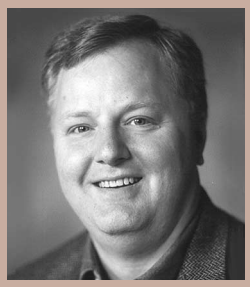
Editors: Dave Thomas and Andy Hunt ■ The Pragmatic Programmers
dave@pragmaticprogrammer.com ■ andy@pragmaticprogrammer.com

OO in One Sentence:

Keep It DRY, Shy, and Tell the Other Guy

Andy Hunt and Dave Thomas

Some people feel that "getting" object-oriented programming is a difficult, time-consuming process. But does it need to be that hard? And is the difficulty even specific to OO programming? Many of the cornerstones of OO programming benefit other programming paradigms as well. Even if you're writing shell scripts or batch files, you can use these techniques to great advantage.



What's good code?

There are many aspects to writing good code, but most of these hinge on a single underlying quality: flexibility. Flexibility means that you can change the code easily, adapt it to new and revised circumstances, and use it in contexts other than those originally intended.

Why does code need to be flexible? It's largely because of us humans. We misunderstand communications (be they written or oral). Requirements change. We build the right thing the wrong way, or if we manage to build something the right way, it turns out to be the wrong thing by the time we're done.

Despite our fondest wishes, we'll never get it right the first time. Our mistakes lie in continually assuming that we can and in searching for salvation in new programming languages, better processes, or new IDEs. Instead, we need to realize that software must be soft: it has to be easy to change because it *will* change despite our misguided efforts otherwise.

Capers Jones, in his book *Software Assessments, Benchmarks, and Best Practices* (Addison-Wesley, 2000), showed that requirements change at a rate of about 2 percent per month (which really starts to add up after a year or two). But the problem with changes to projects is by no means limited to the common scapegoat of "requirements," nor is it limited to the software industry.

According to a study of building construction in the UK ("Rethinking Construction," Construction Task Force report to the Deputy Prime Minister, 1998), some 30 percent of rework isn't due to requirements changes at all. It's due to mistakes: plain, old human errors, such as cutting a joist 2 inches too short. Using the wrong kind of nail. Cutting the window in the wrong wall. It's just human nature that we'll get some things wrong, so what differentiates software quality is how well—and how quickly—we can fix or change something. Flexible code can be changed easily and cheaply, regardless of whether the change is necessitated by volatile requirements or our own misunderstanding.

Most of the important lessons to be learned about object technology—how to avoid many

0740-7459/04/$20.00 © 2004 IEEE

```
getOrder().getCustomer().getAddress().getState()
```

Figure 1. Example of the "train wreck" style of dynamic coupling.

common mistakes and keep code flexible—can be summed up in one sentence: "Keep it DRY, keep it shy, and tell the other guy." Let's take a look at what that means and how you can apply these lessons to all good code, not just OO code.

Keep it DRY

Our DRY (Don't Repeat Yourself) principle deals with knowledge representation in programs (see *The Pragmatic Programmer,* Addison-Wesley, 2000). It's a powerful idea that states:

> *Every piece of knowledge must have a single, unambiguous, and authoritative representation within a system.*

In other words, you should represent any idea, any scrap of knowledge, in a system in just one place. You might end up with physical copies of code for various reasons (middleware and database products could impose this restriction, for instance). But only one of these physical representations is the authoritative source. Ideally, you'd be able to automatically generate or produce the nonauthoritative sources from the single authoritative source.

Why go to all this trouble? So that when a code change is required, you only have to make it in one place. Anything else is a recipe for disaster, introducing inconsistencies and potentially hard-to-find bugs.

DRY applies to code but also to every other part of the system and to developers' daily lives—build processes, documentation, database schema, code reviews, and so on.

Keep it shy

The best code is very shy. Like a four-year old hiding behind a mother's skirt, code shouldn't reveal too much of itself and shouldn't be too nosy into others affairs.

