Smalltalk Chronicles

Inside this Issue

VOL 2, NO 1, MARCH 2000

Home Page
Vol 2, No 1

A Generalized Null Object Pattern

In Smalltalk, as we all know, *nil* is the distinguished object which is typically used as the initial value of all variables. It answers *true* to the message #isNil, and throws a "Does Not Understand" exception when almost any other message is sent to it.

Do all dynamic languages have a similar concept, and a similar *nil*? No, they do not. In this article, I am going to briefly compare Smalltalk's *nil* behavior to that of another dynamic language —Objective-C. Even though it has been about six years since I have done any programming in Objective-C (because I switched to Smalltalk), I have found some Objective-C techniques to be helpful and useful to my Smalltalk career. In particular, there are certain situations where I actually prefer Objective-C's concept of a *nil* over Smalltalk's. This article explores some of those situations.

If you answer *nil* in Objective-C, the *nil* that is returned is, in some respects, a lot like Smalltalk's *nil*, except that instead of the *nil* generating exceptions when you message it, it just silently eats the messages. Thus, in Objective-C, *nil* acts more like a "black hole". It is emptiness. It is nothingness. If you send anything to *nil*, *nil* is all you get back. No exceptions. No return values. Nothing.

Obviously, if we wanted this same behavior in Smalltalk, we would simply override the #doesNotUnderstand: instance message of the UndefinedObject class to just answer *self*. But would making this change be a good idea in Smalltalk? No it wouldn't—but I'm getting ahead of myself in saying that. So, let's look at it a bit closer, because in doing so it will eventually lead us to what I really *do* like!

In Objective-C, *nil* didn't always have this "message eating" behavior. Brad Cox, the inventor of Objective-C, originally gave *nil* behavior that more closely modeled Smalltalk's *nil*. That is, messaging *nil* originally generated a runtime exception, as is evidenced in his release (via StepStone Corporation) of his "ICPack 101" class library and accompanying compiler and runtime. Then, beginning with "ICPack 201" (and accompanying compiler and runtime), this behavior was changed to the current "message eating" behavior. And, NeXT Computers followed suit with their Objective-C implementation, in which they gave their *nil* a "message eating" behavior, as did the Free Software Foundation with their GNU Objective-C compiler. But it wasn't always that way.

So why did it change?

As you might guess, this change created two diverging camps among the programmers. On one side sat the programmers that preferred the original "exception throwing" behavior, and on the other side sat the programmers who preferred the new "message eating" behavior. And they each gave their best arguments to try and illustrate why the philosophy of their side was superior to the other. It was a lively and interesting debate that had no victors, other than the de-facto victor voiced by the compiler implementers themselves; namely NeXT Computers, StepStone Corp, and later the FSF, all of which chose the "message eating" behavior. But, the opinions voiced in the "pro" and "con" arguments were interesting, and especially interesting in what they both actually *agreed* upon!

Both sides agreed that the "message eating" behavior tended to create more elegant code!

But of course, the "exception throwing" side responded by saying, "*seemingly* more elegant code, yes, but potentially troublesome and unreliable", and they gave their reasons for asserting this. We will look at some of those arguments, but first I will demonstrate how the "message eating" behavior can tend to make the code more elegant looking.

Suppose, for example, that we wanted to find out the last telephone number that you dialed from your office telephone, and we wanted to save this last number into a variable called, say, 'lastNumber'. Suppose further that we wanted to save it as a string so that we could display it in a GUI widget (as well as so we could use it later). If you are the 'person' in the message sequence below, would the message sequence to accomplish this request be as follows?

```
lastNumber := person office phone lastNumberDialed asString.
widget setStringValue: lastNumber.
```

Maybe.

But then, what if you don't have an office? Or, what if you have an office, but the office doesn't have a phone? Or, what if it is new phone, and the phone has never been dialed yet? In any of these cases, using the "exception throwing" *nil* convention, an exception will be thrown, thus potentially halting the program if an exception handler hasn't been created to handle that exception.

But, what if *nil* has the "message eating" behavior? In this case, 'lastNumber' could potentially have a final value of *nil*, but the code above works just fine for this. Even passing *nil* as an argument to the #setStringValue:[1] method doesn't hurt, because the "message eating" *nil* convention is used by the widgetry as well. It doesn't matter that the argument is *nil*. Everything still works, and there's no exception thrown, and no immediately apparent strange side-effects (but we'll analyze that one some more later).

