

Re: Lahman, how ya doing?

Re: Lahman, how ya doing?

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2005-05/msg00069.html>

- *From:* "H. S. Lahman" <h.lahman@xxxxxxxxxxx>
 - *Date:* Sun, 08 May 2005 19:35:44 GMT
-

Responding to Hansen...

Hopefully it is a good book. B-) You might check out the Books section of my blog for some suggestions. Bruce Douglass has several books on the OOA/D-based use of event-based processing in R-T/E. One also has to be careful about books with 'UML' or a specific OOPL in the title; they are good at describing how to express your design in the UML or OOPL syntax but they tend to be short on advice about coming up with a good design in the first place.

Applying UML and Patterns, 2nd ed, by Craig Larman. The author has made clear in an early chapter that the book is about OOA, and what UML is introduced will serve that purpose.

I've read some of his other books but not that one, so I can't comment directly. His other books have been well-written and clear for their stated objectives.

I appreciate the help, though.

That's what these forums are for. Being retired after 40+ years in the business I have an oversupply of opinions and I need to get rid of them someplace.

And you're still paying attention to me. I get the feeling

Re: Lahman, how ya doing?

Re: Lahman, how ya doing?

that the other readers of the forum see the thread now and then and think "Are they still going at it?"

Few people, as lurkers, follow the threads I get involved in because of the message lengths. It is a busy forum so it is tough to follow threads with long messages unless one has a vested interest in the specific thread.

Since I usually only get involved in threads with complex OOA/D issues, I think it is necessary to make lengthy explanations in that case for two reasons. One is that the OP usually hasn't a lot of formal OOA/D training or the issues wouldn't have come up in the first place, so there are gaps in the knowledge and it is hard to know where they are. Disconnects are common enough even when the bases are covethought of, but have been finding no reason to try to do.

If you meant you are finding no reason to have methods return values, then ignore this next bit. If you meant you are finding to no reason to try to not return value from behavior methods, then read on...

The reason is hierarchical dependencies. One can argue that the entire OO paradigm is directed at eliminating exactly these sorts of hierarchical implementation dependencies.

```
AClass::method1 (x)
  tmp = x + 3
  tmp = myBClass.doIt(tmp)
  this.attr1 = tmp * 5
```

This is a poorly formed OO method because it cannot be specified properly without also specifying exactly what BClass::doIt does. Nor can it be unit tested without having an implementation of BClass::doIt available. (One can stub return values in the test harness for a given 'x' test value, but that is just self-delusion; one is testing the test harness rather than the method.) In effect, AClass::method1 is a higher level node in a hierarchical functional decomposition tree that is coordinated lower level procedures. In such trees the the lower level procedures are literally extensions of the higher level procedure.

That's not a problem so long as the requirements don't change or one never invokes the higher level procedure in multiple contexts (i.e., it is called from exactly one parent higher level procedure). The reality, though, is that one reuses such higher level nodes because it is convenient to do so. That turns the tree into a lattice and one has the legendary Spaghetti Code. If one changes a lower level procedure to accommodate a change in requirements for one context of invocation of a higher level procedure, that change affects all contexts of invocation of that procedure. If the other contexts are not affected by the requirements change (i.e., they still want to do things the way the old

Re: Lahman, how ya doing?

requirements did them), then one has a problem and the entire tree has to be reformulated.

One resolves this the OO way by reorganizing the responsibilities:

```
AClass::method1 (x)
  tmp = x + 3
  mtBClass.doIt (tmp)
```

```
AClass::method2 (x)
  this.attr1 = x * 5
```

Now BClass::doIt invokes AClass::method2. Now both of the AClass methods can be fully specified and tested without the presence of BClass::doIt. All one needs to demonstrate is that a message with the right value was sent to BClass::doIt (e.g., BClass::doIt was called with the right argument). What BClass does with that value is not the concern of Calais.

The daisy-chaining of the messages ensures that things get done in the right sequence (add 3, then modify result, then multiply result by 5). The requirements are still specified and validated the same way; they have just been allocated differently.

Note that this view of specification depends on several things methodologically: separation of message and method, peer-to-peer collaboration, responsibilities that represent intrinsic entity responsibilities, context-independence, implementation hiding, and a flexible view of logical indivisibility. IOW, all this good OO methodological stuff is designed to play together to eliminate hierarchical dependencies.

The first, though, that behavior messages very rarely carry data, that one accesses it directly from the source when it is needed, I need more on that. For starters, I thought behavior messages **do** carry data, or that they can. E.g. when a mouse down event is generated, you're going to want to know where it went down. But in the case we're discussing, I get the idea that if Controller needs to know an elapsed time it should ask Timer for the time, keep its own record of the previous time, and do its own calc