypeOption

ReifiedBlockValueTypeOption (argumentIndex)

ProgramNodeTypeOption (className, taxonomy)

```
BlockClosureTypeOption (valueType,  
argumentTypes)  
HomogeneousCollectionTypeOption  
(elementType)
```

HomogeneousCollectionTypeOption describes a homogeneous collection, meaning all elements have the same elementType. *Homogeneous* can be very flexible, as the type can be <'Anything'>, for example. This type option takes selector information from the instance side, as well as the class side. An instance of the collection is created and given one element: a copy of elementType.

Here's an example of selector information from the *instance* side of Collection:

```
^TypedSelector  
  singleArgumentSelector: #select:  
  receiverRequiredType: (ProgramNodeType  
    homogeneousCollectionOfName: self class name  
    elementType: self first copy)  
  argumentType: (ProgramNodeType  
    blockClosureSingleArgument: self first copy  
    valueType: ProgramNodeType boolean)  
  isArgumentPrototyping: true  
  blockArgumentEvaluator:  
    MessageWrapperBlockArgumentEvaluator nothingOrLoop  
  returnType: (ProgramNodeType  
    homogeneousCollectionOfName: self class name  
    elementType: self first copy)
```

The argument of the #select: message is specified as a block whose single argument is a copy of elementType. The return type is specified as a collection of this type. Note that this selector information specifies using a prototype argument; that is, the argument instead of being nil will be a block with an argument of the correct type.

ReifiedBlockValueTypeOption is used to determine the return type of a message. Here's the selector information for #ifTrue: from the class side of Boolean:

```
^TypedSelector  
  singleArgumentSelector: #ifTrue:  
  receiverRequiredType: ProgramNodeType boolean  
  argumentType: (ProgramNodeType  
    blockClosureNoArgumentsAndValue:  
      ProgramNodeType anything)  
  isArgumentPrototyping: true  
  blockArgumentEvaluator:  
    MessageWrapperBlockArgumentEvaluator onOrOff  
  returnType: (ProgramNodeType options:  
    (OrderedCollection  
      with: (ReifiedBlockValueTypeOption onArgumentIndex: 1)  
      with: ProgramNodeTypeOption nil))
```

This states that the return type for #ifTrue: consists of type options for nil, and for the value of the block expected as the message's first argument. For example, the type of message aBoolean ifTrue: [123] is <UndefinedObject | SmallInteger>; for aBoolean ifTrue: ['abc'], it is <UndefinedObject | ByteString>.

TRAVERSAL OF THE WRAPPER TREE

Traversal of the wrapper tree yields valuable information such as node-wrapper type. Traversal is done in postorder fashion.

A wrapper knows the order in which to traverse its children. For example, A MessageNodeWithArgumentsWrapper has the child traversal order: {receiver, argument-collection}.

The Block Evaluator. If at least one of the arguments of a MessageNodeWithArgumentsWrapper is a block defined (via selector information) as potentially evaluating, we append pseudo-child MessageWrapperBlockArgument Evaluator to the child traversal order (*pseudo* in the sense that the MessageNode itself has no such child). The evaluator serves to simulate evaluation of a block by traversing it. Without the evaluator, the BlockNodeWrapper is treated as a leaf wrapper and is not traversed. The evaluator poses as having the block-argument grandchildren of its parent, as its own children.

The evaluator is defined with a collection of EvaluationMetaSpecs. An EvaluationMetaSpec is a description for one step through of the method specified in the message. This spec is used to produce a collection of child traversal orders. An EvaluationMetaSpec, in turn, is defined with a collection of ArgumentMetaSpecs. The ArgumentMetaSpec states whether the block argument is optional, and whether it is possibly a looping block; this spec is identified by the message-argument index.

For example, here is Boolean>>ifTrue: ifFalse: with its two EvaluationMetaSpecs:

- {required, noLoop, index 1}
- {required, noLoop, index 2}

The ifTrue: argument is the first argument of the message, and thus is designated by *index 1*; the ifFalse: being the second is designated by *index 2*. These EvaluationMetaSpecs produce the two traversal orders: {{1},{2}}; in other words, we must either evaluate the first message argument, the ifTrue: block; or else we must evaluate the second message-argument, the ifFalse> block.

Here is Collection>>detect:ifNone: with its two EvaluationMetaSpecs:

- {required, loop, index 1}
- {optional, loop, index 1}, {required, noLoop, index 2}

We either loop one or more times evaluating the detect: block or we possibly loop one or more times evaluating the detect: block, followed by definitely evaluating the ifNone: block once.

BRANCHING AT THE EVALUATOR

The evaluator always terminates the current traversal. The result for the evaluator is obtained by branching new traversals and combining the results; each child traversal order of the evaluator produces another branch.

Let's find a block's value type.

[2 even

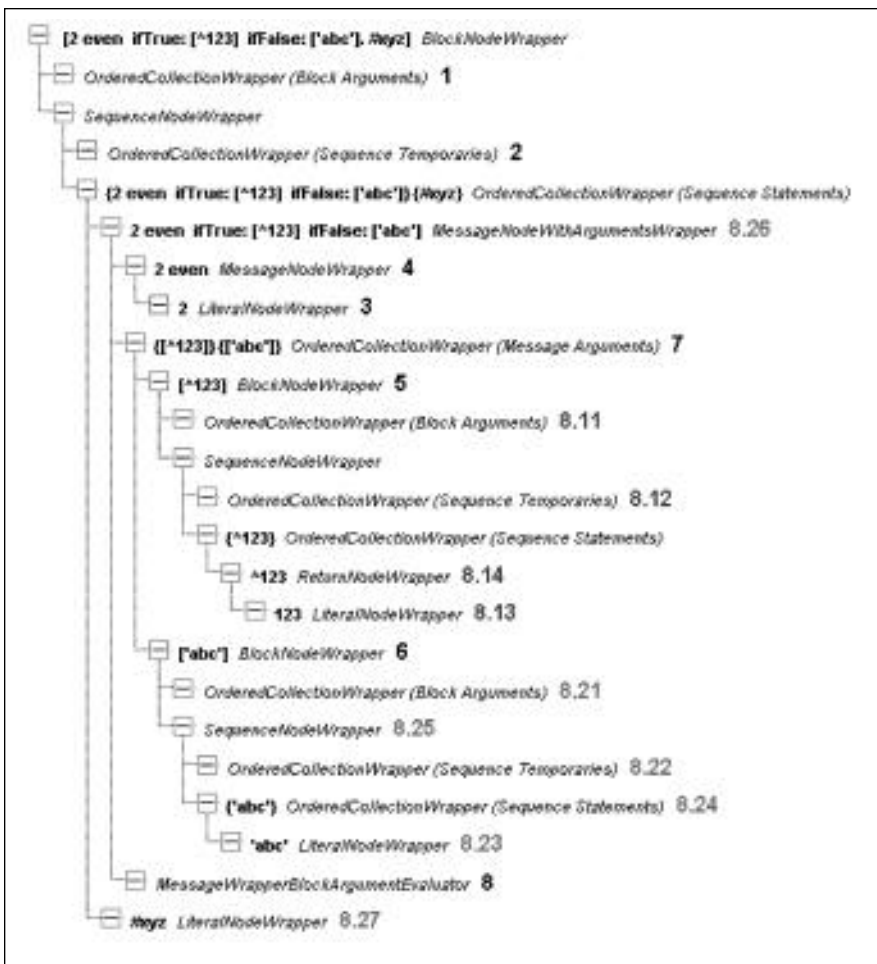


Figure 9. Branching.

```

ifTrue: [^123]
ifFalse: [^abc].
#xyz] "what is value-type for the block?"

```

We start at the first terminal node wrapper of the block and perform a forward traversal on the lookout for either a ReturnNodeWrapper or the very last statement of the block. We eventually arrive at the evaluator and proceed to find its result, which in this case will be our answer. Because the evaluator has two child traversal orders, it branches two traversals. We take the first traversal and find a ReturnNodeWrapper of type <SmallInteger>. The second takes us past the confines of the ifFalse: block. We visit a MessageNodeWithArgumentsWrapper, the final outer block statement and stop with a type of <ByteSymbol>. We combine the results to get our answer of <SmallInteger | ByteSymbol>. This traversal is depicted in Figure 9. The first branch occurs in visitations 8.1x shown in dark grey; the second, in 8.2x shown in light grey. Note that although 2 even is always true, the tool is unaware of this.

Looping. Let's look at type inferencing for a temporary variable. Let's find the type of t2 in the final statement of:

```
[: a1 | | t1 t2 t3 |
```

```

t1 := 123.
t2 := 'abc'.

a1 myCollection
detect: [: a2 |
    t3 := t1.
    t1 := $a.
    t2 := t3]
ifNone: [nil].
t2 "what is my type?" ]

```

If we don't loop at all, t2 type = <ByteString>; if we make one pass, t2 type = <SmallInteger>; if we make two passes, t2 type = <Character>. Thus, results differ depending on the number of times the detect: block is traversed.

The EvaluationMetaSpecs generate the appropriate child traversal orders, based on the number of descendent assignment statements (excluding those found in child block descendants). Each assignment generates another child traversal order. In the example, we find three assignments in the detect: block. The first EvaluationMetaSpec for #detect:ifNone: therefore generates {{1}, {1,1}, {1,1,1}}, and the second generates {{2}, {1,2}, {1,1,2}, {1,1,1,2}}. (We play it safe to loop the maximum amount, even though in this case only two passes are needed.) If we had no assignments, the resulting child traversal orders would simply be {{1}} from the first spec, and {{2}, {1,2}}

from the second.

This brute-force technique is of exponential order, but this is not a concern for minimal branching. The standard technique for type inferencing uses polynomial-order symbolic-execute via solution of equations.³

Type inferencing for a temporary variable involves traversing the wrapper tree backwards from the VariableNodeWrapper. Let's find the result for a double pass by walking the child traversal order {1,1} branch of the #detect:ifNone: evaluator. As shown in Figure 10, we proceed backwards on the lookout for AssignmentNodeWrappers with VariableNodeWrapper t2, and eventually reach the detect: block's third statement and stop at the AssignmentNodeWrapper. The result of the AssignmentNodeWrapper is the type of its righthand side, VariableNodeWrapper on t3. So we trigger a new traversal for inferencing the type of t3 (shown in dark grey), but the technique is to continue using the current child traversal order of the evaluator. This new traversal in turn triggers another for inferencing the type of t1 (shown in light grey), again keeping the child traversal order. We know to traverse the detect: block once more, and find type t1 to be <Character>.

TREATMENT OF BLOCKS

The contents of a block are treated as live, whether the block is evaluated or not. This is not an oversight, but a



Figure 10. Looping.

necessity. Remember that the rule block itself is dead if it is not being evaluated; surely we want this type checked, as this is the whole point. Likewise for any descendent block.

Also, remember that due to the morphing process, what is currently not a block can perhaps be morphed to a block's first statement, and vice versa (construct *Enclose statement in block* ($123 \rightarrow [123]$) and construct *Return block's first and only statement* ($[123] \rightarrow 123$)).

Traversal beyond the confines of a block depends on the situation. If the block is a statement, then there is no traversal beyond it. If the block is a message argument, then it depends on the block evaluator *EvaluationMetaSpec* acknowledging the block. Even if the block evaluator knows that a given block is never evaluated, the contents of the block are treated as fully viable (e.g., if the user selects inside the block); if it is not evaluated we simply don't traverse past the confines of the block. (An unknown type in this situation is defined as <'Nothing'> as discussed). Currently, all block-evaluator *EvaluationMetaSpecs* have live blocks.

CREATION OF THE WRAPPER TREE

Creation of the wrapper tree occurs on string input to the tool, as well as acceptance of text from the *FreeStyle* text view. The creation is achieved in iterative fashion, using forward traversal, during which we force our-

selves into blocks in order to traverse them.

The *MessageNodeWrapper* does the main job. It finds the appropriate selector information based on the type of its receiver. From this information, the required types of its arguments are instated, as well as the type of the message itself.

Validation is performed, ensuring that a wrapper conforms to syntax and the limitations of the tool. For example, a *ReturnNodeWrapper* ensures that it is last in a sequence of statements, and an *AssignmentNodeWrapper* ensures that its value child is not a *BlockNodeWrapper*. A faulty wrapper is highlighted in the *FreeStyle* text view along with the appropriate error.

CONCLUSION

Eagle needed a rule language, and Smalltalk itself was chosen. A point-and-click rule editor was developed that constrains the user to produce valid syntax with valid message selectors, via selection of valid *ProgramNode* tree manipulations. The user is responsible for valid message arguments by selecting from manipulations that satisfy type requirements. State and behavior wrapped around the *ProgramNode* tree make it type aware and traversable.

Can such a tool be generalized for editing Smalltalk in general? Probably not. Several issues would quickly get out of hand, such as specifying and maintaining selector information for all selectors. But for a rule language subset, the tool has a niche. ■

References

1. Graver, J. O. and Johnson, R. E., "A Type System for Smalltalk," In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 136-150, Jan. 1990.
2. Graver, J. O. *TYPE-CHECKING AND TYPE-INFERENC FOR OBJECT-ORIENTED PROGRAMMING LANGUAGES*, PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Aug. 1989. UIUCD-R-89-1539.
3. Palsberg, J. and Schwartzbach, M. I., *OBJECT-ORIENTED TYPE SYSTEMS*, John Wiley and Sons, New York, 1994.

Figures 9 and 10 inadvertently ran as Figures 6 and 7 in part 1 of this article in September. We apologize for any confusion.

Paul Davidowitz is a senior developer at Andersen Consulting. He can be reached at paul.davidowitz@ac.com.

JustCloningAround *subclass*: #CloningExtensions

Keith Piraino

All applications wrestle with the problem of copying objects at some point. In the course of some recent work, I used a variation of the cloning approach published in the *Journal of Object-Oriented Programming* (JOOP) two years ago.¹ This article will describe two extensions I added:

- Allow for customized behavior after the deep copy (#postDeepCopy:); and
- Provide ability to NOT copy certain instance variables.

JUSTCLONINGAROUND

The JOOP article described an implementation of a deep copy that can handle arbitrarily deep object structures in all three major Smalltalk dialects. Circularities are handled by keeping track of the objects copied in an IdentityDictionary. The keys are the original object and the values are the copies.

As the title suggests, I look at this article as a “subclass” of the JOOP article. Just as you wouldn’t expect to understand a class without browsing its superclass, you shouldn’t expect to understand this article without having read the JOOP article. I’ll present a complete implementation in code,* but I won’t cover the concepts from the original article.[†]

EXAMPLE

Consider the OMT² style diagram in Figure 1. Each customer contains a collection of invoices. Each invoice contains a collection of ordered items. These are aggregation or ownership³ relationships.

Each invoice holds a reference to the customer as well as to the salesperson that created the invoice. Each ordered item holds a reference to the product it is order-

ing. These are not aggregation relationships. For example, an invoice does not own the salesperson.⁴

Consider the instance diagram in the top half of Figure 2.[‡] Jane Profit might use this existing invoice as the basis for creating a new one. The bottom half of Figure 2 shows the resulting instance diagram, if the JOOP approach had been used to copy the invoice. There are two problems with this result. The first is that everything is copied when what we want is to copy only the objects referenced by aggregation relationships. For example, what we don’t want is a new instance of Jane Profit.

The second problem is that when we copy an invoice we want to reset the quote to the customer. The quote is what the customer actually pays, and it might be higher or lower than the actual cost.

Determining the quote is up to the salesperson. In this particular case the customer has been given a \$25 discount on an order for three “blue widgets;” however, that doesn’t mean he or she will always get that discount. In this application we’ve established the rule that all quotes should be set to zero when copies are made.

After copying, what we actually want is shown in Figure 3.

[‡] I’m not showing the Customer to Invoice relationship for now.

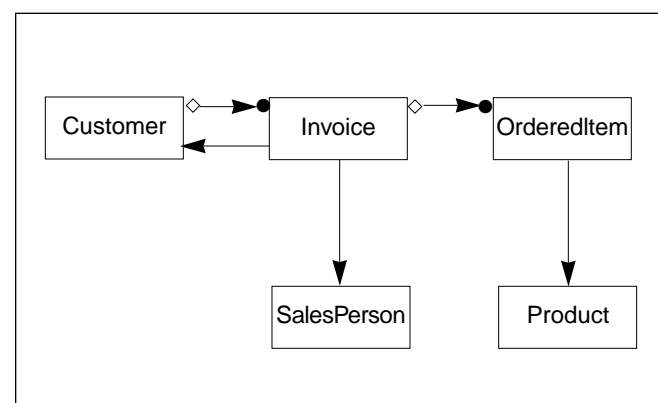


Figure 1. Example Object Model.

* I used VisualWorks, but this should work in other dialects if you keep in mind the issues raised in the JOOP article.

[†] The JOOP article also describes how to implement a deep equal, which I don’t cover at all.

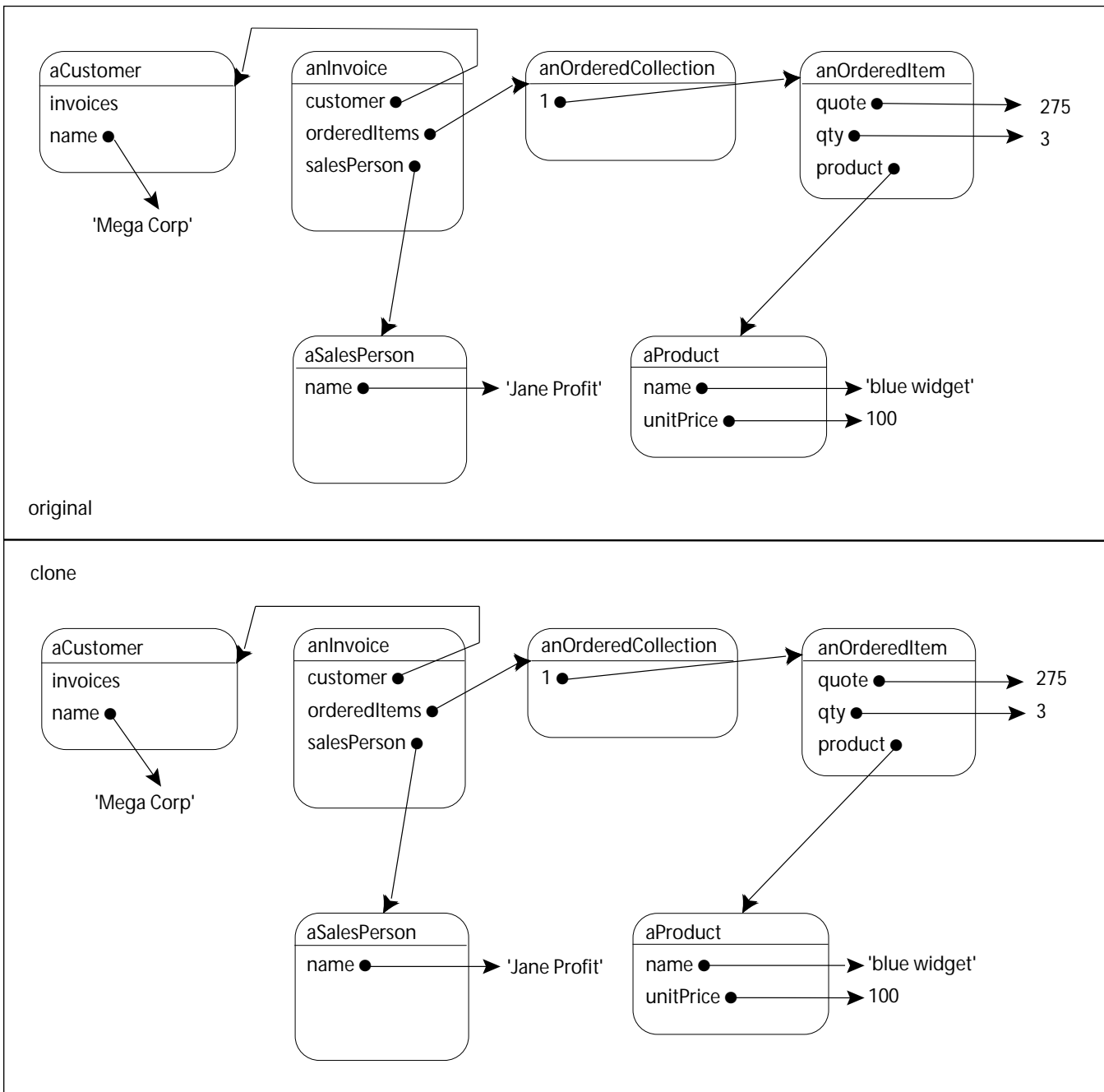


Figure 2. Cloning an invoice using the JOOP approach.