To contrast this, how then would you have to code it if *nil* has the "exception throwing" behavior? You would do it similar to this:

```
| tmp |
tmp := person office.
tmp notNil ifTrue: [tmp := tmp phone].
tmp notNil ifTrue: [tmp := tmp lastNumberDialed].
tmp notNil ifTrue: [lastNumber := tmp asString].
widget setStringValue: lastNumber.
```

Yuck...all those explicit tests for *nil* are really ugly! Of course, you could have instead wrapped the original code in an exception handler, and thus avoided the *nil* tests, something like as follows:

```
[lastNumber := person office phone lastNumberDialed asString.
widget setStringValue: lastNumber]
        on: Object messageNotUnderstoodSignal do: [].
```

This looks a bit simpler than the previous example, but even this example contrasts poorly to the first example. The first example of these three is much simpler! You just "do it", without worrying about exceptions, exception handlers, or explicit tests.

But, is this kind of code common? With the "exception throwing" *nil*, do we really end up typically testing for *nil* like this, or else setting up exception handlers like this?

Yes, it is common. While the above example was contrived, let's look at a real-life example, from the #objectWantingControl method of the *VisualPart* class of VisualWorks. The *VisualPart* class is a superclass of the *View* class, and views in VisualWorks are coded to generally expect to have a collaborating controller object for processing user input (mouse and keyboard events). Thus, the #objectWantingControl method is a method of the view object, and it asks it's controller if it wants the user focus. If it does, #objectWantingControl answers *self*, otherwise it answers *nil*. If *nil* is answered, then the view is considered to be read only, and will not process user input. The actual implementation of #objectWantingControl is as follows:

```
objectWantingControl


    | ctrl |
    ctrl := self getController.
    ctrl isNil ifTrue: [^nil].
    " Trap errors occurring while searching for
```

```
        the object wanting control. "
        ^Object errorSignal
              handle: [:ex |
                    Controller badControllerSignal
                    raiseErrorString:
                          'Bad controller in objectWantingControl']
              do: [ctrl isControlWanted ifTrue: [self] ifFalse: [nil]]
```

Notice that this method has both an explicit #isNil test, as well as an exception handler. How would this method instead be written if the system had a "message eating" *nil* throughout? While there are a several variations of possibilities, including at least one variation that is shorter (but not necessarily clearer), we would probably write it as follows:

```
objectWantingControl
self getController isControlWanted ifTrue: [^self].
^nil
```

Notice how much simpler it suddenly became. The programmer's intentions are much clearer. No #isNil checks, no exceptions, no extra code to confuse the issue.

Furthermore, with the "exception throwing" *nil*, even when the code is written to avoid #isNil tests and/or exception handlers, the coding style is usually altered in other ways to compensate. And, invariably, these style alterations don't produce as elegant of code as if you had a "message eating" *nil*.

"Message eating" *nil* creates simpler, more elegant code. This was almost the unanimous opinion of both camps of the Objective-C debate on this. But of course, the "exception throwing" camp argued that this "simpler" code was also potentially more troublesome, and sometimes even buggy. And, they gave examples to illustrate. But their examples also all seemed to fall into one of two arguments.

The first argument boiled down to the observation that, with a "message eating" *nil*, if a message sequence produces *nil* as the final result, it is more difficult to determine exactly where the breakdown occurred. In other words, what was the message that produced the first *nil*?

And, the response to this argument was: the programmer typically doesn't care what message produced the first *nil*, and even if he did, he would explicitly test for it.

And of course, the response to this response was: the programmer *should* care, but typically *won't* care, therefore the "message eating" *nil* is a feature which promotes bad programming habits.

And of course, this in turn illicited a response that essentially just disagreed with their conclusions and challenged their statements, such as *why* the programmer should care, etc.

And so the debate raged. But none-the-less, both sides seemed to admit that the "message eating" *nil* tended to create simpler, more elegant looking code. And, the existing code base tended to substantiate this conclusion. And simple code is good code, as long as it is also accurate code.