But you might find that your shy code grows up too fast, shedding its demure shyness in favor of wild promiscuity. When code isn't shy, you'll get unwanted coupling; these axes of ill-advised coupling include *static, dynamic, domain,* and *temporal.*

Static coupling exists when a piece of code requires another piece of code to compile. This isn't a bad or evil thing—far from it. Even the canonical "Hello World" program requires the standard I/O library and such. But you have to be aware of accidentally dragging in more than you need.

Inheritance is infamous for dragging in a lot of excess baggage. Often it's more efficient, more flexible, and safer to use delegation instead of inheritance (which should be reserved for true *is-a* relationships, not *has-a* or *uses-a*). Shy people don't talk to strangers, and shy code should be equally wary of other code that wants to come along for the ride.

Dynamic coupling occurs when a piece of code uses another piece of code at runtime. This can get seriously out of hand using a style we call the "train wreck" (see Figure 1).

To get the state for an order, this code has to have detailed knowledge of an address, a customer, and an order—and rely on these three components' implied hierarchal structure. If anything in that mix changes, we're in trouble; this code will break. Shy code only talks to code it deals with directly and doesn't daisy-chain through to strangers as in the previous example.

Always plan on writing concurrent code because the odds are good that it will end up that way anyhow, and you'll get a better design as a fringe benefit.

Domain coupling takes place when business rules and policies become embedded in code. Again, that's not necessarily a bad thing unless mirroring real-world changes becomes difficult. If the real world is particularly volatile, put the business rules in metadata, either in a database or property file. Keep the code shy by not being too nosy about the details: the code can act as an engine for the business rules. The rules can change at whim, and the code will merrily process them without any change to the code itself. Small interpreters work well for this (really small—like a case statement, not a large yacc/lex endeavor).

Temporal coupling appears when you have a dependency on time—either on things that must occur in a certain order, at a certain time, by a certain time, or worse, at the same time. Always plan on writing concurrent code because the odds are good that it will end up that way anyhow, and you'll get a better design as a fringe benefit. Your code shouldn't care about what else might be happening at the same time; it should just work regardless.

Code shouldn't be nosy. I used to have a neighbor who would peer hawklike over her kitchen sink out the front window and keep track of every neighbor's comings and goings. Her life hung at the mercy of every whim of the entire neighborhood; it wasn't a healthy position for her to be in, and it isn't a healthy position for your code either. A big part of not being nosy lies in our next item.

Tell the other guy

One of our favorite OO principles is "Tell, Don't Ask" (see *IEEE Software*, Jan./Feb. 2003, p. 10).

To recap briefly: as an industry, we've come to think of software in terms of function calls. Even in OO systems, we

view an object's behavioral interface as a set of function calls. That's really not a helpful metaphor. Instead of calling software a function, view it as sending a message.

"Sending a message" to an object conveys an air of apathy. I've just sent you an order (or a request), and I don't really care who or what you are or (especially) how you do it. Just get it done. This service-oriented, operation-centric viewpoint is critical to good code. Apathy toward the details, in this case, is just the right approach. You tell an object what to do; you don't ask it for data (too many details) and attempt to do the work yourself.

By "telling the other guy" in this way, you ensure an imperative coding style that keeps your code from becoming too nosy and from getting involved in details that it shouldn't care about. Such involvement would make your code much more vulnerable to change. To make this work in a system, you'll need to preserve the commonsense semantics of commands (that is, every object that has a `print` method should behave similarly when called).

This isn't an OO-specific technique either. Even shell scripts can benefit from this approach. In fact, a common Linux command employs polymorphism at the command line. The command fsck (which is not a cartoon swear word—really) performs a file system check. When you invoke fsck, it determines the file system type and then runs a delegate, such as `fsck.ext2`, `fsck.msdos`, or `fsck.vfat`, that performs the actual tests for that kind of file system. But you, as the requester, don't care. You tell the system to check the disk via an fsck command and it just does it. It's just the right amount of apathy.

So remember to "keep it DRY, keep it shy, and tell the other guy."

Andy Hunt and Dave Thomas are partners in The Pragmatic Programmers and authors of the Jolt Productivity Award-winning *The Pragmatic Starter Kit* book series. Contact them via www.PragmaticProgrammer.com.