



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 Others argue that, although many of them represent a throwback to outdated ways of thinking, they can be very useful under the right circumstances.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.