#POSTDEEPCOPY:

Of our two problems, resetting the quote is the easier one so let's tackle that first. Our generic problem is being able to have customized behavior after an instance is copied. Using the normal VisualWorks mechanisms, this would be done via #postCopy. The equivalent to this in our generic mechanism is a method called #postDeepCopy: that is sent to the new instance after the deep copy is done. Listing 1 shows the implementation of the generic copy mechanism. It's been factored a little differently, but other than #postDeepCopy: this is the same code as published in the JOOP article.

Below would be the implementation of this method for OrderedItem:

```
postDeepCopy: anObject
super postDeepCopy: anObject.
self quote: 0.
```

The anObject parameter is the original object. I've had no cause to use it yet, but I thought this parameter might come in handy in certain cases. For instance, we could use #postDeepCopy: as a way to solve the problem with copying objects that aren't owned. After the copy is made, we simply set the appropriate instance variable to refer back to the original object. Below is an example for Invoice:

```
postDeepCopy: anObject
super postDeepCopy: anObject.
```

```
self salesPerson: anObject salesPerson
```

One problem with this approach is that there could be side effects to copying a salesperson object, which we want to avoid. Let's see if we can come up with a better approach.

#DONTCOPYVARS

For any given object we want only to copy some of its instance variables. We have a choice, though. We can create a mechanism that forces us to specify what should be copied, or that requires us to specify what should NOT be

Ideally, the vendors should standardize on some way of specifying "ownership" and make use of it in their copy mechanisms.

copied. I've taken the latter approach because, in problem domains I've been exposed to, it appears that more things get copied than not.

This mechanism will also assume that we can specify this information on a class basis. In other words, no instance of Invoice will ever need to create a new instance of SalesPerson when it is copied.

Every class can optionally define a class method called #dontCopyVars that answers a collection of the named instance variables that should not be copied. Below are examples from our object model in Figure 1:

```
Invoice class>>dontCopyVars
^#(#customer #salesPerson)
```

```
OrderedItem class>>dontCopyVars
^#(#product)
```

Customer, SalesPerson, and Product would not have to define a #dontCopyVars method.

#ALLDONTCOPYVARS

In our example we don't have to worry about inheritance, but most structures aren't this simple. We need a class method that will traverse the superclass chain and collect the names of all instance variables to not copy. The code below is patterned after #allInstVarNames[§] and #accumulateInstVarNames but, takes into account that not every class will implement #dontCopyVars:

```
Object class>>allDontCopyVars
| vars |
vars := OrderedCollection new.
self accumulateDontCopyVars: vars.
^vars
```

```
Object class>>accumulateDontCopyVars: aCollection
```

```
self superclass notNil
  ifTrue: [self superclass accumulateDontCopyVars:
aCollection].
(self class includesSelector: #dontCopyVars)
  ifTrue: [aCollection addAll: self dontCopyVars].
```

These two methods ensure that every class can answer a collection of the instance variables that should not be copied. All that's left is a little bit of code to utilize this information during copying. See Listing 2.

MISCELLANEOUS

Every object should answer a copy of itself when sent the message #copy. Users of the object should not have to worry about whether it uses the deep copy mechanism. To account for this, I usually redefine #copy for all my domain objects as follows:

```
copy
^self deepCopy
```

When we copy an invoice we need to know whether we are making a copy for use with the customer that owns the original invoice, or for a different customer. To handle this I defined a #copyFor: method in Invoice that takes the Customer as a parameter:

```
copyFor: aCustomer
| aCopy |
aCopy := self copy.
aCustomer invoices add: aCopy.
aCopy customer: aCustomer.
^aCopy.
```

Avoid the temptation to specify #dontCopyVars based on the current functionality of your application. As an example, assume that OrderedItem instances have a reference to their containing Invoice. Also assume that the application currently allows Invoices to be copied, but not OrderedItems. You might be tempted to not include #invoice in #dontCopyVars for OrderedItem. Because of the support for circularities in the copy mechanism, if Invoice is copied before OrderedItem the OrderedItem will end up pointing to the correct Invoice anyway.

However, at some point these kinds of assumptions will come back to haunt you. Every object should be able to answer a reasonable copy of itself.

CONCLUSION

Unfortunately, Smalltalk does not provide a way to distinguish between aggregation relationships and simple references. What I've provided is one way of doing this. Ideally, the vendors should standardize on some way of specifying "ownership" and make use of it in their copy mechanisms. This kind of standard would, for example, allow CASE vendors to output this information during code generation.

LISTING 1 instance methods in Object

```
deepCopy
```

[§] You can find a lot of good solutions if you try to think of how your problems are similar to something already in the image.

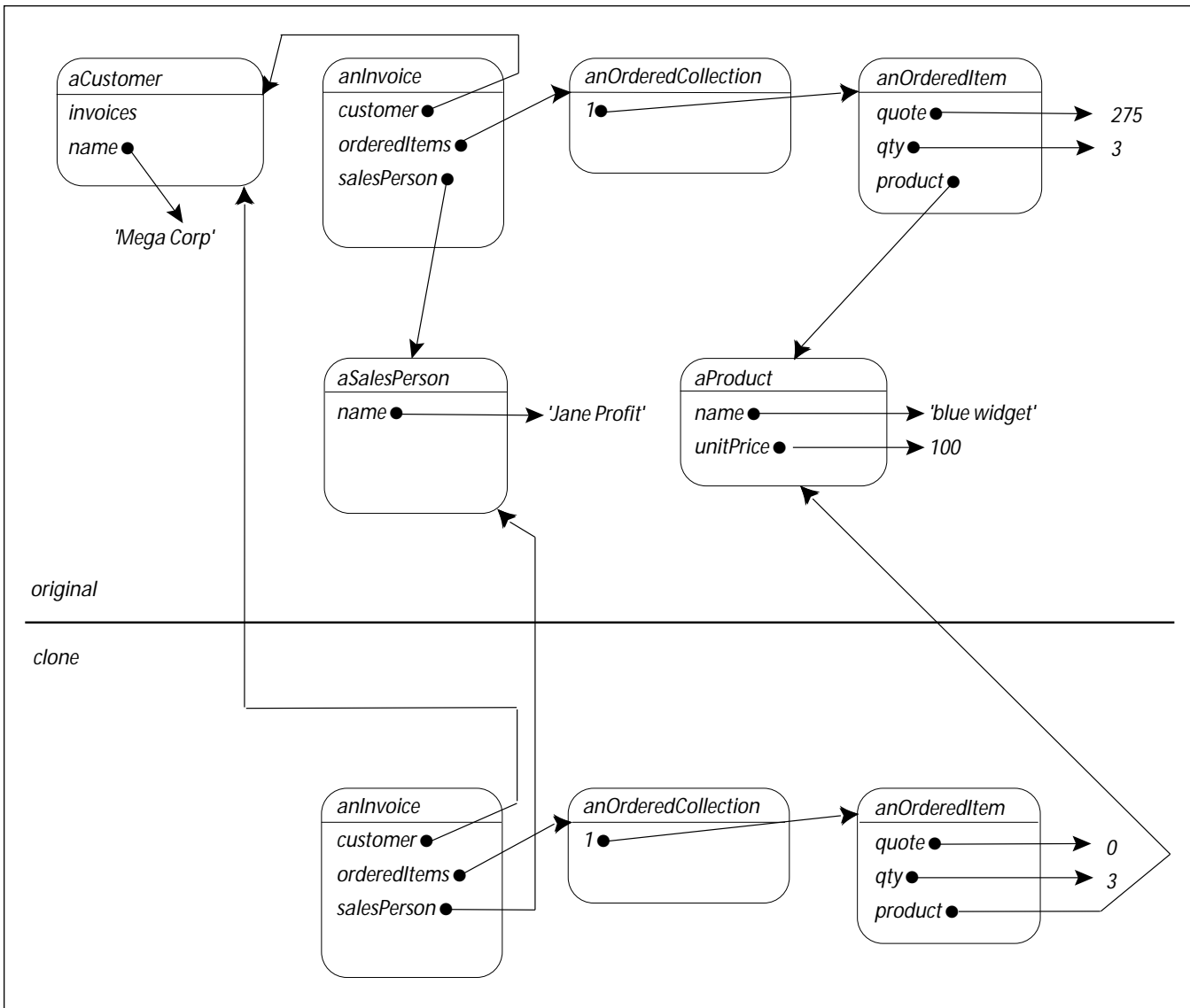


Figure 3. Cloning an invoice using both extensions.

```

^self deepCopyWithoutRecopying: IdentityDictionary new.
^aCopy.

deepCopyWithoutRecopying: anIdentityDictionary
^anIdentityDictionary
at: self
ifAbsent: [self doDeepCopyWithoutRecopying:
anIdentityDictionary]

doDeepCopyWithoutRecopying: anIdentityDictionary
| aCopy |
aCopy := self shallowCopy.
aCopy == self ifTrue: [^self].
anIdentityDictionary at: self put: aCopy.
self class isPointers ifFalse: [^aCopy].
aCopy releaseCopyDependents.
aCopy copyNamedVarsWithoutRecopying:
anIdentityDictionary.
aCopy copyUnnamedVarsWithoutRecopying:
anIdentityDictionary.
aCopy postDeepCopy: self.

releaseCopyDependents
^self breakDependents

copyNamedVarsWithoutRecopying: anIdentityDictionary
| newPart |
1 to: self class instSize do:
[:idx |
newPart := (self instVarAt: idx)
deepCopyWithoutRecopying: anIdentityDictionary.
self instVarAt: idx put: newPart].

copyUnnamedVarsWithoutRecopying:
anIdentityDictionary
| newPart |
1 to: self basicSize do:
[:idx |
newPart := (self basicAt: idx)
deepCopyWithoutRecopying:

```

```
anIdentityDictionary.  
    self basicAt: idx put: newPart].
```


```
postDeepCopy: anObject  
    "The receiver is a deeply copied instance of anObject.  
    Subclasses may override this method to provide  
    behavior after copy is done"
```

LISTING 2

instance methods in Object

```
copyNamedVarsWithoutRecopying: anIdentityDictionary  
    | newPart |  
    self class allVarIndicesToCopy do:  
        [:idx |  
            newPart := (self instVarAt: idx)  
                deepCopyWithoutRecopying:  
anIdentityDictionary.  
    self instVarAt: idx put: newPart].
```

class methods in Object

```
allVarIndicesToCopy  
    "Answer a collection of the indices of all named  
    variables to copy for the receiver"  
    ^self allVarNamesToCopy collect:  
        [:each | self allInstVarNames indexOf: each]  
  
allVarNamesToCopy  
    "Answer a collection of all variable names to copy for  
    the receiver"  
    ^self allInstVarNames reject:  
        [:each | self allDontCopyVars includes: (each asSymbol)]  
  
dontCopyVars  
    "Answer a collection of instance variables defined in  
    this class that should not be copied when a deep copy is  
    made of an instance of the receiver or one of its  
    subclasses. The collections should contain symbols, not  
    strings.  
  
    Only classes that define instance variables that shouldn't  
    be copied need to define this method"  
    ^#() 
```

REFERENCES

1. Lalonde, W. and Pugh, J., "Just Cloning Around," *JOOP* 7(5); 1994.
2. Rumbaugh, J. et al., *OBJECT-ORIENTED MODELING AND DESIGN*, Prentice Hall, Englewood Cliffs, NJ, 1991.
3. Weir, C., "Improve Your Sense of Ownership: Exploring a Design Principle," *ROAD*, 2(6); 1996.
4. Check out www.sigs.com/publications/docs/oc/9608/oc9608.d.dialog.html for an interesting discussion of aggregation vs. association.

Keith Piraino is a consultant who can be reached at
keith.piraino@bug.com.



Wayne Beaton



Dwight Deugo

Visual Programming and Reusable Parts: The Marquee Part

Constructing reusable parts is a difficult task. You have to know what tools are available to use, and you must be familiar with existing reusable parts. Most importantly, you have to know the process to construct them. The task is not impossible, and with experience it becomes easier.

In this column, we will help you gain experience creating reusable parts by discussing the creation of a marquee part. A marquee is a relatively simple part designed to reflect the operation of a light board appearing in the front of many sport facilities. The marquee scrolls text that is too large to display statically. Our implementation scrolls right-to-left by repeatedly removing the first character of its message, and tacking it on to the end.

CONSTRUCTING A MARQUEE

It is simple to construct a window containing a Scrolling label. We constructed one called a `MarqueeDemonstrationWindow`, using both a label part and a timer part. Two connections between the window and the timer ensured that the timer started when the window opened and stopped when the window closed. Another connection updated the contents of the label when the timer ticked. Figure 1 shows the appearance of the `MarqueeDemonstrationWindow` in the Composition Editor.

IBM's VisualAge for Smalltalk Version 3.0 provides a timer part in the Multimedia Package. Version 2.0 includes the same part with an exercise. The timer has several features that can be set using the Settings Editor, shown in Figure 2. These features include the timer's period in milliseconds, and whether or not the timer should fire repeatedly. In our example, the timer is set to repeatedly fire every 100 milliseconds.

To support the scrolling mechanism in the `MarqueeDemonstrationWindow`, we made the following connections in the Composition Editor. First, we connected the

window part's `openedWidget` event to the timer's start action, and then connected the window's `closedWidget` event to the timer's stop action. With these connections, the timer fires one hundred milliseconds after the window opens, and continues to fire every one hundred milliseconds until the window closes. Finally, we connected the timer's `timerFired` event to the following script titled `scroll`:

```
scroll
| oldText newText |
oldText := (self subpartNamed: 'Label1') labelString.
newText := (oldText copyFrom: 2 to: oldText size),
  (String with: oldText first).
(self subpartNamed: 'Label1') labelString: newText
```

The script retrieves the label's current string, constructs a new one containing the current string's first character removed and appended to the end of it, and then updates the label with the new string. As this script executes ten times a second, the label's string cycles continuously.

We made further embellishments to permit the user to dynamically change the label's string. We added a push button and an entry field to our window so that the user could click the push button to set the label's string to the entry field's current string. To support this functionality, we connected the push button's `clicked` event to the label's `labelString` attribute and connected the `clicked/labelString` connection's value attribute to the entry field's object attribute.

WIDGET LAYOUT

Part of constructing a complete application includes specifying how parts react when a window is resized. Figure 3 shows the `MarqueeDemonstrationWindow` in action. When the window is resized, the entry field clings to the top of the window and the scrolling label clings to the bottom. The left and right sides of each part remain constant distances from the sides of the window.

Specifying a part's layout attachments in VisualAge is simple. In the a part's Settings Editor there is a page labeled `Layout`. On this page, you can specify layout information for the part's top, bottom, left, and right sides. For the input field part, we specified that its top

Dwight Deugo and Wayne Beaton are senior members of the development and educational staff at The Object People, in Ottawa, Ontario. Dwight (dwight@objectpeople.on.ca) has immersed himself in objects for more than 10 years and has helped clients with their object immersions as a project mentor and as a course instructor. Wayne (wayne@objectpeople.on.ca) is the coordinator of course construction and a software developer.

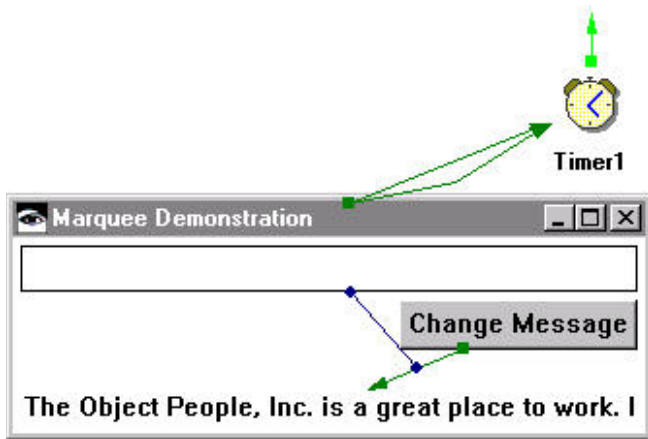


Figure 1. The MarqueeDemonstrationWindow as it appears in the Composition Editor.

edge is attached to the “Parent top edge,” and that there is a four-pixel separation between them (the top edge of the input field part is a constant four pixels away from the top edge of its parent). The parent of any part is the canvas (typically a window) where the part is situated. If the part is positioned on a form, then the form is its parent. Similar attachments were specified for the input field’s left and right edges (connecting them to the parent left and right edges, respectively).

The input field part’s bottom edge used a different attachment. Specifying “No” attachment for any part indicates to VisualAge that the part should use whatever amount of space is appropriate. In the case of an input field part, the appropriate amount of space depends on its current font—the bottom edge will be far enough away from the top edge to accommodate the font, plus an appropriate amount of gutter space. Be aware that the “No attachment” attachment type makes sense only if the opposite edge is attached. For example, it is not possible to have both the top and the bottom unattached.

The “Target attachment” type is also very powerful. It permits you to constrain one part’s edge to another part’s edge. In Figure 3, the push button’s top edge is attached to the input field’s bottom edge. By using all of these arrangements, it is possible to design a window in which the parts always resize correctly, because they are dependent on the positioning of other parts, the current font, and the monitor resolution—fonts sometimes have different metrics in different monitor resolutions.

MAKING A REUSABLE PART

TheMarqueeDemonstrationWindow demon-

Our implementation scrolls right-to-left by repeatedly removing the first character of its message, and tacking it on to the end.

strates one possibility in VisualAge: a scrolling marquee. However, anyone wanting the same marquee feature must drop the appropriated parts and make the correct connections. Since the marquee feature has potential for reuse, we built a reusable Marquee part for others to use, in which its construction was transparent to them.

