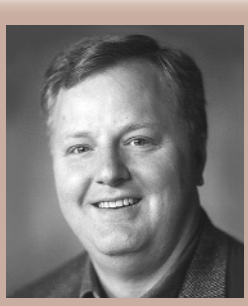
Editors: Andy Hunt and Dave Thomas ■ The Pragmatic Programmers
andy@pragmaticprogrammer.com ■ dave@pragmaticprogrammer.com

The Art of Enbugging

Andy Hunt and Dave Thomas

Many books and articles discuss debugging techniques: how to track down and correct the errors that, like sneaky little bugs, crept into our programs when we weren't watching. Referring to errors as "bugs" is a pleasant anthropomorphic distraction that lets a programmer avoid direct blame, but it does several disservices in the process. The largest of these is probably the concept that bugs somehow spontaneously appear in source code.



They do not. We put those errors in the code ourselves. In fact, on a bad day you might feel that you aren't programming at all, but rather *enbugging* the code—putting the bugs in. But it's actually more insidious than that. We rarely put bugs in directly; instead, we might set up conditions that will trick us into putting in bugs later. How can we prevent this? One of the best ways to keep future bugs out is to maintain a proper "separation of concerns"—that is, design the code so that classes and modules have clear, well-defined, and isolated responsibilities and well-understood semantics.

But in real life, that's tricky. Getting it right takes experience, which means getting it wrong a lot of times and learning to do it better. Fortunately, we've got a few handy shortcuts to help you out. The fundamental goal (while we're in an anthropomorphic mood) is to write *shy* code—code that doesn't reveal too much of itself to anyone else and doesn't talk to others any more than is necessary. Shy code keeps to itself, not like that gossipy neighbor who's involved in everyone else's comings and goings. Shy code would never show its "privates" to "friends," as some more promiscuous C++ code might.

This month we'll examine some ways to help us create shy code. Although we are primarily looking at object-oriented examples, the same principles apply to procedural code as well.

Tell, don't ask

A distinction of OO code (or code written in that style in any language) is the idea of issuing a command to some entity to get something done. You see this explicitly in languages such as Smalltalk and Ruby, where method invocation is viewed as messages being passed between objects, not as function calls. In Java, C++, and similar languages, the fact that method invocation looks an awful lot like a function call tends to distract from the "message-passing" metaphor. This is a shame; there's an important distinction to be made between procedural and OO styles here.

Procedural code tends to get information and then make decisions based on that information. OO code tells objects to do things (see Alec Sharp's *Smalltalk by Example*, McGraw-Hill, 1997). That is, in the model we

0740-7459/02/$17.00 © 2002 IEEE

want to follow, you send commands to objects telling them what you want done. We explicitly do *not* want to query an object about its state, make a decision, and then tell the object what to do. That starts to sound a bit like the gossipy neighbor—and not something that shy code should do.

As the caller, you should not make decisions based on the called object's state and then change the object's state. The logic you are implementing is probably the called-object's responsibility, not yours. For you to make decisions outside the object violates its encapsulation and provides a fertile breeding ground for bugs.

Consider the story of "The Paperboy and the Wallet," from our friend David Bock (www.javaguy.org/papers). Suppose the paperboy comes to your door, demanding payment for the week. You turn around, and the paperboy pulls your wallet out of your back pocket, takes the two bucks, and puts the wallet back. As absurd as that sounds, many programs are written in this style, which leaves the code open to all sorts of problems (and explains why the paper boy is now driving a Lexus).

Instead of asking for the wallet, the paperboy should instead tell the customer to pay the $2.00. Code should work the same way—we want to tell objects what to do, not ask them for their state. Adhering to this notion of "Tell, Don't Ask" is easier if you mentally categorize each of your functions and methods as either a *command* or a *query*, and document them as such in the source code (it helps if all commands are grouped together and all queries are grouped together). A routine acting as a command will likely change the object's state and might also return some useful value as a convenience. A query just gives you information about the object's state—and does not modify the object's externally visible state. That is, queries should be free of side effects as seen from the outside world. Now, we might want to do some precalculation or caching behind the scenes as needed, but fetching the value of x should not change the value of y.

Command-query separation keeps code shy; the caller doesn't know too much about how its command will be performed. That means we have reduced coupling by some measure, which is a good thing. Those implementation details are free to change with less chance of affecting the caller. Because the query methods are known to be side effect free, we can use them freely in unit tests and call them from assertions or from the debugger.

The Pretty Good Idea of Demeter

The more objects you talk to, the greater your risk of breaking when one of those objects changes. So not only do you want to say as little as possible to other objects, you also want to talk to as few other objects as possible.

A helpful tool to apply to this problem is called the "Law of Demeter for Functions," but that's a pretty dogmatic name. Like most "laws" in software development, it's more of a good idea than a physical constant of the universe. This good idea suggests that an object should only call

- Itself
- Any parameters that were passed in to the method
- Any objects it created
- Any directly held component objects

Conspicuous by its absence in the list is any method belonging to objects that were returned from some other call, as shown in the following Java code:

```
my_television.front_panel.
    switches.power.on();
```

Direct access of a child like this extends coupling from the caller farther than it needs to be. The caller is depending on navigating the object model such that

- A Television object has a front panel.
- A Front Panel has some switches.
- One of those switches is "power."
- Power has an "on" method, which will turn on the television.

Instead of asking for all this information, we just want to tell the television what to do:

```
my_television.power_up();
```

Now the caller doesn't need knowledge of `my_television`'s internal object model. This higher-level call sounds more like a user requirement and less like an implementation detail, which means we are programming closer to the problem domain as well (for more on this, see our book *The Pragmatic Programmer*, Addison-Wesley, 2000). When we discover the additional requirement that you can also turn the TV on using the remote control, we have a single, authoritative method to call.

The disadvantage of this approach is that you end up writing many small wrapper methods that do very little but delegate container traversal and such. The cost trade-off is inefficiency versus higher-class coupling. In the long run (and even in the short run in some cases), higher-class coupling is simply unacceptable, as that increases the odds that any change you make will break something somewhere else. It doesn't take much high-class coupling to create a lot of fragile, brittle code. In most cases, higher coupling's lifetime development and maintenance costs can easily swamp any minor runtime inefficiencies.

But for those occasions when speed is paramount and high coupling is acceptable, couple it to the hilt! Don't be shy. Make it clear in the documentation that these particular classes are inextricably wed to each other, and why.

Writing shy code is just a small start at preventing the introduction of bugs, but it really helps. Just as in the real world, good fences make good neighbors—as long as you don't peek through the fence.

Andy Hunt and **Dave Thomas** are partners in The Pragmatic Programmer, LLC. They feel that software consultants who cant program shouldn't be consulting, so they keep current by developin g complex software systems for their clients. Contact them via www.pragmaticprogrammer.com.