Alan Knight

# Principles of OO design:
## or, everything I needed to know in life, I learned from Dilbert*

EVERYONE KNOWS THAT objects and object-oriented (OO) design are the hottest things since sliced bread (and, of course, slices of bread are objects). The problem is that it's hard to agree on what exactly they are. There have been many attempts to define principles of OO design or coding, with varying degrees of success. In my opinion, most of them suffer from two flaws. First, they don't tell me enough about how to code. Reading a definition of "polymorphism" doesn't tell me how to exploit it in my programs. Second, and more important, is that they're dull. Even if the definition of polymorphism did tell me how to code, it's hard to stay awake long enough to finish reading it.

Therefore, I modestly present some of my own principles of OO-ness, which I hope address both of these flaws. Furthermore, I believe that these principles relate well to the corporate environments that have seen so much Smalltalk use recently.

*If you must accept a responsibility, keep it as vague as possible.*

### NEVER DO ANY WORK THAT YOU CAN GET SOMEONE ELSE TO DO FOR YOU

This is always good advice, but it's particularly applicable in OO. In fact, I consider it the fundamental principle of OO. As an object, my responsibilities are very clearly defined, and so are those of my co-workers. If something is (or ought to be) one of their responsibilities, then I shouldn't be trying to do that work myself.

Let's look at a concrete example:

```
total := 0
aPlant billings do: [:each|
    (each status == #paid and: [each date > startDate])
        ifTrue: [total := total + each amount]].
```

versus

```
total := aPlant totalBillingsPaidSince: startDate.
```

Alan Knight has had great success avoiding responsibility with The Object People, 885 Meadowlands Dr. East, Ottawa, Ontario, K2C 3N2. He can be reached at 613.225.8812 or by email as knight@acm.org.

In the first case, we're asking the plant for all of its billings, figuring out for ourselves which ones qualify, and computing the total. That's a lot of work, and almost none of it is our job. Far better to use the second option, where we simply ask for something to be done and get a result back. In real-world terms, the first example is like the following conversation:

"Excuse me, Smithers. I need to know the total bills that have been paid so far this quarter. No, don't trouble yourself. If you'll just lend me the key to your filing cabinet I can go through all the records myself. I'm not familiar with your new filing system, but I'm sure I can figure it out. I'll try not to make too much of a mess."

Smithers actually understands his filing system, so he can probably do the work faster than we can, and he's much less likely to mess everything up. In attempting to do his job for him, we're just making things worse. Things will get worse when he switches over to that new filing system next week. We'd be far better off acting like a stereotypical tyrant boss.

> "SMITHERS! I need the total bills that have been paid since the beginning of the quarter. No, I'm not interested in the petty details of your filing system. I want the total, and I'll expect it on my desk within the next half millisecond."

Let's look at a simpler example, which is all too common.

```
somebody clients add: Client new.
```

versus

```
somebody addClient: Client new.
```

There's always a temptation to choose the first option, because it saves writing a couple of methods that do nothing but add and delete on the other class. But you know it's wrong. You're trying to do somebody's work for

* Dilbert is a trademark of United Feature Syndicate.

them, and ultimately it's only going to cause problems. Writing those extra methods keeps the responsibility where it belongs and will make the code cleaner in the long run.

This principle is close to the more conventional idea of "encapsulation", but I like to think it makes the idea somewhat clearer. I often see people who are happily manipulating the internal state of another object, but think it's OK because they're doing it all through messages. Encapsulation is not just about accessing state, it's about responsibilities. Responsibility is about who gets stuck doing the real work.

*Never do any work that you can get someone else to do for you.*

## AVOID RESPONSIBILITY

If responsibilities are about getting stuck with work, it's important to avoid them. This has some important corollaries:

- If you must accept a responsibility, keep it as vague as possible.
- For any responsibility you accept, try to pass the real work off to somebody else.

Our first principle tells us to take advantage of other objects when writing code. We also have to avoid being taken advantage of. Any time I (as an object) am tempted to accept a responsibility, I should ask myself, "Is this really my job?" and "Can't I get someone else to do this?"

If I do accept a responsibility, it's important to keep it as vague as possible. If I'm lucky, this vagueness will help me avoid doing the work later. Even if I must do the work, it may allow me to take some shortcuts without anybody else noticing.

For example, I've seen CRCs with responsibilities like:

Maintain a collection of the whosits to be framified

This is much too specific. My job isn't to maintain a collection, it's to be able to report, when necessary, which whosits need framification. That may be implemented by maintaining a collection, or by asking one or more other objects for their collection(s), it may be hard-coded, or computed dynamically as `Whosit allInstances select:`. Regardless of which option I choose, there shouldn't be any impact on my responsibilities.

