Managing Objects

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-Digitalk 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.

```
Message Node
    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)
```

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 method, 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

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" 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 column[2] 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 June[3] and September[2] 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.