The construction of the Marquee part began by creating a new visual part named Marquee. In the Composition Editor, we removed the default window and copied to it the label and timer parts from the MarqueeDemonstrationWindow, as shown in Figure 4. The script was also copied to the new part.

Next, we restored the timer’s timerFired: event connection to the script named scroll. However, we had to change the way the timer started and stopped—there was no longer a window to trigger openedWidget and closedWidget events. These events still exist because all visual parts give notice, via the triggering of events, that they have opened. The Marquee part is no different. The

context menu on the free-form surface area (the white space in the Composition Editor) contains the connect menu for the Marquee part. We connected the openedWidget event from the Marquee part to the timer’s start action, and connected the closedWidget event to the timer’s stop action.

This new Marquee part was now ready to use in the

MarqueeDemonstrationWindow. First, we removed the old label and the timer parts from the window. Since the new Marquee part was not accessible in the palette, we added it using the Composition Editor’s Options menu entry “Add” part. We placed the Marquee part in the window, opened the window, and it worked!

PRIMARY PARTS

Visual parts do not necessarily contain a window. However, all parts created with the Composition Editor have a Primary Part. The primary part is the first thing the

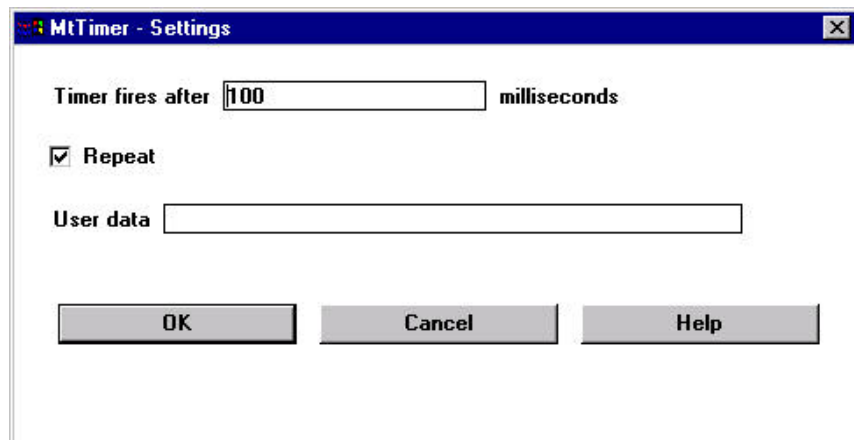


Figure 2. The Settings Editor for a Timer part.

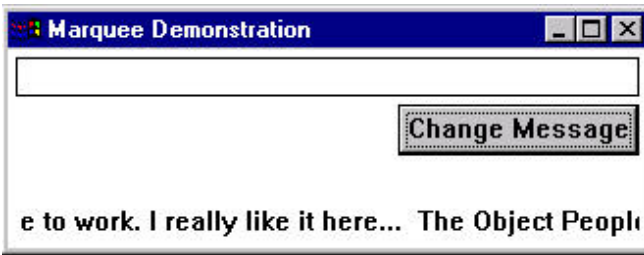


Figure 3. The MarqueeDemonstrationWindow in action.

user sees, which is usually a window. Sometimes the primary part can be another visual part. In our Marquee part, the primary part is the label. There is no special reason to have a window part as default primary part, short of windows being useful.

In the Composition Editor all visual parts that are not the primary part have an entry “Become” primary part in their context menu. We used this menu item to change the label to the new primary part.

In the original MarqueeDemonstrationWindow, the user could change the contents of the marquee. That ability is still available in our new Marquee part. Every part created using the Composition Editor has the same public features as those of its primary part. Therefore, the Marquee part has the same features as the label, including the labelString attribute. In fact, if you open the settings for the Marquee part, you will see entries for most of the label’s attributes, including labelString.

As with the previous version of the MarqueeDemonstrationWindow, we connected the push button’s clicked event to the Marquee part’s labelString attribute and connected the value attribute of the resulting connection to the object attribute of the text field.

PROMOTE PART FEATURE

The connection between the push button and the Marquee part’s labelString attribute identifies several problems with the Marquee part. First, should a Marquee part actually have a labelString attribute? We tend to think of a marquee as having a message that is displayed. Second, the name labelString implies something about the internals of the Marquee part that should be irrelevant, especially if those internals may change at some point in the future.

Two improvements are required. We need a new name for the labelString attribute (perhaps message). Also, the labelString attribute, along with the other inappropriate features, should be removed from the Marquee’s public interface. The first problem is easy to correct, the second is not. We will now address the first problem and defer the second problem for a future discussion.

VisualAge can promote a part’s features to features of its parent part. In our case, we wanted the label’s labelString attribute promoted as a message attribute of the

Marquee part. The benefit is that users of the Marquee part no longer know about the Marquee’s internal representation, and, if at a later time we decide to change the implementation, we only have to ensure that the message feature persists.

In the current version of VisualAge, features of the **primary part** cannot be promoted. We assume that the designers built this into the product because all those features are made available anyway. Apparently, the designers did not have our reasoning. One workaround was to make another part become the primary part, promote the required feature of the label, and then change the primary part back to the label.

COMPOSITION EDITOR VARIABLES

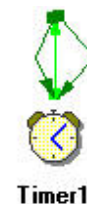
Another workaround requires the use of variables. First, create a variable and connect its self attribute to the label’s labelString attribute with an attribute-to-attribute connection. With this connection in place, the contents of the labelString attribute will always be reflected in the

variable, and vice-versa. Rather than “fudging” the feature promotion as before, you can now directly promote the variable’s self as the feature message. The self attribute of a variable actually refers to the contents of the variable, in this case a string. With the connection between the variable and the label, promoting the variable’s self attribute has

the same effect as promoting the label’s labelString attribute.

VisualAge makes these tasks easy. The pop-up menu for the label part contains an entry “Tear-Off Attribute.” When this entry is selected for a part such as a label, VisualAge prompts you to select an attribute from the label’s attribute list. Based on your selection, VisualAge creates an appropriate variable and an attribute-to-

The notion of a primary part permits one to have more than one window in a single Composition Editor.



The Object People, Inc. is a great place to work.

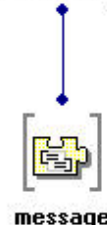


Figure 4. The Marquee part in the Composition Editor.

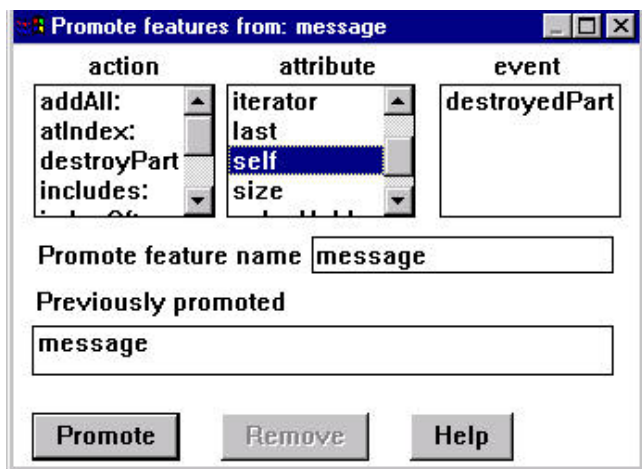


Figure 5. The Promote Part Features window.

attribute connection between the label's selected attribute and the variable's self attribute.

To promote the variable's self attribute as the Marquee's message attribute, position the mouse over the newly created variable in the Marquee part, pop up the menu and then select "Promote Part Features...." In the promote window, shown in Figure 5, select the self attribute and provide message as the feature name. Finally, click the Promote button to add the feature to the public interface of the Marquee part.

After saving the Marquee part, its feature list will include the promoted feature message. Further, the Marquee part's

Settings Editor automatically includes the promoted message feature for you to initialize.

SUMMARY

We've discussed the construction of one reusable part: the Marquee part. Although its use is limited, what is interesting about it is the process and the VisualAge's features used for its construction. The notion of a primary part permits one to have more than one window in a single Composition Editor. It also makes it possible to have a primary part that is not a window (perhaps a push button that opens a window might find some application as a reusable part). Promoting part features is a wonderful mechanism for selectively granting access to the internals of objects. The Composition Editor's variables are useful for passing values from one part to another. They also permit access to parts of parts.

In future columns we intend to discuss more about each of these mechanisms, and many others, in the contexts of other interesting reusable parts we have created.

THE CODE

The code presented in this column and in future columns is available at <http://www.objectpeople.com>. The code is presented as an IBM Smalltalk library file, containing two versions of the software. Version one provides the single-class implementation of the Marquee; version two presents the reusable form with an example. **S**



Alan Knight

Two New Smalltalks

Smalltalk is a very loose term. While there are some clear, defining characteristics, there are also many variations as well. This is important because software development is a large field, with different solutions appropriate to different problems. This article will discuss some of these possible variations, with reference to two promising new implementations.

The first of these new implementations is Object Connect's "Smalltalk MT," a high-performance implementation for Windows NT 4.0, with full compilation and true multithreading. The second is Intuitive Systems' "Dolphin Smalltalk," which boasts a low cost and a very low memory footprint.

Neither one has been officially released yet, but free prerelease copies are available for download from their Web sites (see contact information at the end). Even if you've missed the free periods, both companies are aiming at product prices in the low hundreds (of U.S.) dollars.

Note that I looked at prerelease versions, and that I've only played with them, so all I can do is relay first impressions and generalities about their implementation choices. I hope you'll see a full review in these pages once they are officially released.

COMPILATION

Most current Smalltalks use dynamic compilation, now commonly referred to as Just-In-Time (JIT) compilation. This is an example of a process I call "cross-domain buzzword hybridization." In this process, a recognized buzzword from one domain crosses over to another, where it attaches to a concept almost, but not quite, entirely unlike the original. If you don't like it, think of it as payback for the term "object-oriented."

In dynamic compilation, the methods are stored in bytecode form. At runtime, some (or all) of these bytecodes are translated into machine code. This can provide most of the speed of machine code with a much smaller executable size, comparable to that of bytecodes.

Although this is a good approach, it's not a perfect solution for every circumstance. It takes more space than bytecodes (you now have to store both compiled forms) and

still won't run as quickly as compiled code (you have to spend CPU cycles to translate, and you can't afford to think too hard about optimization). Finally, it requires a VM to do the translating, which makes it more difficult to create standalone executables, DLLs, and callback functions.

Smalltalk MT is fully compiled, and has "an optimizing compiler that generates fast, compact code." Full compilation should give significantly better performance than existing implementations, and the absence of a VM allows simple programs to be very small indeed. MT claims to be able to make DLLs (which, to my knowledge, no existing Smalltalk can do), and to fit a trivial windows application into a 100K executable (their runtime-support DLL adds only 52K).

Dolphin Smalltalk is bytecode interpreted. I would expect this to offer significantly slower performance than current implementations, but with much lower space consumption, even for significant-sized applications. The company states that it "will be producing an optional JIT in future." When space is more important than speed, this is a very reasonable option, and this is often true in applets. The firm also says it will shortly have its VM encapsulated as an ActiveX downloadable.

SIZE

Both companies advertise their ability to make very small executables. Traditionally this has been a weak area for Smalltalk. In many of the application areas for which Smalltalk is used, one can argue that size doesn't matter as long as it does the job. But with the growing importance of small applets over monolithic applications, size is becoming more significant.

Dolphin's approach to minimizing size uses bytecoding and a very small interpreter (I counted 155K including all the DLLs) with a relatively standard (but small) image. Their prerelease packaging support wasn't very sophisticated, but when a development image starts at only 1.4 MB a good stripper is more of a luxury.

Smalltalk MT can provide extremely small sizes for simple programs, but for larger programs the space cost of full compilation becomes significant. The firm tries to combat this through modular class libraries and a minimalist framework.

The typical Smalltalk programmer doesn't think too

Alan Knight is a Smalltalk guru with The Object People. He can be reached by email as knight@acm.org, or at 613. 225. 8812.

much before using something that's in the development image. This is nice for development but bad for packaging. Everything depends on everything else, and all you can do is remove the compiler and development tools. MT claims to have kept the base classes small and modular, so if you don't use a particular class or subsystem it can easily be removed. They also claim to have a sophisticated packager, though I'm skeptical. Packaging is a difficult problem. I figure their base image, including the compiler and development DLLs, is 1.3 MB. "Image generation is largely automatic. An Image Builder computes the set of referenced classes and methods, starting with the image-entry point."

MT has also minimized the size of the runtime by sticking very close to the OS functionality.

"All GUI and operating system-related classes present a Win32-like protocol. This ensures that applications run with minimum overhead on the Windows family of operating systems, and leverage on existing know-how."

FRAMEWORKS

A Win32-like protocol for windowing is definitely a two-edged feature. It makes Windows programmers who have programmed in other languages feel right at home, and it minimizes the amount of code you need to map a Smalltalk framework onto the operating system. I think there's no question that this reduces the amount of code in the image.

On the other hand, frameworks that are written with C or C++ in mind look *really* ugly compared to those designed for Smalltalk. Look at Visual Smalltalk vs. IBM. Visual Smalltalk has tight integration with Windows, to the point where you can implement your own `wm:what-ever: messages`, but provides a very clean, simple, Smalltalk-like framework above that. IBM has its X/Motif layer, which translates into calls to the real OS. It works very well, and it's extremely portable, but it's way too close to C programming for my taste. IBM's saving grace is that with VisualAge or WindowBuilder you rarely have to descend to that level. Smalltalk MT doesn't appear to have a GUI builder. On the other hand, I'm sure there are lots of people using C/C++ for whom a Win32-like framework in Smalltalk (especially a Smalltalk that can produce fast, compact executables) would be a big step up.

Dolphin Smalltalk, in contrast, uses MVC. This is the oldest and best known of the Smalltalk frameworks, and has been passed down from PARC into VisualWorks. Dolphin has adapted it to a native-widget, event-driven form, but it still has Views, Controllers, and even ValueModels. They've also included a mediator class (called Tool) analogous to the Application Model of VisualWorks. I haven't tried building a window with either dialect but I'd expect to feel more comfortable with Dolphin.

THREADS

One of the most innovative features of Smalltalk MT is its full support for operating system threads ("true" multi-threading). Making a decent compiler isn't that hard, but there are many issues involved in making Smalltalk use OS threads. The only other implementation I know of is OTT's embedded systems implementation, and it only supports threads for real-time operating systems.

I believe OS threads are often given too much importance. For 90% of applications, the standard Smalltalk process model will suffice, along with the ability to make system calls without blocking. The critical issue is that when one Smalltalk process waits on a system service, the entire Smalltalk system should not wait. Given this capability, which most implementations provide, it's not difficult to render demanding applications like Web or terminal servers, and you don't have to deal with the additional complexities of OS threads. With the Smalltalk model

you can even have thread-manipulating code that's portable between operating systems.

That's a very good solution when it works, but there are times when you really need OS threads. One example is symmetric multiprocessing (SMP).

Operating systems that support multiple CPUs in one machine can map OS

threads to different processors, but not Smalltalk processes. Presently, this affects only very high-end machines, but it will become more and more important in the future.

Given that there are lots of implementations out there with the standard threading model, the choice of using OS threads in Smalltalk MT is a very welcome one indeed.

PLATFORMS AND PRICE

By now I'll bet you are impatient to get your own copy of either or both of these implementations for your favorite platform. Unless you're a Microsoft fan you're out of luck. Both of these implementations have opted away from portability in favor of close integration with Win32. In fact, the first version of Smalltalk MT runs only on Windows NT 4.0 due to problems with Win95's threads, but they expect to have a Windows 95 version soon.

Dolphin Smalltalk has gotten more attention on the Net, because it can also run on older versions of NT and Windows 95. Don't expect them to expand their list of platforms too much. In response to a number of calls for an OS/2 version, Andy Bower of Dolphin's support group (Dolphin.support@intuitive.co.uk) wrote:

"The initial design aims for Dolphin were firmly directed to producing a great Win32 development system and this assumption is built into the product at a low level....we'd be reluctant to compromise this for compatibility with other environments."

As for pricing, both are very low compared to the current standards. Neither has fixed a firm price yet, but from what I've heard MT is approximately U.S. \$300, and Dolphin at under U.S. \$200.

continued on page 26

Smalltalk is a very loose term. While there are some clear, defining characteristics, there are also many variations as well.

never really “get OO.” After a reasonable time period (perhaps as long as nine months), the people who still haven’t gotten it need to be given alternatives. Neither the mentor nor the developer is to blame. Not all people are able to think abstractly, and they need to be given the chance to contribute to the organization in a job for which they are suited.

CONCLUSION

“Smalltalk guru” is not the equivalent to Smalltalk mentor. Not all team members will accept mentoring, and not all team members will get OO. Do the best you can as a mentor and as a developer, and try to keep egos out of the equation. If personality clashes are a problem, maybe the mentor has to go. This is a tough call that the manager will have to make. Good luck and happy mentoring! **S**

References

1. THE WORDSWORTH CONCISE ENGLISH DICTIONARY, Hertfordshire. Wordsworth Editions Ltd., 1994.
2. Steinman, Jan, and Yates, Barbara, “Secrets to Building Successful Smalltalk Teams,” tutorial at Smalltalk Solutions, March 1996, New York, N.Y.
3. Steinman, Jan, and Yates, Barbara, “Special” Team Members,” *The Smalltalk Report*, V5N6, February 1996, pp. 15-17, 28.

continued from page 23

OVERALL

Overall I’m quite impressed with both of these implementations. As prerelease products, they’re obviously immature in some areas. For example, I had difficulty with the debugger in both of them. The GUI builder in Dolphin didn’t work yet, and MT doesn’t appear to have one. On the other hand, in a lot of areas, they’re surprisingly mature. They already have advanced features like finalization and exception handling in place. Inevitably, it will be a while before they’re fully loaded with those features that have nothing to do with a language, and everything to do with a successful project: industrial-strength source code control, native database connections, extra widgetry, report writers, business graphics, and so forth. Nevertheless, they show enormous potential, and are well worth your while to investigate.

Of the two, I expect MT to be the first choice for those looking for cool new features, and for those doing things that are traditionally difficult in Smalltalk (e.g. server-based Smalltalk, very tiny apps). Restriction to the newest version of NT will lessen their impact in the short term. Dolphin Smalltalk, with a very low price, Win95 support, MVC, and ActiveX applet support, has real potential to become the Smalltalk for the masses.

Both are filling niches that are under-represented by current implementations, and I hope they will enjoy great success. **S**



Jan Steinman



Barbara Yates

Mentoring

Does anyone out there think the word mentor is overused and abused in our industry? We are often dismayed at the ads we see from companies trying to recruit Smalltalk “mentors.” Judging by some of these ads, it appears that anyone with a year or more of Smalltalk experience is deemed capable of mentoring. We want to tell you that it just ain’t so!