So, with a "message eating" *nil*, is the resulting code accurate? Or does the "message eating" *nil* tend to introduce subtle bugs? To this question, the "exception throwing" crowd said it introduces subtle bugs, and they gave specific examples. Interestingly enough, all of their examples that I looked at were with statically declared variables, and those examples typically illustrated the platform dependent idiosyncracies that developed when a *nil* was coerced into a static type. One specific example was illustrated via the following code snippet (which I have modified to conform to Smalltalk syntax instead of Objective-C syntax):

```
value := widget floatValue
```

In this example, if 'value' is statically declared to be a variable of type *float* (floats are not objects in Objective-C, but are instead native data types), and the #floatValue[2] message returns *nil* instead of a valid float number, then after the assignment has completed, 'value' will equal zero on the Motorola M68K family of processors, but on the Intel processor family, it ends up being a non-zero value, because of the peculiarities of implicit casting of a *nil* to a native *float* datatype.

This is clearly an undesirable result, and can lead to subtle bugs.

But, while those examples might be relevant for Objective-C, they are totally irrelevant for Smalltalk. There is no static declaration of variable types in Smalltalk, nor are there native data types (non-objects). It's a non-issue in Smalltalk.

So, should the semantics of *nil* in Smalltalk be changed such that it eats messages? This is easy to do by changing #doesNotUnderstand: to just answer *self*. Should we do it?

No, I don't think so. There has been too much code already written that is now expecting *nil* to throw exceptions. To change the semantics of *nil* from "exception throwing" to "message eating" at this time would likely break a large body of that code. It could be a very painful change, indeed.

Furthermore, even in Smalltalk, the first objection to a "message eating" *nil* still stands; to whit, in a given message sequence whose final value is *nil*, it is difficult to determine what object first returned the *nil*. While it is purely a subjective opinion as to how important that objection really is, I don't know how anyone could not agree that it is indeed a valid objection. Minor perhaps (and perhaps not), but valid.

So, instead of modifying Smalltalk's *nil*, let's now briefly look at an alternative, that of sending back a specialized Null object that has message-eating semantics. The first public document that I am aware of that explored this alternative is Bobby Woolf's excellent white paper, "The Null Object Pattern"[3], although earlier works likely do exist. In that paper (which is now about five years old—almost an eternity in computer time), he also uses the *VisualPart* example from above to illustrate his pattern. In fact, that is precisely why I also chose to illustrate the results of using a "message eating" null via this same *VisualPart* example. That way, I could keep things simple and consistent, without introducing too much additional code for all of the illustrations.

The "Null Object Pattern" essentially recommends the creation of a "do nothing" null object which implements the same protocol as the original object, but does nothing in response to that protocol. For the *VisualPart* example above, this pattern requires the creation of a class called *NoController* which implements the protocol of a controller, but does nothing in response to it.

Doing nothing, however, means something special to a controller. For example, the *NoController* is expected to answer *false* to the #isControlWanted message. Why is this important? Because clients of *NoController* expect a boolean result to the #isControlWanted message, and they might in turn try sending #ifTrue: or #ifFalse: to that result, and only booleans (and perhaps "message eating" *nil*s) respond to #ifTrue: and #ifFalse. The *NoController* has to return something that will respond to these boolean messages, or else the *NoController* is not going to be plug-compatible with a real controller.

But, suppose we instead had #isControlWanted return a "message eating" *nil*? Or better yet, what if the #getController method of the *VisualPart* returned a "message eating" *nil*? I believe everything would still "just work", and that this also would be a simple way to generalize the "Null Object Pattern" of Woolf's paper.

Interestingly enough, in Woolf's paper, he describes an advantage of the "Null Object Pattern" by saying it...

> *...simplifies client code. Clients can treat real collaborators and null collaborators uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a real or a null collaborator. This simplifies client code, because it avoids having to write special testing code to handle the null collaborator.*

This testimony dovetails nicely with the NeXTSTEP community's assertion that the "message eating" *nil* behavior of Objective-C appears to simplify code, as I have already demonstrated.

But, to implement the "Null Object Pattern", do we create a *NoController* class, and a *NoOffice*, and a *NoPhone*, and a *NoLastNumberDialed* class? Where does it end? Indeed, this potential class explosion of the "Null Object Pattern" is also mentioned by Woolf, as follows:

> *[One of] the disadvantages of the Null Object pattern [is]...class explosion. The pattern can necessitate creating a new NullObject class for every new AbstractObject class.*

A "message eating" *nil* would avoid this class explosion, as it is protocol-general instead of protocol-specific. I personally feel that this difference is even a bit reminiscent of the static typing vs. dynamic typing differences (and ensuing debates), as the following chart illustrates:

Static Typing vs. Dynamic Typing	**Null Objects vs. Message Eating Nil**
Should we allow any type of object to be handled by (assigned to) this variable, or only objects of a specific type?	Should we allow any type of message to be handled by (sent to) this object, or only messages of a specific type (protocol)?

