

Re: Passing by reference

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2006-11/msg00008.html>

- *From:* "Matt McGill" <matt.mcgill@xxxxxxxxx>
 - *Date:* 31 Oct 2006 19:29:16 -0800
-

H. S. Lahman wrote:

Responding to Veeneman...

Is passing a method argument by reference inherently dangerous?

It depends upon the reason one is passing the reference. One can't instantiate all relationships via a constructor so there will always be a need for passing an object reference to instantiate conditional relationships of relationships where participation varies depending on context. So passing an object reference solely to instantiate a relationship is not Bad simply because it is sometimes unavoidable.

However, passing object references is generally not a good idea in any normal solution collaboration.

<snip detailed explanation of the above>

I've been mostly lurking here for some time, and am finally beginning to grok what you're getting at here (this being the third or fourth reply of yours on this topic which I have read). I'll have to re-evaluate some of the code I've recently written to see how many relationships are being instantiated via the passing of a reference before it really sinks in, but the reasoning does make sense to me.

But I'm still missing some puzzle pieces.

In your list of the degrees of coupling, you mentioned data-by-reference arguments as being distinct from object-by-reference arguments. What distinguishes the two? I am by no means an OO expert, but when I read about Alan Kay's original vision of Smalltalk, for example, I don't see where the concept of 'data' fits into the picture. My understanding is that even the integers were envisioned as instances of objects, like tiny computers, sending and receiving messages. That the basic data types in Java exist was, I thought, essentially a performance-driven compromise. Or do you mean data as in 'value type,'

Re: Passing by reference

like Java's String, Integer, etc.? If so, what allows us to treat these objects any differently from other objects in our design? (I don't mean to suggest that we *should* treat them the same, I just don't understand why an immutable object is all *that* different from a mutable one)

I would also be very interested to hear more about 'knowledge accessors.' What sorts of characteristics and/or constraints distinguish a knowledge accessor from a more typical method? Living in the Java world, I naturally think of 'getter' methods as 'knowledge accessors,' and have lately come to the conclusion that the presence of getter methods (the way they are typically used) is in most cases violating encapsulation and a symptom of misplaced responsibilities. In my current project at work, I have only found the need for 'getter' methods on the lowest-level domain objects in the problem space, and these objects are so far little more than data containers with some basic validation and consistency checks. The presence of the 'getters' and the lack of any substance in these low-level classes makes me think I must have committed some sort of design sin.

Are my logic-starved low-level objects more like data to be passed around than objects with collaborations, or am I simply missing some point entirely?

Any thoughts would be welcome,
-Matt McGill

.