WHAT A MENTOR DOES

A dictionary defines a mentor as “a wise counsellor (one who advises and warns).”¹ A good mentor gets great personal satisfaction from helping others to learn and grow. They can be compared to good teachers; they are not Smalltalk gurus who take over your keyboard and write your code for you. Mentors need patience and excellent communication skills, and must know the technical content of the subject they are mentoring. Mentoring is a one-on-one activity, so it is important that there be a “personality click” for the mentoring to be successful.

When managers want to recruit one or more mentors for a team, it’s important that they check references. When talking to a manager of a team the mentor has worked with, it would be useful to ask to speak with some of the team members about their experiences. Given the time crunch most managers are in, there might be little time for checking references. At a minimum, here are some questions to ask the candidate-mentor :

- How would you handle a team member who doesn’t want to ask for your help?
- How would you deal with a team member who asks very low-level questions most of the time?
- What aspect of mentoring do you like most?
- What percentage of your time would you expect to spend programming in Smalltalk vs. mentoring?

Jan Steinman and Barbara Yates are cofounders of Bytesmiths, a technical services company that has been helping companies adopt Smalltalk since 1987. Between them, they have over 22 years of Smalltalk experience. They can be reached at Barbara@Bytesmiths.com, Jan@Bytesmiths.com, or at their website at <http://www.bytesmiths.com>.

- What is the hardest problem you’ve had to deal with as a mentor and how did you solve it?

Mentoring is a lot more difficult (and sometimes a lot less fun, to be honest) than programming in Smalltalk. Every mentor can probably tell a few horror stories about his or her experiences. We’ll briefly share a couple with you.

We were called upon to mentor a small team of novice Smalltalkers. We were not supposed to mentor full time, having major architecture and development responsibilities as well. We found that the proportion of mentoring to

other tasks would vary in different stages of the project, so that when people were deeply into coding, our mentoring would only be about 30% of our work load. (More mentoring occurs at the early stages, and then it gradually tapers off.)

The project had one team member who was a very smooth manipulator. This person would ask for help with a given problem and within a

few hours we would find ourselves sitting at the keyboard writing their code! This keyboard switchover took place without our intention. Regardless of the mentoree’s hypnotic abilities, it was our responsibility to help the team member to learn, not to do the coding for him.

This is where patience comes in. Of course the mentor can do it faster (maybe 10 or 20 times faster!), but that’s not why the project needs a mentor. If the project required just cranking out code, management wouldn’t have given you a mentoring role.

The second horror story points out the importance of having communication skills and being able to anticipate peoples’ actions. In the course of a six-week iterative cycle on another project, one of us worked with a select group from the team on a special prototype. One member of this prototyping group appears to be unable to deal with abstraction at all. He repeatedly asked extremely low-level questions that had the mentor stumped. The prototyping group had a tight schedule and a lot to accomplish. The low-level questions didn’t need immediate and exhaustive answers because, well, this is Smalltalk and we trust that

A good mentor gets great personal satisfaction from helping others to learn and grow.

the longstanding base classes do what they are supposed to do.

The mentor's response to many of these questions was, "Don't worry about that right now. You don't need to know that level of detail." It wasn't until several weeks later that the mentor found out the team member interpreted those answers as condescending. Unfortunately, the mentor wasn't told how the member felt early on, and the mentor had no clue he had caused problems with those answers.

The subsequent resolution came about when a third person on the team talked to both the mentor and the mentoree about how the prototyping project had gone, and discovered the communication impasse. So, for the mentors out there (and the mentor wannabes!), be careful about the way you phrase things, and pay close attention to the reactions of your mentorees. Epilog: Once the mentor was told of the problem, the mentor-mentoree relationship became productive again.

It is crucial in a project featuring mentors that there be a regular mechanism for mentorees to be told of their progress, and for mentors to be told of their progress. In a previous column we touched on the subject of end-of-cycle reviews. If the team is reflecting about what went well and what needs improvement at the end of each iteration, then telling the mentors how it is doing should be incorporated into this process. If necessary, make this feedback anonymous to encourage people to be honest in their comments.

MENTOREES

Just because a manager has recruited one or more mentors for a team doesn't mean they will be well utilized. The old expression "You can lead a horse to water but you can't make him drink" appears to be applicable to mentoring. No matter who the mentor is and how terrific he or she might be in that role, some people just won't drink! No doubt there are various reasons for it, and it is not something the manager can mandate. It might be helpful for managers and would-be mentors to know that team members typically fall into one of three categories when it comes to mentoring: eager, neutral, and (dare we say it) hostile or suspicious.

Mentor-Eager. The mentor-eager developer doesn't fear looking "dumb." He or she doesn't hesitate to ask questions, being concerned about proper OO design and learning how to do things "right." This sort of team member asks the mentor for reading suggestions, explores the class library, and doesn't want the mentor to do the work for them. The member wants reviews of decisions about design, algorithms, and so forth and requests design and code reviews. Working with this kind of team member is a pleasure—it's what gets the mentor through the tough assignments!

Mentor-Neutral. The mentor-neutral developer needs to be shown the mentor's value. If there is a good personality match, the neutral developer might become eager. If there's a personality clash, he or she can become mentor-hostile. If personalities are not an issue, this developer might still not make much use of the mentor.

A "neutral" might simply be a person who always prefers to figure out things for themselves. Regardless of how good the mentor is, a "leave me alone" developer will not make use of him or her. There's no point in the manager or mentor forcing the issue.

If a neutral feels comfortable "picking the mentor's brain" on his own terms, then short periods (a couple of hours) of "two-on-a-tube" might be the best way to work with him. Our Smalltalk team used this technique at Tektronix in the 1980s. The mentor sat with the mentoree, with the mentoree in control of the keyboard and mouse. They both worked through a specific problem, making use of the white board and exploring in the image.

This short-term, focused activity, aimed at solving a specific problem, was very effective in demonstrating the mentor's value to a neutral.

Some people thoroughly enjoy working together closely, and two-on-a-tube periods can sometimes be over a week in duration, depending on the problem to be solved. We recently used this technique for several days in a Smalltalk boot camp we ran, and some people enjoyed it so much they did it for almost the entire two weeks!

Mentor-Hostile. A senior person in nonOO areas who lands back at the beginning of the learning curve with Smalltalk will sometimes reject any attempts at mentoring him. We've found that sometimes the mentor-hostile feels so threatened or insecure that he becomes downright hostile. The mentor hostile is used to being the mentor, not learning from a mentor.

In other cases, team members pay lip service to the value of mentoring, but are in fact disguising the desire to prove they can do it without help. They express this desire as needing "the freedom to make my own mistakes." This is a sure indication that the mentor-hostile developer is one who challenges all suggestions the mentor makes, and delights in finding and pointing out bugs in the mentor's code (hey, no one is perfect).

As we already said, there is no point in trying to force these developers to accept mentoring. The manager needs to determine whether their insistence on making their own mistakes is hurting the project. If it is, perhaps the person belongs back in a role that makes use of their existing expertise.

Organizations that adopt object technology—Smalltalk in particular—must realize that their whole team might not learn and progress at the same rate. Some people may

*Just because a manager has
recruited one or more
mentors for a team
doesn't mean they will be
well utilized.*

never really “get OO.” After a reasonable time period (perhaps as long as nine months), the people who still haven’t gotten it need to be given alternatives. Neither the mentor nor the developer is to blame. Not all people are able to think abstractly, and they need to be given the chance to contribute to the organization in a job for which they are suited.

CONCLUSION

“Smalltalk guru” is not the equivalent to Smalltalk mentor. Not all team members will accept mentoring, and not all team members will get OO. Do the best you can as a mentor and as a developer, and try to keep egos out of the equation. If personality clashes are a problem, maybe the mentor has to go. This is a tough call that the manager will have to make. Good luck and happy mentoring! **S**

References

1. THE WORDSWORTH CONCISE ENGLISH DICTIONARY, Hertfordshire. Wordsworth Editions Ltd., 1994.
2. Steinman, Jan, and Yates, Barbara, “Secrets to Building Successful Smalltalk Teams,” tutorial at Smalltalk Solutions, March 1996, New York, N.Y.
3. Steinman, Jan, and Yates, Barbara, “Special” Team Members,” *The Smalltalk Report*, V5N6, February 1996, pp. 15-17, 28.

continued from page 23

OVERALL

Overall I’m quite impressed with both of these implementations. As prerelease products, they’re obviously immature in some areas. For example, I had difficulty with the debugger in both of them. The GUI builder in Dolphin didn’t work yet, and MT doesn’t appear to have one. On the other hand, in a lot of areas, they’re surprisingly mature. They already have advanced features like finalization and exception handling in place. Inevitably, it will be a while before they’re fully loaded with those features that have nothing to do with a language, and everything to do with a successful project: industrial-strength source code control, native database connections, extra widgetry, report writers, business graphics, and so forth. Nevertheless, they show enormous potential, and are well worth your while to investigate.

Of the two, I expect MT to be the first choice for those looking for cool new features, and for those doing things that are traditionally difficult in Smalltalk (e.g. server-based Smalltalk, very tiny apps). Restriction to the newest version of NT will lessen their impact in the short term. Dolphin Smalltalk, with a very low price, Win95 support, MVC, and ActiveX applet support, has real potential to become the Smalltalk for the masses.

Both are filling niches that are under-represented by current implementations, and I hope they will enjoy great success. **S**



John Pugh



Paul White

News from the PP-D Users Conference

We have literally just gotten off the plane from attending the ParcPlace-Digitalk Users Conference in Anaheim, CA. It's deadline time, but we thought we would try to squeeze some thoughts and news from the conference into this issue. In our next issue, Ron Charron will provide us with an in-depth review.

The major news and a surprise, we're sure to many of you, is that there will be new releases of VisualWorks, VisualWave, and Visual Smalltalk Enterprise. Yes, this is not a typo, we said Visual Smalltalk Enterprise. This will be good news to Digitalk customers, but we wonder whether we would have seen far fewer defections by these customers to IBM had this announcement been made at last year's conference, rather than the announcement that support for Visual Smalltalk would end with the first release of "Jigsaw." So what of the merged product? The strategy remains the same—to move to a common image—but it will be a gradual process culminating in a future VisualWorks 4.0 (or Jigsaw 1.0) release. Some features of the "Jigsaw" technology were previewed, such as native widgets (initially for Windows only), a much enhanced ADE known as Cascade, a small base image augmented by SLL technology, and solutions to the "namespace" problem.

PP-D's plans are both Web- and server-centric. It describes itself as a client/server/Web company that would like to occupy the application server space dominated currently by GemStone, but with a heavy emphasis on Web technology. VisualWave and Distributed Smalltalk were dominant in demonstrations of forthcoming technology, and a VisualWorks server product was promised.

So what of Java? PP-D has handed responsibility for its Parts for Java product, and the development of Java components (Java Beans), to ObjectShare Inc. (of WindowBuilder/Pro fame). The guys from OSI have long demonstrated their ability to deliver quality components in the Smalltalk world, and this would appear to be a smart move on the part of PP-D, which should allow it to concentrate on its core Smalltalk business.

So what of merging Smalltalk and Java technologies? This is clearly going to (and needs to) happen—in some form or another. It was a delight to listen to the venerable George Bosworth describe the plusses and minuses of various possibilities, such as Java running in a Smalltalk engine or a Smalltalk plug-in for Netscape. Unfortunately, it appears that we will have to wait awhile before we see any significant happenings from PP-D in this area. There was also an excellent keynote talk from a Mr. Peter Coffee (no kidding!), but we'll leave that to our correspondent Ron Charron.

Smalltalk did meet Java in an interesting third-party product called Route 1 from Applied Reasoning, which allows existing VisualWorks and VisualWave applications to deploy on the web using a Java-based client. An advantage of this approach is that the constraints of HTML are removed. The real application—multiple windows, dialogs, and all—runs inside a Web browser; Java objects on the client side are connected via a lightweight ORB to a VisualWorks server. Performance is enhanced, since there is no longer a need to send an entirely new Web page across the network with each user interaction. Java widgets (clones of the Smalltalk widgets) reside on the client side, which again enhances the speed.

PP-D has plans to beef up its Partners Program, an announcement that is long overdue. But PP-D clients will not move to a new release until third-party products such as Envy and TOPLink also have new releases to match. Working with its partners is essential.

In summary, PP-D is on the way to merging the disparate technologies embedded within the Visual Smalltalk and VisualWorks product lines, using an interesting technology in the pipeline. While it takes shape, the existing product lines will continue to be supported.

Oh, and thanks to PP-D for giving us the run of Disneyland for an evening. No lines! We "big kids" were in Disney Heaven.

Enjoy this issue.

Product Announcements

Smalltalk-Based ORB Features Tools

DNS Technologies Inc. is releasing SmalltalkBroker for IBM VisualAge. Developed with assistance from the IBM Object Connection Program, SmalltalkBroker is a Smalltalk-based CORBA 2.0-compliant Object Request Broker (ORB) and a set of supporting development tools. The ORB provides standard CORBA interoperability between VisualAge Smalltalk objects and other CORBA ORBs, connecting objects written in Smalltalk, C++, and Java.

DNS Technologies Inc.,
160 Spear Street, Suite 740,
San Francisco, CA 94105;
v:415.536.1600, f:415.536.1616;
<http://www.dnssf.cfs.com>.

IBM Connects Smalltalk to Web

IBM announces connections to Lotus Notes and the Web for its Visual Age for Smalltalk application development tool. The VisualAge for Smalltalk directly accesses and modifies Lotus Notes databases, and uses Lotus Notes data alongside a relational database. In addition, users can create reports and multimedia applications or access data across networks, using the communication protocols supported by VisualAge for Smalltalk. IBM has also expanded its family of VisualAge tools: VisualAge for BASIC for Windows (code-named BART) introduces an easy-to-use object-oriented scripting technology that supports multiple platforms and component models (a version was also announced for OS/2); TeamConnection for Windows brings enterprise-level management services for application development to the Windows desktop, and offers an advanced object repository.

VisualAge for COBOL is a rapid application development tool that generates C++ and COBOL applications from a Windows desktop.

IBM, Route 100, PO Box 100,
Somers, N.Y. 10589;
v:914.766.1211, info@ibm.com

Providing Smalltalk Dynamic Links

Virtuality Corp.'s EyeClasses 2.0 is a significant upgrade to its code navigation tool for IBM's VisualAge Smalltalk environment. As a special promotion, any site can receive one copy free of charge. EyeClasses turns Smalltalk into a dynamically linked hypertext environment. This upgrade works with whatever browser you've chosen, standard or Trailblazer, and brings up part editors for VisualAge parts and class browsers for other classes. Links can also be followed from method names, pool constant names, and other symbols.

Virtuality Corporation,
682 Mason Road, Milford, NH 03055;
v:603.672.3731, f:603.672.3743

Booking on VisualAge

Walter Fang's VISUALAGE FOR SMALLTALK DISTRIBUTED: DEVELOPING DISTRIBUTED OBJECT APPLICATIONS describes and explains the entire process of designing and building a distributed object application with the VisualAge Smalltalk Distributed feature. The book contains an overview of the features and architecture of Smalltalk's Distributed feature; sample applications components with supporting documentation to illustrate design and coding; and recommendations for building distributed object applications with VisualAge. The target audience includes software

development managers, designers, and other planning to develop client/server and peer-to-peer applications with distributed objects when using VisualAge.

ISBN 0-13-570805-2, \$40.00

Prentice Hall PTR, Order Processing Ctr.,
PO Box 11070, Des Moines, IA 50336;
v:800.947.7700, v:515.284.6751,
f:515.284.2607; orders@prenhall.com

Smalltalk Tool Analyzes Behavior

Arbor Intelligent Systems Inc. announces Probe, a tool used to determine the exact behavior and performance characteristics of VisualWorks Smalltalk applications. Probe collects runtime performance data including method invocation, process activity, instance creation and life span, memory usage, garbage collection, and exception tracking. Probe is also useful for tasks such as code review, architecture review, reverse engineering, learning Smalltalk theory, and testing.

Arbor Intelligent Systems Inc.,
538 North Division, Ann Arbor, MI 48104;
v:313.996.4238, f:313.996.4241;
<http://www.aisys.com>

GUI Components For Visual Smalltalk

BOK Technologies Incorporated announces ObjectSchedule, a set of advanced GUI components for Visual Smalltalk for Win32, and OS/2. ObjectSchedule includes advanced Smalltalk GUI components for scheduling, resource management, and project monitoring applications.

BOK Technologies Inc.,
5476 Trans-Island Avenue,
Montreal, PQ, Canada H3W 3A8;
v:514.485.6690, f:514.485.2095;
72730.655@compuserve.com

Editors

John Pugh and Paul White
Carleton University & The Object People

SIGS Publications Advisory Board

Tom Atwood, *Object Design*
 François Bancelhon, *O₂ Technology*
 Grady Booch, *Rational*
 George Bosworth, *ParcPlace-Digitalk*
 Jesse Michael Chonoles, *Lockheed Martin ACC*
 Stuart Frost, *SELECT Software*
 Adele Goldberg, *ParcPlace-Digitalk*
 Thomas Keffer, *Rogue Wave Software*
 R. Jordan Kriendler, *IBM Consulting Group*
 Thomas Love, *Consultant*
 Bertrand Meyer, *ISE*
 Meilir Page-Jones, *Wayland Systems*
 Bjarne Stroustrup, *AT&T Bell Labs*
 Dave Thomas, *Object Technology International*

The Smalltalk Report Editorial Board

Jim Anderson, *ParcPlace-Digitalk*
 Adele Goldberg, *ParcPlace-Digitalk*
 Reed Phillips
 Mike Taylor, *ParcPlace-Digitalk*
 Dave Thomas, *Object Technology International*

Columnists

Jay Almarode, *GemStone Systems Inc.*
 Wayne Beaton, *The Object People*
 Kent Beck, *First Class Software*
 Dwight Deugo, *The Object People*
 Juanita Ewing, *ParcPlace-Digitalk*
 Bob Hinkle, *Consultant*
 Tim Howard, *FH Protocol, Inc.*
 Ralph E. Johnson, *University of Illinois*
 Alan Knight, *The Object People*
 Mark Lorenz, *Hatteras Software, Inc.*
 Jan Steinman, *Bytesmiths*
 Rebecca Wirts-Brock, *ParcPlace-Digitalk*
 Barbara Yates, *Bytesmiths*

SIGS Publications Group, Inc.