I make no secret that I prefer dynamic typing over static typing. And, I also believe that often a general "message eating" *nil* is more desirable than the more specific "Null Object Pattern", provided of course that the "message eating" *nil* is implemented correctly. What follows is my implementation of a "message eating" *nil*, which I call a *null*, which is an instance of my class *Null*.

Recall that the first objection against the *null* was that in a given message sequence whose final value is *null*, it is difficult to determine what object first returned the *null*. How do we handle that objection?

Simple. Just ask it.

The Null class should have an *originator* instance variable that records who originally invoked the 'Null new', as well as a *sentFromMethod* instance variable that records from what method of the originator the 'Null new' was invoked.

But does that mean that the creator of the null must now tell the null so that the null can tell you? That sounds like a lot of work! And, what if someone forgets those extra steps?

Simple. Don't require the extra steps. Anybody should be able to send 'Null new', and the Null class itself should be able to figure this information out.

But that is not possible to do unless the Null class can somehow automatically determine who is calling one of it's instance creation methods. In other words, we need to detect who the sender of the message is. How do we do that? It is not part of standard Smalltalk!

Well, here is how to do it in VisualWorks:

```
Object>>sender
	^thisContext sender sender receiver

Object>>sentFromMethod
	^thisContext sender sender selector
```

And here is how to do it in VisualAge:

```
Object>>sender
	| sender |
	sender := Processor
		activeProcess stackAtFrame: 2 offset: -3.
	(sender isKindOf: BlockContext)
		ifTrue:
			[sender := sender methodContext instVarAt: 2].
	^sender

Object>>sentFromMethod
	^Processor activeProcess methodAtFrame: 2
```

And, here is how to do it in GemStone:

```
Object>>sender
	^ (GsProcess _frameContentsAt: 3) notNil
```

```
                ifTrue: [frame at: 10]
                ifFalse: [nil].

Object>>sentFromMethod
      ^ (GsProcess _frameContentsAt: 3) notNil
            ifTrue: [(frame at: 1) selector]
            ifFalse: [nil]
```

Now, the Null class method #new just does the following:

```
Null class>>new
      | tmp |
      tmp := super new.
      tmp originator: self sender.
      tmp fromMethod: self sentFromMethod.
      ^tmp
```

Now, in your other code, anytime you want to return a *nil* that also has message-eating semantics (which I call a null), you use '^Null new' from your code instead of '^nil'. Then, your caller can easily discover the originator of the null if it wishes to simply by asking.

If you are concerned about the potential proliferation of nulls with such a scheme, another trick you might try is to create a default null using a 'Default' class variable:

```
Null class>>default
      Default == nil
            ifTrue:
                  [Default := super new initialize.
                  Default originator: Null.
                  Default fromMethod: #default].
      ^Default
```

The default null can then later be accessed via 'Null default' instead of 'Null new'. I actually use this quite often for automatic instance variable initialization in my abstract DomainModel class, which is the superclass of all of my domain objects:

```
 DomainObject>>initialize
       1 to: self class instSize
             do: [:ea |
                (self instVarAt: ea) isNil
                   ifTrue:
                         [self instVarAt: ea put: Null default]].
       ^self
```

I have found that such a scheme does indeed simplify the domain logic, just as this article indicates that it should. In fact, sometimes it has *dramatically* simplified things. And, I have never had any problems with this scheme, as long as I have limited its use to the domain layer only. I have, however, had problems trying to integrate some of these ideas into the GUI layer, and decided long ago that it was a bad idea in that layer.

My own implementation of the Null class was originally written in VisualAge, and was originally part of a much larger domain-specific class library. This class library originally tried a number of ideas on an experimental basis, to see if problems resulted from their use. The use of the Null pattern described in this article was one of those experimental ideas. Even though it is actually a small idea, it's widespread use in the domain layer was encouraged from my previous Objective-C experience, but I still didn't know if I would run into other subtle issues while using it in Smalltalk. But I feel comfortable with it now in the domain layer (but not in the GUI layer).