My preference for phrasing a responsibility of this kind is:

Know which ...

but I'm flexible as long as the phrasing is suitably vague. I'd probably be even happier with

"Be able to report which ..."

Carried to the extreme, it seems this could lead to the situation where everyone passes information around and nothing ever gets done. Exactly. Object bureaucracy at it's finest.

Seriously, a good OO system can actually approach this state. Each object will do a seemingly insignificant amount of work but somehow they add up to something much larger. You can trace through the system, seeking the place where a certain calculation happens, only to realize that the calculation is finished and you just didn't notice it happening.

## POSTPONE DECISIONS

The great virtue of software is *flexibility*. One way we achieve flexibility is through late binding. We most often discuss late binding between a method name and the method it invokes, but it's also important in other contexts. When faced with a decision, we can gain flexibility by postponing it. The remaining code just needs to be made flexible enough to deal with any of the possible outcomes.

The ideal is when we can avoid making the decision at all, leaving it up to someone else (the end user, other objects). For example, consider the question of how to implement dictionaries. The standard thing to do is use a hash table. That works well for medium-sized collections, but it's a waste of space and effort for very small collections. For very large collections, it may also be wasteful, particularly if the number of elements exceeds the resolution of our hash function. We must make a decision here, so we'd like to postpone it or pass it off to someone else.

Some implementations of the collection classes do precisely this. The collections transfer much of their behavior to an implementation collection that actually does the work. Depending on the size, the nature of that collection can change. In VisualAge 2.0, small dictionaries were stored as arrays because the overhead of hashing was more than the cost of a linear search. Larger dictionaries could be represented as either normal or bucketed hash tables. This seems to have disappeared in 3.0, so I suppose the overhead of this mechanism became more than the cost of using a single representation. Visual Smalltalk also has dictionaries that are capable of switching between normal and bucketed hash tables.

Be careful in applying this principle because it's possible to take it too far. Decisions aren't just sources of problems, they give us the power to solve problems. Because we cannot solve all the problems of the world at once, we make the decision to limit ourselves, and we make assumptions about the problems we'll be given. The problem arises when our decisions were poor, or our assumptions don't hold any more. The trick is to make

*continued on page 28*

## THE BEST OF COMP.LANG.SMALLTALK

*continued from page 14*

enough decisions to be able to work, but few enough that our code doesn't become brittle. That's one of the things that makes software difficult.

Passing off decisions to another object is often referred to as using policy or strategy objects. This is discussed in DESIGN PATTERNS[1] as the Strategy pattern.

Other related ideas are "Open Implementations," which can allow important decisions to be postponed so far that even the end user of the module can control them. I can't do justice to this topic here, but there's a web page available at http://www.xerox.com/PARC/spl/eca/oi.html

Because web pages change so rapidly, I'll also mention that I found it using the search terms *open implementation* and *Gregor Kiczales* (the project leader).

### POSTSCRIPT

Although there is a significant element of humor in these principles, I do take them quite seriously and urge you to do the same. They illustrate some very important aspects of OO design and coding. I've even come up with enough of them to fill another column, so the next issue will continue this theme.

### Reference

1. Gamma, E. et al. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1994.

## MANAGING OBJECTS

*continued from page 17*

person can cause damage more quickly on a Smalltalk project than they can on a traditional project, and corporate cultural checks that normally help such people, such as peer reviews, management one-on-one meetings, and performance reviews, are tuned to the slower beat of the traditional project.

Beginning a Smalltalk project offers the opportunity for a "behavioral context switch," in which old patterns can be broken. By catching behavioral difficulties early, you can keep them from becoming established patterns. Once behavioral patterns are established, their impact on productivity must be carefully monitored and humanely dealt with.

### References

1. Kroeger, O. and J.M. Thuesen. TYPE TALK AT WORK, HOW THE 16 PERSONALITY TYPES DETERMINE YOUR SUCCESS ON THE JOB, Tilden Press, New York, 1992. [*This book concentrates on applying Jungian personality type theory in the workplace, and is much more approachable than defining works on the topic.*]
2. Bramson, R.M. COPING WITH DIFFICULT PEOPLE, Anchor Press/Doubleday, Garden City, NY, 1981.
3. Brooks, Jr., FP. THE MYTHICAL MAN-MONTH (20th anniversary ed.), Addison-Wesley, Reading, MA, 1995. [*A wonderful classic.*