Richard P. Friedman, Founder, President, and CEO
 Chris Keating, Publishing Director—US Magazines
 John McCormick, Editorial Director
 Margherita R. Monck, General Manager

Editorial/Production

Kristina Joukhadar, Senior Managing Editor
 Elisa Varian, Director of Production and Manufacturing
 Jan Foster, Cover Design
 Douglas Finlay, Assistant Managing Editor
 Serena Tesler, Production Editor
 Erika Romero, Desktop Designer
 Margaret Conti, Manufacturing Coordinator

Circulation

Elayne Glick, Circulation Director
 Byron Scarlett, Fulfillment Manager

Advertising/Marketing

Gary Portie, National Sales Manager
 Elisa Marcus, Advertising Manager, Central US
 Michael W. Peck, Advertising Representative
 Allesandra Kath, Exhibit Sales
 212.242.7447 (v), 212.242.7574 (f)
 Nancy Beuschel, Promotions Manager for Magazines

Administration

Margherita R. Monck, General Manager
 David Chatterpaul, Senior Accounting Manager

SIGS PUBLICATIONS
 Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, THE SMALLTALK REPORT, THE X JOURNAL, JAVA REPORT, OBJECT CURRENTS and THE X SPOT (ONLINE), OBJECT EXPERT (UK), JAVASPEKTRUM and OBJECTSPEKTRUM (GERMANY)



For more object news and analysis, check out SIGS online at <http://www.sigs.com>

Table of Contents

November/December 1996

Vol 6 No 3

Features

Controlling the VisualWorks NewSpace 4

John McIntosh

This article is the first in a series on how Garbage Collection is done in VisualWorks. It focuses on how to improve the performance of your application and reduce its memory footprint by altering NewSpace Thresholds.

Class Naming and Privacy in Smalltalk 11

Nik Boyd

Smalltalk lacks mechanisms for defining private classes, and private methods. Without private classes, class naming conflicts can occur. Without private methods, encapsulation suffers. First class subsystems with private classes can resolve both problems.

Columns



Getting Real: Putting it All Together 18

Jay Almarode

Facing intense competition and a changing marketplace, the IS department has been given the directive to reengineer its software systems to be more flexible and easier to maintain. Here's what its going to do...

Conference Review 24

ParcPlace-Digitalk User Conference

Ron Charron

PP-D seems to have some very good engineers, and hopefully they will manage to bring all of this marvelous technology together. If PPD can pull it off, it is likely to become a major vendor in the new OO CSW arena.

Departments

Editors' Corner 2

Recruitment 28

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar-Apr, July-Aug, and Nov-Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. © Copyright 1996 by SIGS Publications. All rights reserved. Reproduction of this material by electronic transmission, Xerox or any other method will be treated as a willful violation of the US Copyright Law and is flatly prohibited. Material may be reproduced with express permission from the publisher. Bulk rate U.S. postage paid Lancaster, PA, permit 161. Canada Post International Publications Mail Product Sales Agreement No. 290386.

Individual Subscription rates 1 year (9 issues): domestic \$89; Mexico and Canada \$114, Foreign \$129; Institutional/Library rates: domestic \$199, Canada & Mexico \$224, Foreign \$239. To submit articles, please send electronic files on disk to the Editors at 885 Meadowlands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

POSTMASTER: Send domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sigs.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

Controlling VisualWorks' NewSpace

A space for everything in its space

John M. McIntosh

Recent discoveries made while building a system that worked as a server instead of as a typical GUI application have led me to write this article. During the testing phase, I was surprised to see that the image grew from a starting size of about 8 MB to roughly 18 MB before it stopped requesting memory from the operating system. This behavior led to the questions: Why does it grow? Why does it stop growing? A manual garbage collection usually returned the response that 8 MB of memory had been freed. Most puzzling. Although the image didn't grow continuously, I thought I had understood how it used memory, and now was forced to take a closer look at the problem. This article is the first of a series explaining how Garbage Collection (GC) is done in VisualWorks.

My starting point was extensive reading of the ParcPlace-Digitalk manuals and examination of the image. I ran across the MemoryPolicy class comment, which states: "A typical memory policy might be to run the Incremental Garbage Collector (IGC) in the idle loop, in low-space conditions, and periodically in order to keep up with the OldSpace death rate." Light bulbs! My application does not fit the regular pattern of GUI applications! Idle times are a rarer event for server applications, since they do not enjoy human interface pauses. Instead, they might service many users, and always have a high-activity level. Armed with my apparent lack of memory-management knowledge, I embarked on a journey to discover exactly how Smalltalk deals with memory beyond the Scavenge.

GARBAGE COLLECTION

To understand how the MemoryPolicy class interacts with the image, one first needs to step back and understand how garbage collection works. Some background information can be found in Kent Beck's "Garbage Collection Revealed," *The Smalltalk Report*, Vol. 4, No. 5, Feb. 1995, which talks about VisualSmalltalk, and gives the reader a

detailed introduction into GC theories. More information can be found on the Web at <ftp://ftp.netcom.com/pub/hb/hbaker/home.html>, which contains access to a number of papers on GC theories. Hewitt and Lieberman's paper "Lifetimes of Objects,"¹ and David Ungar's classic paper "High Performance Smalltalk Systems,"² also lay groundwork for the GC logic used by current commercial Smalltalk systems. These papers point to a key discovery: namely, that most objects are short lived. In fact, 80 to 98

percent of objects die shortly after birth. The survivors generally live for a long time.

With this observation in mind, a typical Smalltalk system divides memory into two areas: NewSpace for object creation and OldSpace for long-lived objects. The objective of a memory-management system is to aggressively do GC work on a

The objective of a memory-management system is to aggressively do GC work on a small area of memory.

small area of memory. This activity can be done within the pause time between keyboard keystrokes, so the impact on the user isn't noticed. From time to time, checking all objects in a multimegabyte image for survivors is done, by using an incremental GC that runs when the system isn't doing more important work. Running short on memory will trigger more drastic measures to find and remove dead objects, before declaring a critical memory shortage. An application that doesn't match well with the expected behavior will suffer from pauses, and possibly use excessive swap space as the OldSpace GC logic attempts to fix a problem that might be solvable within the domain of the NewSpace GC logic.

NEWSPACE GC LOGIC

NewSpace is really three areas, with a tenuring extension³ to make four. These areas are used as a base for the garbage-collection algorithm. 'Eden' is where objects are first allocated, or born. The next two areas are Survivor spaces. One contains live objects, while the other is empty, being used during an event called "The Scavenge." A

fourth space, 'LargeSpace', is used to handle large objects as a modification to the original algorithm, and attempts to reduce the movement of large byte objects between Survivor spaces. Strings that exceed roughly 1K are created in LargeSpace with a link to Eden.

In a small VW 2.5 Windows NT image, the sizes of the various spaces are:

Eden	204,800 bytes
Survivor Space A	40,960 bytes
Survivor Space B	40,960 bytes
Large Space	204,800 bytes

THE SCAVENGE

When Eden fills to the Eden byte-used threshold, the VirtualMachine (VM) invokes an event called a scavenge. The verb "to scavenge" is aptly defined in the *Webster's New Collegiate Dictionary* as: "to remove (as dirt or refuse) from an area, or to salvage from discarded or refuse material." During a scavenge, the VM locates all objects in Eden and the active SurvivorSpace reachable by the systems roots, and copies them into the empty SurvivorSpace. Once the scavenge examines Eden and the original active SurvivorSpace, those spaces now only contain dead objects, and are deemed empty. Memory allocation starts again, with objects being placed into Eden.

As you can see, survivor objects are shuffled between the two SurvivorSpaces on each scavenge, with new objects being added from Eden. Objects that die in

SurvivorSpace are not copied during the scavenge, and overall growth is based on how good or bad your application is in creating and holding new objects. You will notice that 40K or so of memory isn't very big, so once the number of bytes in SurvivorSpace reaches the defined threshold, the scavenger will tenure objects from SurvivorSpace to OldSpace until there is room in the SurvivorSpace.

OLDSPACE

OldSpace is many memory segments that combine to form a virtual chunk of contiguous memory. This leads to the external behavior shown to the hosting operating system. A VW image will grow by chunks; once the memory is allocated from the hosting operating system, it is not returned. Images never shrink, they just grow. Great, but with all that activity hidden in the VM, can NewSpace garbage collection be controlled? Certainly. In fact, you need to control garbage collection to solve a problem known as the 'Early Tenuring Issue'—the result of an application holding objects for a few hundred milliseconds too long.

When this happens, objects are tenured into OldSpace too early and promptly die, thus defeating the generation scavenging logic. This shifts NewSpace GC work to the OldSpace GC logic. To show how this happens, let us create an object that will artificially hold items and cause the early tenuring problem. We will then alter the size of SurvivorSpace to observe how it will affect performance. Source code will follow this article. To alter the size of NewSpace, you must use `ObjectMemory class>>sizesAtStartUp:`

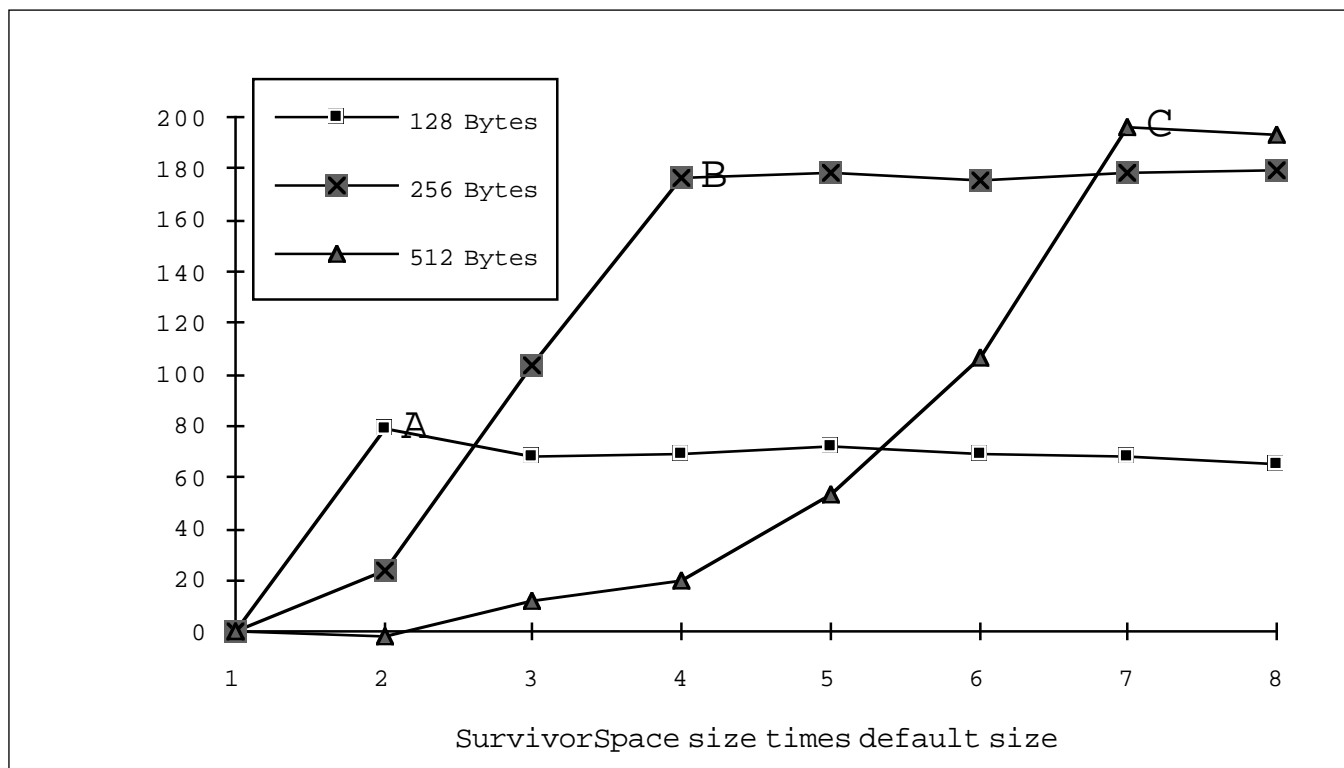


Figure 1. This figure shows restricted image growth, and allocation-rate improvements against default NewSpace size of 40K.

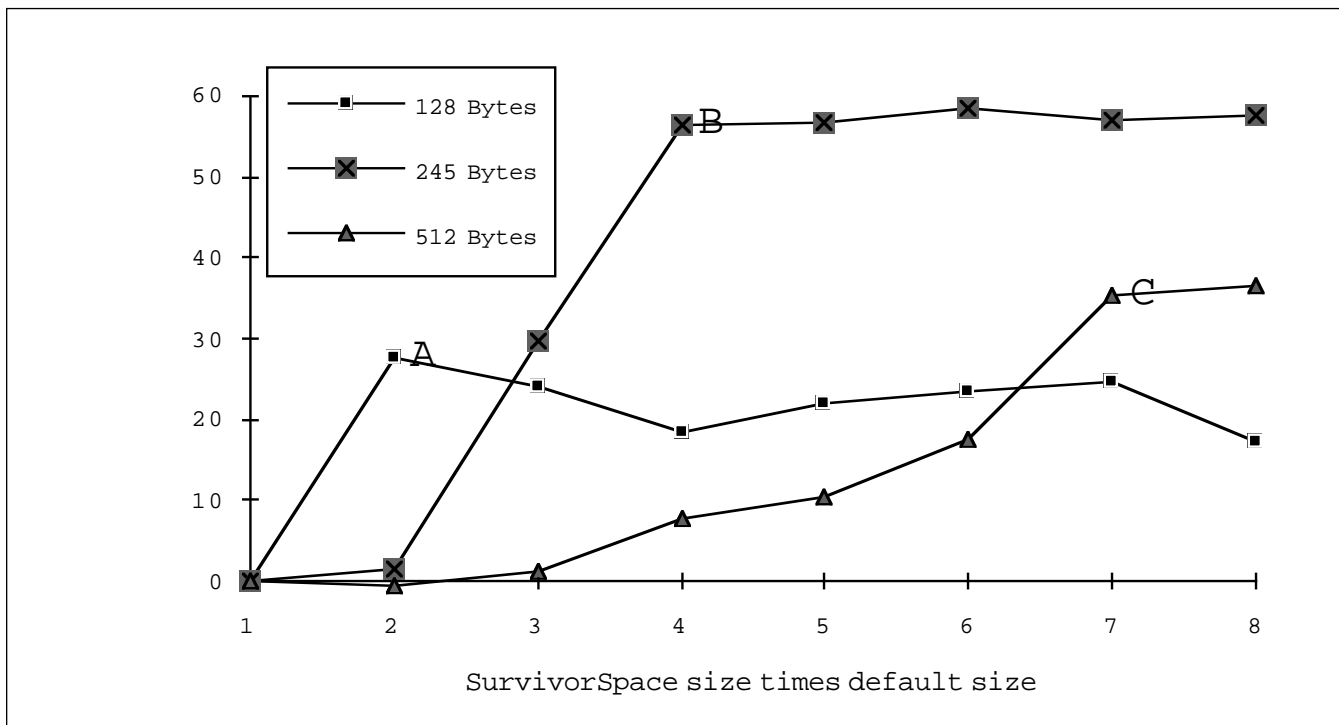


Figure 2. This figure depicts unrestricted memory growth.

and `ObjectMemory class>>thresholds`: to change the default sizes and thresholds, which were chosen by ParcPlace. The `sizesAtStartup`: method allows me to alter the amount of memory VW allocations for each memory area at startup. The `thresholds`: method dictates the thresholds for Eden, Survivor, and LargeSpace.

Two conditions were tested. Since memory is not free, I limited the amount of memory that the image could allocate for one of the tests. The other had full freedom to extend the image. For both tests, I altered the size of NewSpace from two times the default size up to eight

OldSpace is many memory segments that combine to form a virtual chunk of contiguous memory.

times the default size. These changes allowed us to observe the effects of a survivor space that varied from 80K to 320K.

The `EarlyTenureTest` object allocates an Array of 500 elements. For a certain number of seconds, a loop is performed, where a new String object of a given size is allocated and placed in the Array, starting at element one. The index is incremented, and a new string is allocated into the next element. When the last element of the Array is reached, it starts again at element one. If the SurvivorSpace isn't large enough to contain the full working set of the

Array and its components, some of these strings will be tenured into OldSpace.

To show how the image behaves under different conditions, and how OldSpace GC really impacts performance, we first restrict the image's size to 8MB.

In Figure 1, we see allocation-rate improvements against the default NewSpace size of 40K. `EarlyTenureTest` instances are created using a string size of 128, 256, and 512 bytes. These instances require a working set size of at least 64K, then 128K, and finally 256K bytes. On reviewing Figure 1, it is clear that the 128-byte allocation rate improves by about 80%, when we go to a Survivor size of 2x (see point A). For the 256-byte instance, the SurvivorSpace needs to go to 4x before the allocation peak (see point B). Finally, for the 512-byte instance, we peak at a SurvivorSpace size of 7x, with an improvement of almost 200% to the allocation rate (see point C). The impressive improvement for the 512-byte instance happens when we avoid expensive OldSpace GC work. In all three cases, there is a net improvement in the overall work done by the application. Figure 2 shows what happens if memory growth is not restricted.

Although the increase in memory-allocation rates is not as impressive as in our first case, image growth is affected. Using the default size, the image grows from 8MB to 13MB. Changing the size to 7x keeps the image at 8MB, and improves allocation performance from 25% to almost 60%. Again, points A, B, and C show the plateaus where we get the best allocation improvement for the 128-, 256-, and 512-byte instances. Image growth may be free, but allowing it means managing a larger OldSpace, and this can ultimately impact performance.

For both free-growth and restricted-growth situations,

the ending memory-allocation rate is roughly the same, once we reach seven times the default SurvivorSpace size. The image in both cases stabilizes at a dynamic footprint of about 8MB. A larger SurvivorSpace improves memory allocation throughput and reduces image growth. More is better—a “win/win” situation.

In many cases, changing the size of SurvivorSpace means tenuring problems can be traded for slightly

A larger SurvivorSpace improves memory allocation throughput and reduces image growth.

more time spent on NewSpace GC work. Many of the thresholds decided by ParcPlace-Digital for VW date from 1990, and CPU performance, have greatly increased since then. One can easily increase the amount of memory that the scavenger needs to examine, without noticing any effects on response time; and as our examples show, you can improve your application's performance by 200%!

Of course, your application may not have a small working set. Even so, some tests are worth doing. Consider altering your SurvivorSpace allocation by a factor of 10x, and observe the final dynamic memory footprint and time needed to complete a certain task.

This article addresses only NewSpace GC work. In an upcoming issue, I will discuss how memory is allocated, and what happens if you don't have sufficient memory on hand when you ask for another MB (or two) of that elusive resource.

SUPPORTING CODE

From VisualWorks(R), Release 2.5 on September 26, 1995:

```
Object subclass: #EarlyTenureTest
  instanceVariableNames: 'holdTooLong counter
  allocationSize trackAllocations logStream canStop
  waitSync '
  classVariableNames: ''
  poolDictionaries: ''
  category: 'JMM-Memory-Paper'!
EarlyTenureTest comment:
```

©1996 John M. McIntosh, All Rights Reserved.
johnmci@ibm.net.

