Shift Your Paradigm!

Why Your Code Sucks

by Dave Astels

September 27, 2004

Summary
If you are like most, and possibly even all, programmers (this author humbly included) then your code sucks. Maybe not all of it, and maybe not all the time, but certainly some of it, some of the time.

Your code sucks if it doesn't work.

Does your code work? How do you know? Can you prove it?

If you don't have tests for your code, it sucks. And I mean comprehensive, fine grained, programmer tests (aka something like unit tests), as well as higher level functional and integration tests. Tests that are automated. Tests that are run routinely. With the programmer tests run after any change to the code.

Without adequate tests (in terms of both quantity and quality) you simply can't have confidence that your codes works as required. And even if you are confident with your work... why should anyone else be? What about people who work on the code later? How do they know it still works after they make their changes?

Your code sucks if it isn't testable.

OK, so you decide to add tests to your code. That typically isn't so easy. The odds are that your code wasn't designed and/or implemented to be easily testable. That means you will have to make changes to it to make it testable... without breaking any of it (this is usually called *refactoring* these days). But how can you quickly and easily tell if you've broken anything without that comprehensive set of fine grained tests? Tricky.

It's even trickier than that. If you take the approach that you'll write tests for all your code *after* you write the code, you have to go to extra effort to try and design/write the code to be testable.

Maintaining an adequate level of testing when writing tests after the code being tested takes time and planning. And if you have to do additional work to make your code testable... well... that sucks.

Writing tests before you write the code means that the code is testable by definition. If you work one test at a time, you reap even more benefit, as each test only requires incremental changes to code that you know is solid.

Your code sucks if it's hard to read.

Code should be easy to read. Steve McConnell made the statement at his SD West '04 keynote that code should be convienent to read, not convienent to write.

There are several things involved here. Choosing good names that are self-explainatory is a good place to start. Strive for simple solutions, even if they are more verbose or inefficient. Whether inefficiency is a problem won't be known until profiling is done later in the project, in a real production situation.

Choose style and formating conventions early in the project, and conform to them. Require that everyone conform to them. This will be easier to do if everyone has a hand in deciding on the conventions and buys into them. This will make the code uniform and thus easier to read.

And get rid of any and all duplication. More on this later.

Your code sucks if it's not understandable.

This is closely related to the previous point. Code can be readable, but still not easily understood. Maybe it reads well but is misleading. Maybe you took the time to pick a good name for that variable, but the way it's used and what it means has changed, but the name hasn't. That's worse than using a cryptic name... at least in the latter case, the reader knows the name is meaningless. If the name is missleading, the reader can make ungrounded assumptions about what's happening in the code, leading to misunderstanding, and eventually bugs.

Your code sucks if it dogmatically conforms to a trendy framework at the cost of following good design/implimentation practices.

For example, Bob Martin recently raised the issue of dogmatically using private fields and getters/setters for a simple data structure (e.g. a DTO). If a field is transparently readable and writable why not simply make the field public? In most languages you can do that. Granted, in some you can't. For example, traditionally in Smalltalk all fields are private and all methods are public.

In general it's a good thing whenever you can throw out, or avoid writing, some code. Using a heavy framework generally requires that you must write a significant amount of code that has no business value.

There are a variety of lightweight frameworks for Java that are a response to the heavyweight frameworks (e.g. EJB) that have become matters of dogma lately. O'Reilly has a new book out on this topic, coauthored by Bruce Tate.

When making framework decisions, consider if a lighter framework will do the required job. Using something like Hibernate, Prevayler, Spring, PicoContainer, NakedObjects, etc. can be a real win in many situations. Never blindly adopt a heavy framework just because it's the current bandwagon. Likewise, don't blindly adopt a lightweight framework in defiance. Always give due consideration to your choices.

Your code sucks if it has duplication.

Duplication is probably the single, most important thing to banish from your code.

Duplication can take several forms:

textual
: This is the simplest and often the easiest to find. This is when you have sections of code that are identical or nearly so. This is often a result of copy & paste programming. Several techniques can be used to eliminate it, including: extracting methods and reusing them, making a method into a template method, pulling it up into a superclass, and providing the variations in subclasses, and other basic refactorings. See Fowler's "Refactoring" for more detail.

functional
: This is a special case of textual duplication and is present when you have multiple functions/methods that serve the same purspose. This is often seen in projects that are split up between groups that don't communicate enough. This could be a result of copy & paste, or it could be the result of ignorance of existing code or libraries. Practices and policies that encourage and promote knowledge of the entire codebase and all libraries and frameworks being used can help avoid functional duplication.

temporal
: This is a bit odder, more obscure, and not as easy to find. You have temporal duplication when the same work is done repeatedly when it doesn't have to be. This isn't a problem in the same way as the previous two forms of duplication since it doesn't involve having *extra* code laying about. It can become a problem as the system scales and the repeated work starts taking too much time. Various caching techques can be used to address this problem.

Of the three types I've identified above, textual duplication is the worst. It is the quickest to bite you as the system grows. It has a nasty habit of being viral: when you see code duplicated in the system, it can be tempted to think "Oh, well then, it's not a problem if I just nic this code and put a copy of it over here and fiddle it a bit. Hey, everyone else is doing it!" It's a lot like the Broken Window that Dave Thomas & Andy Hunt talk about. If you don't keep on top of the *little things* as they appear, the whole thing will soon go down the toilet.

Summary

So, chances are pretty good that some of your code sucks.

What are you going to do about it? Well, first you have to recognise the fact (and hopefully this article has helped with that) and admit that there's a problem.

Then you have a choice. You can choose to do something about it. You can learn new ways to work... ways that help you write better code. That's great. If only one programmer out there does this, I'll be happy.

Your other option is to do nothing about it, and continue writing code the way you always have.

It's your choice. Choose wisely.

Talk Back!

Have an opinion? Readers have already posted 21 comments about this weblog entry. Why not add yours?

RSS Feed

If you'd like to be notified whenever Dave Astels adds a new entry to his weblog, subscribe to his RSS feed.

Digg | del.icio.us | Reddit

About the Blogger

Dave Astels has been developing hardware and software solutions for more than 20 years in domains ranging from environment control systems to electrical energy trading systems to mass market products. Since the late 1980s he has been working exclusively with object technologies: a mix of C++, Smalltalk, Java, and some more obscure OOPLs. Since the late 1990s, he has been studying, using, evangelizing, and teaching Agile Development processes and practices. He has coauthored/authored two books for Prentice Hall: "A Practical Guide to eXtreme Programming" and "Test-Driven Development: A Practical Guide". He also edits the TDD edition of The Coad Letter, which is part of the Borland Development Network. He co-founded and runs Adaption Software, Inc. (www.adaptionsoft.com).