Some time after creating the class library I mentioned above, the entire class library was ported to GemStone, and then finally the entire class library was moved to VisualWorks. A filein of the VisualWorks implementation of the Null class follows. Email me at nevinop@xmission.com if you want either the VisualAge or GemStone versions, and I'll try to dig them out.

If you instead decide to create your own Null class in VisualAge, another thing to realize is that #isNil is inlined in VA (but not in VW). Thus, something like 'Null new isNil' will always answer

false in VA, even though your Null>>isNil method explicitly answers *true*. Hence, in your domain code, with VA you probably want to create an #isNull method and use that instead of #isNil. That is what I originally did, and that convention carried to the GemStone version, but I have since broken that convention in the VisualWorks version.

Nevin Pratt has been Smalltalking since 1994, mostly with VisualWorks and GemStone. Before that, he was an avid NeXTSTEP fan, with three years Objective-C experience, and at one time the President of the Salt Lake NeXT User Group (SLUG). Nevin is currently consulting with Cincom's VisualWorks. Nevin can be reached at: (801) 558-9992, or E-mail at: nevinop@xmission.com.

```
'From VisualWorksÂ® NonCommercial,Release 5i.1 of January 24, 2000
on March 16, 2000 at 10:08:00 pm'!

Smalltalk defineClass: #Null
      superclass: #{Core.Object}
      indexedType: #none
      private: false
      instanceVariableNames: 'originator    fromMethod '
      classInstanceVariableNames:    ''
      imports: ''
      category: 'PawnDomain'!

Smalltalk.Null defineStatic:    #Default
      private: true
      constant: false
      category: 'As yet unclassified'
      initializer: nil!

"-- -- -- -- -- --    -- -- -- -- -- -- -- -- -- -- -- -- "!

!Null class methodsFor:    'instance creation'!

default
      Default == nil
            ifTrue:
                  [Default := super new initialize.
                  Default originator: Null.
                  Default fromMethod: #default].

      ^Default!

new

      | tmp |
      tmp := super new.
      tmp originator: self sender.
      tmp fromMethod: self sentFromMethod.
      ^tmp! !

!Null class methodsFor:    'initialize release'!

initialize
      Default := nil.! !

"-- -- -- -- -- --    -- -- -- -- -- -- -- -- -- -- -- -- "!

!Null methodsFor: 'comparing'!

= anObject
      ^anObject isNil!

hash
      "Make the hash the same as a nil. Otherwise, each instance
of a null will have a different hash"
```

```
      ^nil hash! !

!Null methodsFor: 'printing'!

asString
      ^''!

printOn: aStream
      "Print an representation of the receiver on the
      Stream, aStream."

      self == Null default
            ifTrue: [aStream nextPutAll:    'defaultNull']
            ifFalse: [aStream nextPutAll:    'null']! !

!Null methodsFor: 'forwarding'!

doesNotUnderstand: aMessage
      ^self!

respondsTo: aSymbol
      ^true! !

!Null methodsFor: 'accessing'!

fromMethod
      ^fromMethod!

fromMethod: aSymbol
      ^fromMethod := aSymbol!

originator
      ^originator!

originator: aClass
      ^originator := aClass! !

!Null methodsFor: 'testing'!

isDefaultNull
      ^self == self class default!

isNil
      "Implemented because on some systems, #isNil actually
is a message send."

      ^true!

notNil
      "Implemented becauseon some systems, #notNil actually
is a message send."

      ^false! !

!Null methodsFor: 'compatibility'!

mustBeBoolean
      "Override Object's implementation because 'ifFalse:' and
'ifTrue:' are inlined in VA and VW."

      ^false! !

Null initialize!
```

---

1. As a sidebar, one could also argue about the appropriateness of a #setStringValue: method in this example, and its implied limitation of only setting, or showing, strings, rather than having perhaps a more generic #show: method that can show other types as

well. To this, I have three things to say:

- first, consider the commonly used #show: method of the Transcript class in Smalltalk, and the argument type it expects (strings)
- second, #setStringValue: is the actual method name used for TextField widgets in NeXTSTEP
- third, who cares, this is just a contrived example anyway.

2. #floatValue also is an actual message implemented by TextFields under NeXTSTEP, just as the #setStringValue: of the earlier code snippets is.
3. Published in *Pattern Languages of Program Design. Addison-Wesley*, James Coplien and Douglas Schmidt (editors). Reading, MA, 1995; http://www.awl.com/cseng/titles/0-201-60734-4.