An object that creates the EarlyTenure problem so we can examine NewSpace behavior:

Instance Variables:

holdTooLong <Array> holder for strings of
size allocationSize

table>
counter	<Integer> iterates over the array to place elements
allocationSize	<Integer> holds current allocation size for Strings
trackAllocations	<Integer> hold total number of allocations
logStream	<Stream> log of information about test cycle
canStop	<Boolean> true when I can stop
waitSync	<Semaphore> used to sync work and result log

Note: to test this you must invoke the sizesAtStartup:, then quit and save your image. After restart of the image, invoke thresholds: to reset the NewSpace memory threshold. You may need to adjust sizesAtStartup: if your installation has already altered some of the other memory space sizes.

```
ObjectMemory sizesAtStartup: #(1.0 7.0 1.0 1.0 1.0 1.0).
ObjectMemory thresholds: #(0.96 0.95 0.90)!

!EarlyTenureTest methodsFor: 'actions'!

createElement
  self holdTooLong at: self incrementCounter
  put: (String new: self allocationSize)!

incrementCounter
  self trackAllocations: self trackAllocations + 1.
  ^counter := counter >= self defaultCounter
  ifTrue: [1]
  ifFalse: [counter + 1]!

runForThisManySeconds: aNumber
  "Fork the delay timer, fork the work, when done return the logStream contents"

  self forkTimer: aNumber.
  self forkAllocation.
  self waitSync wait.
  ^self logStream contents!

writeSize
  "Print the time, memory footprint, scavenges and total allocations for our records"

  self logStream nextPutAll: Time now printString;
  space;
  nextPutAll: ObjectMemory
  dynamicallyAllocatedFootprint printString;
  space;
  nextPutAll: ObjectMemory current numScavenges
  printString;
```

```
space;  
nextPutAll: trackAllocations printString;  
cr! !
```

```
!EarlyTenureTest methodsFor: 'defaults'!
```

```
defaultCounter  
^500!
```

```
defaultPriority  
^Processor userBackgroundPriority!
```

```
defaultSize  
"Do not change over 1000 bytes "  
^512! !
```

```
!EarlyTenureTest methodsFor: 'accessing'!
```

```
allocationSize  
^allocationSize isNil  
  ifTrue: [allocationSize := self defaultSize]  
  ifFalse: [allocationSize]!
```

```
allocationSize: aNumber  
allocationSize := aNumber!
```

```
canStop  
^canStop!
```

```
canStop: aFlag  
canStop := aFlag!
```

```
holdTooLong  
^holdTooLong!
```

```
holdTooLong: anArray  
holdTooLong := anArray!
```

```
logStream  
^logStream!
```

```
logStream: aStream  
logStream := aStream!
```

```
trackAllocations  
^trackAllocations!
```

```
trackAllocations: aNumber  
trackAllocations := aNumber!
```

```
waitSync  
^waitSync!
```

```
waitSync: aSync  
waitSync := aSync! !
```

```
!EarlyTenureTest methodsFor: 'initialize-release'!
```

```
initialize
```

```
holdTooLong := Array new: self defaultCounter.  
counter := 0.  
logStream := WriteStream on: (String new: 1024).  
canStop := false.  
trackAllocations := 0.  
waitSync := Semaphore new! !
```

```
!EarlyTenureTest methodsFor: 'forks'!
```

```
forkAllocation
```

```
[self writeSize.  
[self canStop] whileFalse: [self createElement].  
self writeSize.  
self waitSync signal]  
forkAt: self defaultPriority!
```

```
forkTimer: aNumber
```


```
[(Delay forSeconds: aNumber) wait.  
self canStop: true]  
forkAt: self defaultPriority + 1.! !  
"-----"!
```

```
EarlyTenureTest class  
instanceVariableNames: ""!
```

```
!EarlyTenureTest class methodsFor: 'instance creation'!
```

```
new  
^super new initialize! !
```

```
!EarlyTenureTest class methodsFor: 'Example'!
```

```
example  
^self new allocationSize: 128; runForThisManySeconds:  
15.! ! 
```

References

- 1) Lieberman, H., and Hewitt, C., "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *CACM* 26,6, June 1983, pp 419-429.
- 2) Ungar, D. M., "Generation Scavenging: A Non-disruptive High-Performance Storage Reclamation Algorithm," *Proceedings of the {ACM SIGSOFT/SIGPLAN} Software Engineering Symposium on Practical Software Development Environments*, June 1984, pp 157-167.
- 3) Ungar, D. M., and Jackson, F., "An Adaptive Tenuring Policy for Generation Scavengers," *ACM Transactions on Programming Languages and Systems*, Vol. 14, No 1, January 1992, pp 1-27.

John McIntosh is an independent Smalltalk consultant. After eight years of building client/server applications, he discovered Smalltalk. It was love at first sight! He is currently building Web applications for a Silicon Valley company. He can be contacted by phone at 800-477-2659 or by email at johnmci@ibm.net.

Class Naming and Privacy in Smalltalk

A single remedy for two design and reuse problems

Nik Boyd

Smalltalk lacks mechanisms for defining private classes and private methods. Without private classes, class naming conflicts can occur. Without private methods, encapsulation suffers. While global name spaces can help resolve class naming conflicts, first-class subsystems with private classes can resolve both problems.

Once a Smalltalk developer has learned the essentials of object-oriented design and programming, new issues regarding object system design and systematic reuse begin to surface. There are two factors that contribute to design and reuse problems in Smalltalk—both related to visibility: 1) all classes are visible to (and usable by) all other classes—Smalltalk has a single global name space; and 2) all the methods of a class are visible to (and usable by) its clients, in spite of the fact that some of its methods may be intended for the private use of the class.

This article considers these issues and proposes a single remedy for both problems, without changing the Smalltalk language and with relatively minor changes to the development environment.

The Name Space Problem

Because classes are globals in Smalltalk, they are visible to all other classes (as well as the programmer). This visibility is excessive. It can contribute to information overload for novice (as well as experienced) Smalltalk programmers. It can also cause class-naming conflicts when teams of developers integrate class libraries that have been developed separately.

Few Smalltalk environments include facilities and tools for integrating and organizing large libraries of Smalltalk classes, though some environments and third-party products provide tools for organizing Smalltalk source code into more manageable units (e.g., packages, applications, and the like). While these facilities help to reduce the number of classes immediately visible to a programmer, they do not eliminate the problem of having a single name space.

Adding prefixes to class names has become the common practice for dealing with this limitation. This practice helps prevent conflicts for commonly used names.

However, while class-name prefixes prevent potential name conflicts, they degrade the readability and understandability of the class names. Understanding degrades further when the prefixes abbreviate what should be meaningful subsystem names.

Previous Approaches to Name Space Partitioning

There are only a few workable approaches to partitioning the Smalltalk class name space. Modular Smalltalk¹ addressed some of the issues related to the name-space problem. However, Modular Smalltalk redefined some of the fundamental characteristics of the language and its development environment.

While clear benefits can be gained from a static version of the Smalltalk language—that is, better performance at runtime—it remains unclear whether some of the existing benefits of Smalltalk might be sacrificed (such as dynamic and rapid application development). Perhaps the best of both worlds can be integrated into some future Smalltalk environments. Smalltalk MT looks promising in this regard.

Meanwhile, other commercial Smalltalk implementations still show the early origins of Smalltalk as an interpreted language. Objects and classes are defined and constructed dynamically using messages. These messages are compiled and evaluated in the context of an image-based object memory.

Global Behavior Pools vs. First-Class Subsystems

Another proposal for dealing with the name space problem was explored.² This proposal used global pool dictionaries to supplement the name space provided by the System Dictionary. The article showed how pool dictionaries (which usually contain system constants) could be extended to hold classes. Then, other classes could subscribe to these behavior pools and use the classes defined in such pools—by including the pool names in the poolDictionaries: portion of their class definitions.

While this proposal does help partition the class name space, the pools themselves are still global, and the classes defined in such pools are still public in the sense that

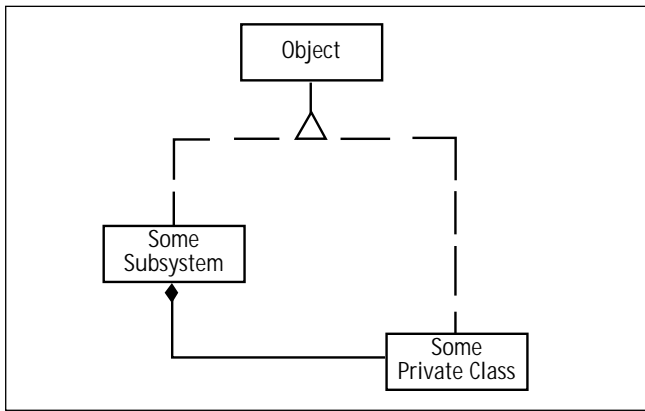


Figure 1. A Subsystem aggregates a Private Class.

they are still global. Such a pool-based facility is similar to C++ namespaces and the hierarchical namespaces provided in QKS' SmalltalkAgents. They provide separate domains for class names without considering whether those classes might need to be encapsulated in an object system design.

In contrast, first-class subsystems support the definition of truly private behaviors. Thus, subsystem classes support the organization and encapsulation of clusters of related classes. This facility is similar to the C++ nested class. Private classes are nested within the scope of a subsystem class, which serves as the public interface for the private classes it contains. A subsystem class can provide or restrict access to the private classes it contains, based on the needs of the subsystem design.

As in C++, global behavior pools and subsystems can complement each other for resolving class name conflicts, and for providing behavior encapsulation. However, the remainder of this article focuses on the benefits of subsystems and support for their implementation in Smalltalk.

The foundations for implementing first-class subsystems in Smalltalk were established in the February 1993 issue of *The Smalltalk Report*,³ in which subsystems were called "modules." The terms "subsystem" and "private class" serve to better describe the intentions underlying this technology, and will be used throughout the remainder of this article.

SUBSYSTEMS AND PRIVATE CLASSES

A Modeling Notation for Private Classes

Subsystems are useful for partitioning the behavior of object systems. Objects are nothing less than small systems, and systems are nothing more than large objects.

*There is no conceptual difference between the responsibilities of a class, a subsystem of classes, and even an application; it is simply a matter of scale, and the amount of richness and detail in your model.*⁴

For these reasons, it would be convenient to have an object model notation that shows the relationship between objects and systems. The notations that have been proposed previously have been internal rather than

external (for example, they depict subsystems by nesting entities graphically). Such internal notations do not scale well graphically when they are applied to the design of large systems, especially when subsystems are nested (for example, when a subsystem class contains a private subsystem class).

First-class subsystems fully contain their private classes, including their definitions. Thus, private classes can be said to be parts of their (public) subsystem class (they are aggregated at the meta level). For this reason, the relationship between a private class and its containing subsystem class will be depicted using a variant of the OMT notation for aggregation.

In Figure 1, Some Subsystem is a first-class subsystem within which Some Private Class is defined. The object models used in further discussions will depict such meta-level aggregations of private classes using a special diamond (as shown in Figure 1).

Defining Subsystems and Private Classes

In addition to introducing a new graphical notation for depicting the design relationship between subsystems and private classes, this article introduces new message formats for defining subsystem classes and their private classes in Smalltalk.

"Define a new subsystem."

SomeSuperclass

subsystem: #SomeSubsystem

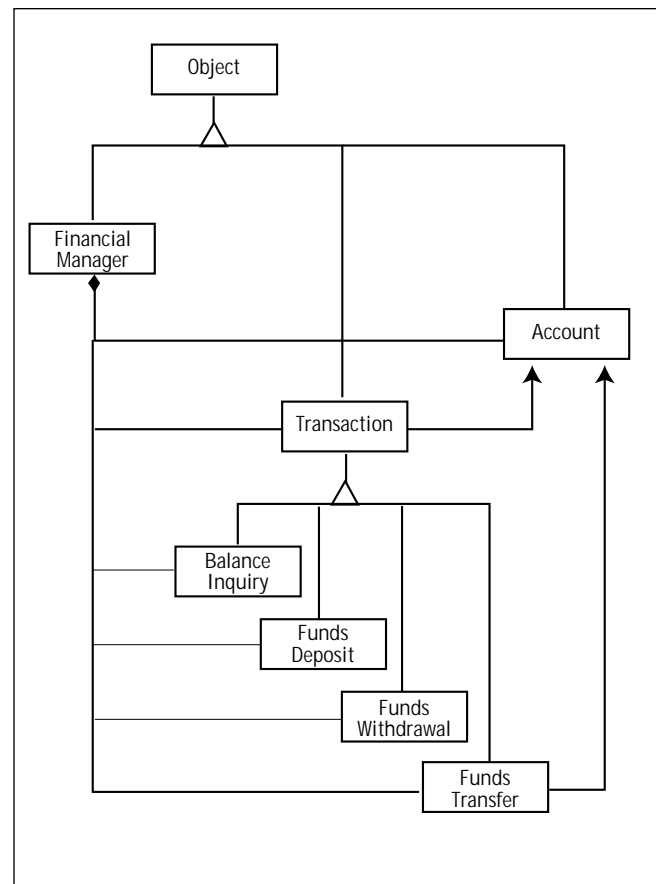


Figure 2. Financial Management Subsystem.

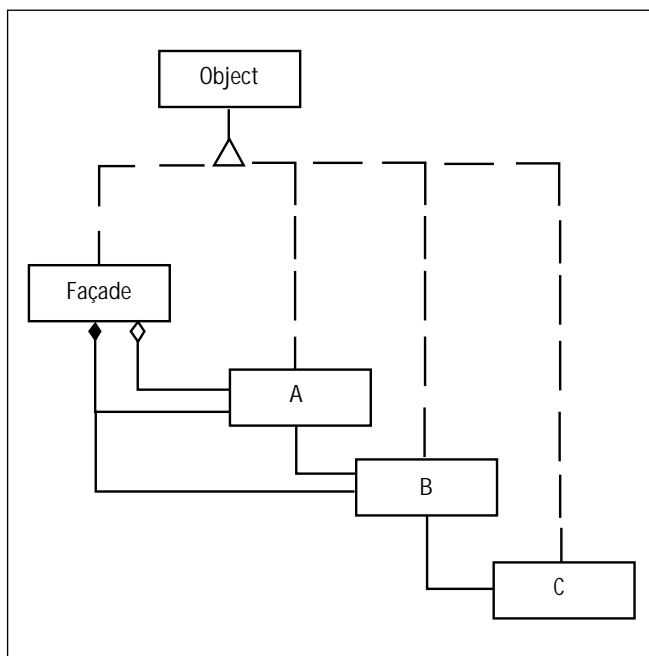


Figure 3. Object Model for a Façade.

```
instanceVariableNames: '...'
classVariableNames: '...'
poolDictionaries: '...' !
```

Notice that subsystem classes support instance variables, class variables, and pool dictionaries, just like ordinary classes. Subsystems are first-class objects. A subsystem class is just like any other class, except that its classPool may contain private classes in addition to the usual class variables. These details will be discussed further in the Implementation section.

Hereafter, in order to simplify the class definitions, the message portions after the subclass name will be elided. However, please remember that the entire list of message arguments is intended and supported for all such abbreviated class definitions. For example, the following partial message shows the abbreviated form of a subsystem definition.

```
"Define a new subsystem."
SomeSuperclass
subsystem: #SomeSubsystem ... !
```

A private class can begin the lineage of a private class hierarchy within a subsystem. Such a private base class must be defined differently from the other private classes derived from it. In particular, such a private base class must identify not only its superclass, but also the subsystem that contains it. The following partial message shows the abbreviated form of such a private base class definition.

```
"Define a private base class."
AnotherSuperclass
subclass: #SomePrivateClass
in: SomeSubsystem ... !
```

Of course, private classes can also be private subsystem classes.

```
"Define a private subsystem class."
AnotherSuperclass
subsystem: #SomePrivateSubsystem
in: SomeSubsystem ... !
```

Once a private class hierarchy has been introduced in a subsystem, the private base class and all the subsequently derived private subclasses can be located relative to the base of the private class hierarchy. The following partial message shows the abbreviated form for defining derived private classes.

```
"Define a private subclass."
SomeSubsystem @ #SomePrivateClass
subclass: #SomePrivateSubclass ... !

"Define a private subsystem."
SomeSubsystem @ #SomePrivateClass
subsystem: #SomePrivateSubsystem ... !
```

Visibility Rules for Classes and Subsystems

The visibility and scoping rules for private classes are similar to those found in C++ for nested classes. The classes defined outside a subsystem are visible to the private classes defined inside a subsystem, while the private classes defined inside a subsystem are not (immediately) visible to the classes defined outside a subsystem. Also, the classes defined within a given scope of visibility are visible to each other. Thus, the private classes defined inside a subsystem are visible to each other, just as the classes defined in the System Dictionary are visible to each other.

Classes defined outside a subsystem may be used directly by name in the methods of classes inside a subsystem. Class names are resolved by looking first in the local scope, and then progressing outward through the enclosing scopes until the named class is found.

The binary message @ serves a role similar to that of the scope resolution operator :: in C++. It can be used to locate a private class relative to its enclosing scope(s). Compare the following Smalltalk and C++ expressions:

Smalltalk	(SampleManager @ #SamplePrivateClass)
C++	(SampleManager :: SamplePrivateClass)

DESIGNING OBJECT-ORIENTED SOFTWARE SYSTEMS

Subsystems provide a coherent way to design and organize Smalltalk classes that collaborate closely. Several examples of subsystem designs are included in the book, *DESIGNING OBJECT-ORIENTED SOFTWARE*,⁵ where the organization of a subsystem for managing transactions against financial accounts is described. Figure 2 shows how this subsystem may be modeled using the new notation for meta-level aggregation.

The class definitions for the financial management classes include the following:

```
Object
subsystem: #FinancialManager ... !
```

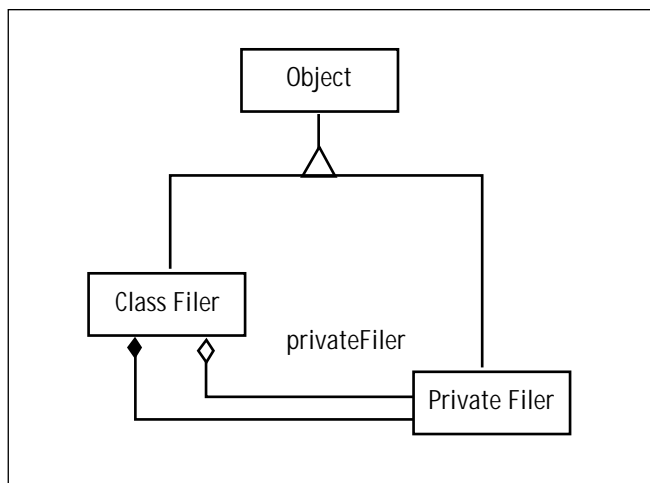


Figure 4. ClassFiler as a Façade.

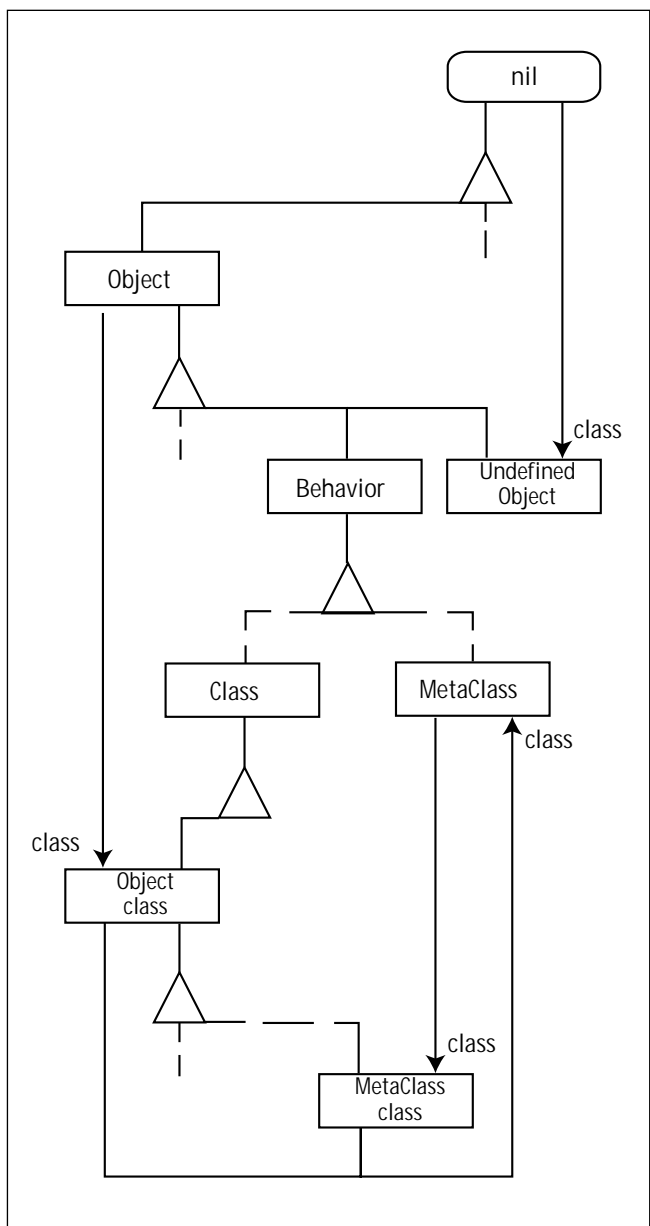


Figure 5. Baseline Behavior Classes.

Object

subclass: #Account in: FinancialManager ... !

Object

subclass: #Transaction in: FinancialManager ... !

FinancialManager @ #Transaction

subclass: #BalanceInquiry ... !

FinancialManager @ #Transaction

subclass: #FundsDeposit ... !

FinancialManager @ #Transaction

subclass: #FundsWithdrawal ... !

FinancialManager @ #Transaction

subclass: #FundsTransfer ... !

Subsystems and the Façade Pattern

First-class subsystems can be used to implement the Façade pattern.

*The Façade pattern provides a unified interface to a set of interfaces in a subsystem. The Façade pattern defines a higher-level interface that makes the subsystem easier to use.*⁵

Depending on the needs of clients, designers can either expose or hide the services provided by the private classes hidden behind the Façade.

Figure 3 shows a Façade class, which uses instances of two private classes and an instance of one public class (defined outside the subsystem). Each instance of the Façade class owns an instance of one of the private classes, while that instance owns an instance of the other private class, which in turn owns an instance of the public class.

This model also serves as an example of the visibility and scoping rules. The Façade can see classes A, B, and C. Classes A and B can see class C (which is public), but class C cannot see classes A and B (which are private).

Private Methods and Client Contracts

Smalltalk systems use classes to encapsulate the structure and state of objects. However, while Smalltalk classes encapsulate the state of their instances, they do not encapsulate their behavior. All the methods of a class are effectively public.

Traditionally, a Smalltalk developer indicates that a method is intended for the private use of the implementing class, using the notation `Private` at the beginning of the method comment. This convention requires the client developer to inspect the source code of the method, in order to discover whether a method is intended for public usage.

In systems that support method organization (i.e., protocols), the method developer can organize the method in a protocol whose name indicates that the methods are private. However, Smalltalk does not enforce the privacy

indicated by either of these conventions. So, client developers sometimes use the private methods anyway, and thereby create dependencies that the class designer did not intend to permit or support.

In Smalltalk, it is not always clear what such privacy means anyway. For example, should subclasses be restricted from using private methods they inherit from their superclasses? While C++ provides explicit access control mechanisms for public, protected, and private members, Smalltalk does not provide any mechanisms for access control.

It can be argued that the traditional notions of access in object-oriented systems are simplified ways of specifying the class of the clients that are permitted to use the methods of a server class. Table 1 suggests how access relates to clients.

This table formalizes a notion that has appeared repeatedly in the literature on object-oriented design: promised behavior. The classes that collaborate closely within a subsystem often exhibit promised behavior, especially when the classes in the subsystem form contractual agreements regarding their services. Thus, it would be advantageous to object system designers if object-oriented languages incorporated and enforced access mechanisms, based on client specifications to establish such formal contracts. Object-oriented languages would improve their ability to model such contracts if they were extended beyond the traditional support for only private, protected, and public access (which are supported by languages like C++ and Java). Indeed, private, protected, and public access mechanisms can be conceived of as specific kinds of promised contracts as shown in Table 2 (with respect to a given server class).

Note that in Table 2 public methods are promised to nil because the class Object and all other root classes are

Access	Implied Client Specification
private	only the implementing class
protected. . . .	the implementing class and all derived classes
promised. . . .	some specific collaborating class (which need not be related by inheritance)
public. . . .	any class (without regard for inheritance)

Table 1 represents how access relates to clients.

Access	Equivalent Contract
ServerClass private	ServerClass promisedTo: ServerClass only
ServerClass protected	ServerClass promisedTo: ServerClass any
ServerClass public	ServerClass promisedTo: nil

Table 2 represents private, protected, and public mechanisms.

Software developers need language facilities that provide design options. This is one of the reasons that C++ has evolved so much over the past several years.

derived from nil—all the root classes have no superclass. Thus, public methods are available to any other defined method, whether the method is defined in a class derived from Object, or any other root class.

Private Classes for Private Methods

The following discussion describes how you can use private classes to implement private methods—even without direct language support for private methods. First, build the public interface using an instance of a subsystem class.

The subsystem instance contains a single instance variable, and the instance variable contains an instance of a private class. The private class contains those methods you want hidden. When the public class (the subsystem) is instantiated, it creates and holds an instance of the private class. Each of the public

methods (in the public class) uses the private methods supplied by the instance of the private class. Figure 4 shows this arrangement using an object model.

The class definitions for this Façade include the following:

```
Object
  subsystem: #ClassFiler
  instanceVariables: 'privateFiler'
  classVariables: "
  poolDictionaries: " !
```

```
Object
  subclass: #PrivateFiler
  in: ClassFiler
  instanceVariables: 'behavior'
  classVariables: "
  poolDictionaries: " !
```

Implementation

This section outlines how the facility for defining subsystems and private classes can be added to Visual Smalltalk. We will focus on those aspects of the Behavior classes that change when subsystems are added. First, note how the baseline Behavior classes are organized in Figure 5. Behavior inherits from Object. Class and MetaClass inherit from Behavior. Object class inherits from Class. The other metaclasses of the subclasses of Object inherit from Object class.

Generally speaking, the class and metaclass inheritance hierarchies parallel each other. Thus, for example:

```
Point superclass == Object
Point class superclass == Object class.
```

However, there is an anomaly at class Object, where

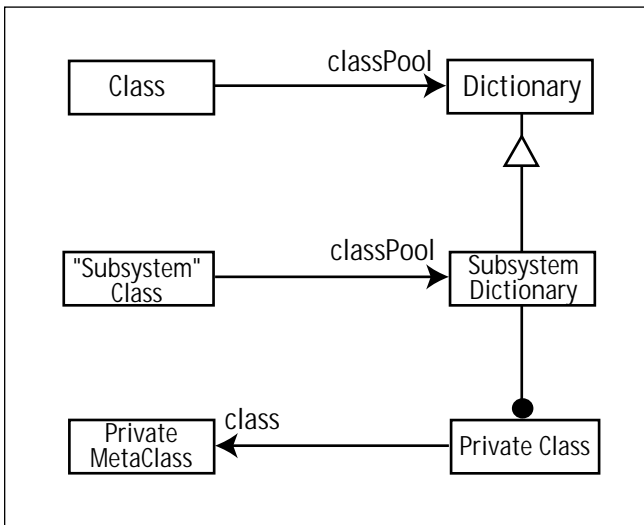


Figure 6. Class vs. "Subsystem" Class.

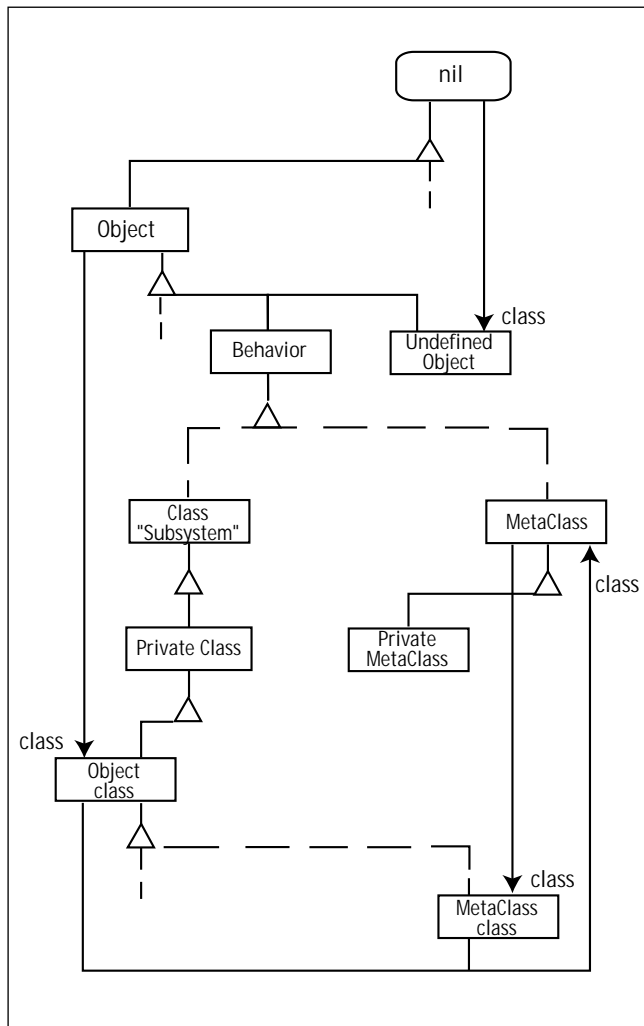


Figure 7. Behavior Extensions for Private Classes.

Object superclass == nil

Object class superclass == Class.

Subsystems and private classes require some minor alterations to these baseline relationships. As noted previously, each subsystem is a class. While an ordinary class uses

a Dictionary for its classPool, a subsystem class uses a Subsystem Dictionary. Each Subsystem Dictionary provides a unique domain for the private classes and class variables of the subsystem. Private class names are mapped to Private Classes, while class variable names are mapped to class variables. Figure 6 provides a model of these relationships.

Each Private Class knows its class (a Private MetaClass). Each Private MetaClass knows its subsystem (a subsystem class). Thus, indirectly, each Private Class knows the enveloping subsystem class. Given the foregoing relationships, Figure 7 shows the relationships for the new Behaviors.

In particular, note how

Object class superclass == PrivateClass.

This relationship replaces the normal baseline relationship, where

Object class superclass == Class.

Because each private metaclass knows the subsystem to which it belongs, the compiler can identify the scopes that enclose the private behaviors (class and metaclass). This simplifies changes to the compiler interface to extend the visibility rules and resolve class names into classes.

CONCLUSION

Benefits of Subsystems and Private Classes

Software developers need language facilities that provide design options. This is one of the reasons that C++ has evolved so much over the past several years. Some of the recent additions (nested classes, templates, namespaces, and runtime type information) show progress toward features found in pure object-oriented systems like Smalltalk, and even some advances over features in Smalltalk.

Two of these C++ features directly address the class naming problem in complementary ways: namespaces and nested classes (i.e., private classes). This article has considered how support for private classes can be added to Smalltalk in conjunction with first-class subsystems. Subsystem classes provide an additional design dimension beyond that provided by ordinary classes. Subsystems and their private classes permit you to:

- resolve class name conflicts with separate name spaces;
- integrate separately developed class libraries;
- organize collaborations between classes into first-class subsystems;
- implement the Façade pattern; and
- define private methods only visible to those in a public interface (subsystem) class. ■

References

1. Wirfs-Brock, A., Wilkerson, B., "An Overview of Modular Smalltalk," OOPSLA Conference Proceedings, ACM, September 1988.

-
2. Beaton, W., "Name Space in Smalltalk/V for Win32," *The Smalltalk Report* 4(1), SIGS Publications, New York, NY, September 1994.
 3. Boyd, N., "Modules: Encapsulating Behavior in Smalltalk," *The Smalltalk Report* 2(5), SIGS Publications, New York, NY, February 1993.
 4. Wirfs-Brock, R., Wilkerson, B., Weiner, L., DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice-Hall, Englewood Cliffs, NJ, 1990.
 5. Gamma, E., Helms, R., Johnson, R., Vlissides, J., DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED ARCHITECTURE, Addison-Wesley, Reading, MA, 1995.

TRADEMARKS

Visual Smalltalk is a trademark of ParcPlace-Digitalk, Inc.

SmalltalkAgents is a trademark of Quasar Knowledge System, Inc.

Smalltalk MT is a trademark of Object Connect, SARL

Nik Boyd has been developing object systems since 1987, when he founded 3rd Person Software (formerly known as XoteryX). In 1993, he released Package Manager/V through The Smalltalk Store. During 1996, he released Package Librarian/V. His experience with OOP includes work with several ParcPlace-Digitalk Smalltalk versions and platforms, as well as work with C++. His research interests focus on tools and techniques that support object-oriented software engineering. Nik may be contacted at 74170.2171@CompuServe.com.



Jay Almarode

Putting It All Together

SCENE 1:

A well-lit conference room in the offices of ABC Corporation. Facing intense competition and a changing marketplace, the IS department has been given the directive to re-engineer its software systems to be more flexible and easier to maintain.

Pat (development manager): Thanks for gathering on such short notice, team members. I've got good news and bad news. As you know, today we gave a demo of our application prototype that we built in Smalltalk to the head of development. The good news is that management was very impressed and decided to give the green light to build all our new applications in Smalltalk. The bad news is that while giving the demo, the VP of development walked in and thought we had already finished the application.

Chris (engineer): What gives? I hope you set him straight.

Pat: The VP saw finished screens with apparently real data and thought the application was running. However, I explained that this was a prototype implemented in single-user Smalltalk, and that the display was using dummy data stored locally in the image.

Bobby (another engineer): So then what happened? What made management sign off?

Pat: I think they were impressed with how quickly we built the prototype. But I think the real kicker was when the VP looked at the order-entry screen and said that we were missing a field showing compatible part numbers. I opened a browser and added an instance variable to class OrderEntry, then added a widget to display it. In less than two minutes, I had the new screen. The VP's mouth just dropped.

Terry (another engineer): Did you save the image so you can check that code into the repository later?

Pat: Of course. So now that we've got the go-ahead, the real work begins. I'm afraid the expectation level is high on this, so we've got to get organized right away. Right

now, our office is organized along functional areas. Each functional area has its own application-development team. I've been reading the latest project management and methodology books, so I've assigned developers from each functional area to design and build our common business object model.

Bobby: You've sent everybody to training classes, but do you think we're ready for prime time?

Pat: I have the highest confidence in all of you. However, I've also contracted mentors from Objects-R-Us to help us out. They'll be on-site for both design and implementation. They're Smalltalk gurus and should get us through any rough spots.

Chris: Yeah, but this is a large system. We're talking over twenty applications, hundreds of users, and lots of objects. How are we gonna make sure that we scale?

Pat: Good question. Our charter is to build objects that span the enterprise. To make sure we know what we're getting into, I've assigned you to teams to look at the following issues. (*Pat goes to the white board and begins writing.*) One team will look at overall system-performance characteristics. We need to get a handle on object counts and required response times. Another team will look at the system configuration and architecture. This includes hardware and fault-tolerance issues, as well as system administration. The third team will look at application-design issues. They should make recommendations for applications to handle multiuser issues and meet performance requirements.

Terry: These are big responsibilities. How are we going to meet them?

Pat: Obviously, these teams do not work in isolation. I expect a lot of cross-communication between these teams, as well as interaction with the individual application-development teams. The three teams I've described and the common business object team have system-wide visibility.

SCENE 2:

A small cubicle filled with books, hardware, and Dilbert cartoons.

Terry: Hey Bobby, got a minute? I was wondering if you could take a look at this questionnaire I created? I'm try-

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a senior software engineer at GemStone Systems Inc., and can be reached at almarode@gemstone.com.

ing to capture the overall system characteristics. I figure I'll get answers for every application, then try to accumulate them for the system as a whole.

Bobby: Hmm, looks interesting. What is the purpose of asking for the number and size of objects?

Terry: I'm trying to come up with some estimate of the object repository size. I've got to make sure we've got enough disks for all applications.

Bobby: Well, just make sure you account for growth. Remember when forecasting built their new models last year, and then didn't have enough space to store them? Also, how can someone tell how much garbage they're producing?

Terry: The mentor from Objects-R-Us gave me this neat little goodie. It should be used only during development, but it tells me which objects were created and which would be collected when a block of code is executed.

```
classmethod: System
newObjectsAndGarbageWhile: aBlock
```

" Return an array of two sub-arrays: the first containing objects that were created, and the second containing objects that became eligible for garbage collection during the execution of the given block. "

```
System _generationScavenge.
System _hiddenSetReinit: 31. "ObjsCreated"
System _hiddenSetReinit: 32. "ObjsDisposed"
System _enableTraceObjs.
```

aBlock value.

```
System _generationScavenge.
System _disableTraceObjs.
```

```
^ #[ System _hiddenSetAsArray: 31,
      System _hiddenSetAsArray: 32 ]
```

(Chris enters the cubicle.)

Chris: Hey, have you heard the latest? We've got approval to go three-tier. Our arguments convinced Pat that we needed a real application server. Now we can implement our applications in whatever vendor's single-user Smalltalk we want, and still share Smalltalk objects on the server.

Bobby: How did you convince them?

Chris: Well, they recognized that we had to access legacy data, and already knew that the object-to-relational mapping was nontrivial. We basically told them that we couldn't afford to send all that data over the wire to each client, perform the mapping to create objects,

and then map changes back to SQL updates. Plus, we couldn't guarantee security or fault tolerance on the client.

Terry: Great. This should make our business object modelers happy. Now they have a single place to maintain their objects. So how are those people doing?

Chris: I heard they've had to do a lot of work. Our original demo did a good job of separating interface from domain objects, but the domain objects mixed application-specific with general behavior. They've been busy figuring out which objects belong on the server and which belong on the client.

Bobby: So how are they figuring this out?

Chris: Our mentor has given us some tips to help them. It's really common sense anyway. You know, large collections, shared objects, secure objects, recoverable objects; they all live on the server. Speaking of recoverable objects, how's it going with system configuration, Terry?

Terry: We're just getting our hands around it. We've come up with an architecture diagram showing which machines will house shared-page caches for the clients. Our plan is to assign certain applications to certain server machines to try to spread out the load.

We're splitting the repository into three raw disk partitions for performance. We'll also dedicate one disk to transaction logs, and schedule a job to run nightly, compressing and copying the logs to tape. We still haven't figured how often we'll checkpoint the repository, though. We haven't heard back from all application areas on their fault-tolerance needs. We asked each area to spell out how

User access:

What is the total number of users?
What is the average number of users logged in at any given time?
What is the maximum number of users that may be logged in?
How many different geographic locations?

Number and size of objects:

What is the total number of objects?
What is the average size of an object?
What will be the largest objects?
What are the sizes of the largest collections?
How much garbage is produced per transaction?

Transactions:

What is the expected transaction rate ...
per day, per hour, per minute, per second?
What is the peak transaction rate?
What is the duration of the longest transaction?
How many objects are read during each transaction?
How many objects are written during each transaction?
How many objects are created during each transaction?

much downtime is acceptable, and when, but I think they're still working on design issues.

SCENE 3:

In the nautilus gym at ABC Corporation. (What? Your company doesn't have a gym?)

(Terry walks over to Chris, who is puffing away on a lifecycle.)

Terry: Hey Chris, how's it going? Have you solved all of our application design issues?

Chris: We're making progress (puff, puff). I've been talking to the chief designers of all the applications, and we've identified the shared collections. We've decided to use some special multiuser collections to reduce the chance of concurrency conflicts. We still have to worry about other chances of conflict, so we've been looking at when to lock objects and when to design for the possibility of conflict.

Terry: Oh really. How do you make that determination?

Chris: I asked developers for each application to define their transactions, and to identify which shared objects would be written for each kind of transaction. We categorized the transactions by their priority, the probability of conflict, and the impact of transaction failure. From there, we began planning our strategies.

Terry: Hmm. . . interesting. How did it go?

Chris: As you can guess, some designers knew exactly which objects they were modifying, while others didn't have a clue. Fortunately, we had the capability to view the set of written objects at transaction boundaries. We ran some test cases and saw which objects were being touched. This really opened up some developers' eyes and we improved the code as a result.

Terry: I'd like to get some of your instrumentation code. So what strategies do you come up with when you don't lock objects?

Chris: We've created a framework to keep a log of important object modifications that should be replayed in the event of concurrency conflicts. Basically, these are temporary objects that are created during the life of a single transaction. We found this was best wired into the application

rather than into the business objects, because it is the application that defines a transaction.

Terry: So how do you make sure these objects remain temporary?

Chris: We found in the manual something called 'transient session state', from which you can reference objects so that they do not get garbage-collected, yet it doesn't cause them to be part of a committed state. Of course, if you reference these objects from some other committed object, they will become persistent.

Terry: I don't understand. How does this all fit together?

Chris: For certain operations, we create objects that encapsulate the modification to a business object. We hang these objects off of a transient session state. If the transaction should experience conflict, we can abort the transaction, then replay the modifications. We had to design this carefully, because sometimes concurrency conflicts are a good thing. Our framework allows us to perform validation after the transaction is aborted, to make sure that conditions still hold to replay the modifications.

Terry: Very clever. I hope you're making this available to all applications.

SCENE 4:

Six months later, in the well-lit conference room.

Pat: Thanks for gathering on such short notice, team members. I've got good news and bad news. As you know, we just deployed our last application and the user response has been positive. There were a few glitches along the way, but we managed to hang in there and deliver all that we said we would. We learned a lot along the way, such as making sure hired consultants really know Smalltalk, and planning for schema modification after applications have been deployed. The bad news is that management has been sufficiently impressed with our deliveries that it wants us to do the same for the international offices, but in less time. In addition, we'll need to replicate objects across distributed servers for local availability. I'm afraid the expectation level is high on this, so we've got to get organized right away. So here is what we're gonna do ... **S**

ParcPlace-Digitalk User Conference Review

Ron Charron

The following is a review of ParcPlace-Digitalk's Third Annual International User Conference, held at the Disneyland Hotel in Anaheim from September 9-11.

For a while, the news coming from PP-D was that Jigsaw was to be the name for the new "merged" image. Somehow, PP-D was determined to pull off the formidable feat of combining Digitalk's VisualSmalltalk image with that of ParcPlace's VisualWorks. The very reason I described this as formidable was because of the resounding differences that separate the two implementations. Beyond the commonality provided by the very similar Collection and Magnitude Classes, and assorted blue-book classes, there lay the tremendous differences in the widget model.

Traditionally, Digitalk's strengths have been in strong platform ties, using native widgets. VisualWorks' customers have commonly sacrificed native widgets for emulated ones. It's not that they favor emulated widgets, but it's both the large range of platforms offered to them and the low-hassle interoperability that have closed many a sale. But many, including me, have longed for the day when native widgets would become available for VisualWorks. So, it was with the hope of hearing this good news that I packed my bag and hitched a ride to the land of Mickey Mouse.

WELCOME TO THE CLIENT/SERVER/WEB COMPANY

Of late, I wake up wondering if I'll have to type in an URL to get my car going in the morning. I've quit subscribing to the newspaper because I surf the airlines about as often as I do the Web. . . and that's a lot. I can now read my favorite newspaper and listen to my favorite radio stations by "jacking-in." Things will never be the same in the Smalltalk world, now that it's caught up in the all-knowing, all-seeing Web.

PP-D has definitely caught Web fever. It takes no more than a quick jump to its homepage to convince you that it looks like it's got more than a 24-hour flu.

And so, in opening the conference, Bill Lyons, presi-

dent and CEO of PP-D, announced that his company was on the path to becoming a major player in the client/server/Web business, and moving away from being a client-centric solutions provider. Indeed, the firm would place more emphasis on server-based solutions. The high profile given to VisualWave during the conference made this fairly clear. Also, Distributed Smalltalk, which is usually a backstage item, was demonstrated in numerous lectures. Let's not forget the high profile given to Parts for Java, available at a computer store near you!

"We're not in Kansas anymore" was a feeling I shared with many an old-hat Smalltalker throughout the conference. This was made obvious in *PC Week* columnist Peter Coffee's keynote address titled: "Smalltalk: Surviving Its Success." He maintained that while OO had become mainstream, management now had to be convinced to go to OO in order to catch up with the competition, rather than as a means to increase the return on investment. His recollection of all the products that had gone OO in the last few years reminded me of a joke about Orville Reddenbacher coming out with "Object-Oriented Popcorn" (P++). And with Java, it was all going to start over with a fury unparalleled. The Java volcano had erupted and no IT shop on earth would ever be safe from its strange seduction. Of course, Coffee reflected—as we all did—that Java was not yet mature, and you really had to be a masochist to want to move away from Smalltalk and into Java. But still, Java is a force to be reckoned with and any tools salesperson who had not smelled the "Coffee" on this one was taking the risk of interacting with Job-Search Beans in line at the welfare office.

Coffee also presented little dark secrets, surprising no one, about how by requesting for `<UnRememberedPrefix Here...>date(96,8,9,0,0,128)` gave an entirely different answer in Sun's Library than it did in Microsoft's Library. (Not even the same day!) So, if you're feeling apprehensive about the year 2000, you're in for a lot more fun if you'll be delivering in Java in the year 2048!

So what's ahead for Smalltalk? In response to the Java

issue, many suggested that a Smalltalk plug-in for Netscape could help fend off the Java tide. During one of the Parts for Java demos, I felt that the coding process that was required, beyond wiring up Java widgets, would bring us to something I hadn't experienced since I started using Methods (introduced by Digitalk in 1985). Although Methods was text-only, you were still coding in an environment that was as interactive as that of modern-day Smalltalks. It is hard for me today to go back to an environment that doesn't have the type of tight integration of the editor, compiler, debugger, and code browser tools that Methods has for PC platforms. Of course, Lisp environments did, but that's another story. So, until Java development environments become as friendly as Methods USED TO BE, most Smalltalkers will feel that coding a non-trivial Java app will feel like taking steps backwards.

THE CASCADE APP-DEVELOPMENT ENVIRONMENT

Cascade was the code name given to the application-development environment to be used in the new Jigsaw images. This new environment for Jigsaw will be able to add constants (or "read-only" objects). There is also a space for entering a string describing variables. So, VisualWorks 4.0 customers will now have a PP-D-supplied version-control and source-management system. The first jigsawed image will be called VisualWorks 4.0. PP-D has indicated that the base runtime image will be small and headless, with just the necessary code to bind with SLLs. Descriptions of system enhancements that support packaging left me believing that VisualWorks customers may have a workable alternative to ENVY/Developer. On that subject, PP-D engineers promised to work with OTI to facilitate support of ENVY products on the new image. OTI says it will be receiving the beta, and will then evaluate the effort involved in doing the port. The presentation on the Cascade architecture led me to believe that this may not be a trivial task. However, OTI is not known for building trivial products, and if we're lucky, we may even see more improvements in ENVY/Manager as it is rewritten for the Cascade environment.

VISUALWORKS 4.0 OR JIGSAW?

On one side we had Bill Lyons saying that Jigsaw was to become "a family of products," and on the other we had just about every engineer describing it as a new image. It is apparent to me there have been some shifts in the way PP-D wants to present its future to the world. The way I pictured it from bits and pieces of lectures and conversations was: Common technology will gradually be integrated into both VisualWorks and VisualSmalltalk images. Eventually, there may be no large, discernible differences between images, so are we looking at "merge by attrition?"

It does make some sense to me, and I think it will make some sense to the customer if PP-D can manage the cost of maintaining the two product lines for a while. Gone is the scare that clients will have to do massive rewrites to get their code to run in the new, merged image. There are

some caveats, however. Wrappers and controllers are going away; a new event model is coming in. If you have invested in custom wrappers and controllers, you will need to re-examine your work and start planning for an event model. Make sure you get on the beta program as soon as possible to plan your migration strategy carefully. The feeling is that market pressures may lead PP-D to release a beta that may indeed be a bit "Alpha-ish." If that is the case, expect possible important changes between beta and general acceptance.

On the negative side of things, that means frequent adjustments to your port strategy. On the positive side, it means that if you are vocal and describe your needs in a precise fashion, you may get a chance to influence the development of the new product. If you have a massive investment in VisualWorks technology, you could protect your investment by making sure that you have some good talent assigned to assessing the beta and the port implications as soon as possible.

So, by talking with PP-D engineers, the first incarnation of Jigsaw will be the VisualWorks 4.0 image shipped with the Cascade environment and with native widgets. Native widgets will include Win32, and Win 95 common controls, with other platforms retaining emulated widgets. Datasets and the like may remain emulated, for lack of an adequate native.

Also, answering my question during the Jigsaw tools session, PP-D had not yet started work on native fonts and color models. Prior to VW4.0, expect releases of VSE 3.12 and VW2.52 in 'Q4.

WE'RE THE OO SERVER PEOPLE

With the acquisition of Distributed Smalltalk from Hewlett-Packard, and with its own VisualWave environment, PP-D wants to position itself as an important player in the OO application-server business. This squarely places them face-to-face with GemStone, which has been gaining more market acceptance since it made this kind of shift in the last few years. One major difference in Gemstone's favor is that it has a solid OODB to back it up.

So what can make PP-D a player in the OO application-server market? The VisualWave demonstrations succeeded in convincing me that it had a viable tool for the development of Web server apps serving mostly HTML, with some support for Java-Script. However, the audience vocalized that more Java support will have to come quickly. Although Parts for Java offered some interesting technology for the Jigsaw line of products, PP-D technical management will have a formidable challenge ahead in making all the parts fit together. If it can put Parts for Java's quality-code development together with the quality server infrastructure of VisualWave, and offer better database support, PP-D may indeed have a winner. However, in the meantime, we will still have to look at "some assembly required" for many corporate applications.

Distributed Smalltalk lectures attempted to dispel the

myth that Smalltalk cannot perform as a server. Exhaustive demos on load-balancing were given. Unfortunately, despite emphasizing that 50 or 100 transactions were running, the engineers failed to explain adequately what these transactions were. Whatever they were, the engineers appeared happy. But people didn't appear too happy with the statement that "two tiers won't do it. . . three tiers won't do it. . . You need a fully distributed system." That may be true someday, but I don't see the masses rushing into implementing distributed systems until they get a better handle on building two- and three-tier OO-based apps. "Thread safety" was an important concern here, so when the subject of database access came forward, the statement that "few database services were thread-safe" did not give me the "warm fuzzies." IIOP was mentioned several times as the preferred communications protocol for the servers. This is interesting, because this is also in line with Netscape's plans.

BYE BYE OSI, HELLO PPD/OSI

If you've made WindowBuilder-Pro your home over the last few years, you'll probably be pleased to know that WindowBuilder Pro for VisualWorks is being rumored. Now that the barriers that once made it impractical are disappearing, along with native widgets and the event model, there is a good probability that the people who brought us WindowBuilder Pro will help bring the new VW Canvas Editor to life. No, it doesn't look like PP-D plans to port Parts to VW anytime soon. Many PP-D engineers have mentioned that the Canvas Editor tools really needed reworking, and that the ObjectShare Inc. acquisition presented PP-D with new "opportunities." I love to see a good virus infect a deserving host.

Since Smalltalk went corporate, programming enthusiasts have been deploring the lack of a low-cost Smalltalk for learning purposes. Now, thanks to ParcPlace-Digitalk and OSI, people wanting to learn Smalltalk will have a decent environment to learn in, and they won't need to take out a second mortgage on their house either. Smalltalk Express is the name of a (FREE) Smalltalk development tool that includes Smalltalk/VTM, Win16, and WindowBuilderTM Pro/V. Surfing the Web all the way down to www.objectshare.com, will give you more details. At the comp.lang.smalltalk.newsgroup, you will learn that SE: (Smalltalk Express) has been proposed as the first

three characters of the subject. CompuServe members will find Section 23 of the ParcPlace-Digitalk forum (CIS:PPDFORUM) devoted to Smalltalk Express.

ALL AHEAD WARP FACTOR NINE


Robi's Dream was entertaining as Robi (Michael Robicheaux), one of PP-D's top gurus and originator of the Canvas Editor Tools, used his VRML RoboBabe as the starting menu for his visions. He began poking and stroking parts of the 'Babe to fire up such phantasms as VRML Class Hierarchy Browsers, Performance Profilers, and Business Bar Charts. PP-D will now follow IBM's lead and have a Developer Certification Program.

What held my attention was OTI's presentation of ENVY/QA, which promises to give you professional code reviews without the hassle of a mentor breathing down your neck. Close to 40 metrics, code coverage, and code publishing tools will no doubt make many large-scale project managers happy. If you aren't excited, then you should be. All of the above are grossly under-addressed problems that were begging for relief. It's nice to see a package addressing all of these issues coming from a major player such as OTI.

CONCLUSION

PP-D appears to have some very good engineers here, and I hope they manage to bring all of this marvelous technology together. If they can pull it off, they are likely to become a major vendor in the new OO CSW arena.

For those who are more interested in what will happen to their investments in VisualSmalltalk or VisualWorks, you no longer have to head for the hills in panic. Neither VSE nor VW are going to be dropped. It may take some time for it to happen, but the goal of a unified image seems likely, maybe for 1998? In the meantime, thanks to Cascade, new goodies should run on both platforms.

Oh, for those who said that they went on the Indiana Jones ride ten times Tuesday night, I don't believe you. . . And, by the way, Disney—five bucks for a beer?. . . Come on! 

Ron Charron is director of corporate services at The Object People. He has assisted major corporations in using Smalltalk in 11 countries... and counting!



John Pugh



Paul White

OOPSLA—Reinventing Itself Again

This month we've just gotten off a plane, this time returning from OOPSLA. This year was the 11th ACM OOPSLA conference, which was held in San Jose, California. As is often the case, OOPSLA can surprise you in terms of the tenor of the conference, and of the focus and interests of those in attendance. This year was no exception.

The overriding focus of this year's conference appeared to be on issues dealing with architectures, frameworks, and patterns. Unlike the past few years, where we've had the feeling the conference served as a showcase for Smalltalk technology, this conference seemed noticeably language-neutral. This was somewhat surprising, given that virtually every other conference having anything to do with objects lately has become a "Java conference." We say this not as either a good or bad thing, but rather as an observation that most attendees appear more interested in discussing new software architectures and their associated benefits and limitations, than waging language wars. Certainly Java played a large part in these discussions, as did Smalltalk and C++, but the language was discussed as part of the solution, rather than as the issue itself. We believe this is a reflection of the audience, which continues to be a group very knowledgeable in object technology, and with roots spread across the academic, research, and industrial communities. Finally, the keynote delivered by Christopher Alexander served to set the tone of the conference with a thought-provoking presentation of his view of patterns—he certainly managed to evoke a great deal of discussion.

OOPSLA has often served as the forum where new products—and organizations—are introduced to the rest of the object community. This year, however, the exhibit area appeared smaller than past years, and appeared to have fewer newcomers. The one exhibit that did capture the attention of many in attendance was the new Virtual Machine (VM) technology called HotSpot, created by Animorphic Systems. It has adapted many of the ideas from the Self project in creating both the Smalltalk and Java VMs, and their most notable characteristic is speed!! They boast execution times of up to three to five times faster than existing commercial Smalltalk VMs. This is significant because it demonstrates that it is possible to create such technologies, if speed is the major factor to be considered. Speed comes

with trade-offs, though, such as execution space and a reliance on in-lining methods, but Animorphic's claim is that it sacrificed relatively little in both categories to achieve the speed. Whatever comes of the technology itself, it certainly created quite a stir. The parade of Smalltalk experts, including many lead engineers from both PP-D and IBM, to Animorphic's booth was testimony to its achievement. For more details on Animorphic's VM technology, you can visit its Web site at <http://www.animorphic.com>.

The other technology that created a great deal of interest from the Smalltalk crowd was OTI's new ENVY/QA. While we plan to include a review of ENVY/QA in an upcoming issue, we can state briefly that QA is a suite of tools designed to provide application developers with a better understanding of the quality of their Smalltalk code. It includes four main features: Code Metrics, which provides a summary of various static metrics; Code Critic, which provides feedback on the "quality" of Smalltalk code, based on a list of standard measures; Code Coverage; and Code Publisher, which generates documentation-style output of an application class library. The important feature of this tool is that it is extensible—that is, it has been designed to allow you to specify your own measures for quality assurance. We suspect it will become a mainstay in many Smalltalk developers' toolkits. You can get more details from OTI's Web site at <http://www.oti.com>.

In the end, OOPSLA appears to be reinventing itself. For the conference to stay relevant, it must continue to attract people from various backgrounds and interests, and offer them a forum to share ideas. This year's conference appeared to be taking steps toward achieving this goal. It is still the case that activities such as the poster sessions, the educators' symposium, and the Ph.D. seminar series offer a forum unmatched by other conferences.

Finally, we'd be remiss not to mention the wild evening spent at Great America theme park. Given that last month we were writing about Disneyland, and now this month about the new Top Gun roller-coaster and "The Drop Zone," object technology appears to have taken a turn toward serving crowds of thrill seekers. We offer no comment on this new relationship between thrills and objects.

Enjoy the issue.