IS SOFTWARE ENGINEERING IMMATURE?

James W. Cooper

Last week I went to a talk where the speaker, a well-known software technologist, made the assertion that Software Engineering is still an immature discipline. I started to wonder about that as I looked around the room and saw a bunch of typical compu-geeks with funny looking hair, dirty jeans and sloppy T-shirts. Maybe, I thought, it is not Software Engineering that is immature, but Software Engineers!

Now that's a thought that might bear some examination. Suppose we think about other sorts of engineering: electrical, civil and so forth. These engineers build concrete products using well-defined disciplines, and following well-known building or electrical regulations. But software engineers don't build anything very concrete. How many programmers does it take to screw in a light bulb? None! That's a hardware problem.

Engineers have disciplines. They wear white shirts, work regular daytime hours and are rewarded for designing and constructing real things. Programmers (whom I assume are the same thing as software engineers) make it a point to keep weird hours, play loud music and wear nontraditional garb whenever they can. Bridges, buildings, circuit boards and chips work mostly as designed. But software doesn't work very well. It frequently presents a quirky, idiosyncratic interface, crashes a lot and annoys users at least at much as it rewards them. Thank goodness we aren't building airplanes. Software crash kills thousands!

Now programming isn't that new a discipline. If you like to believe that Ada Lovelace was the first programmer, it is more than 100 years old. And I've been writing programs for more than 30 years. My kids grew up faster than that!

Lots of real, concrete things are driven by software these days. Cars, washing machines, mixers, CD players. They all work quite well. And if cars crashed as often as Windows, we wouldn't be a commuter society. But construction of cars is supervised mostly by automotive engineers, not software engineers. Shrink-wrapped software is built by programmers who are not part of the larger discipline enforced by concerns of safety and other regulations. So they don't care? They're sloppy?

What's the problem here? Why is software so awful? Are programmers as lazy and indolent as all that? Are they just illustrating the stereotype they are expected to conform to? Everyone knows that we programmers are strange. Maybe the fact that we make strange products isn't so surprising. If it's OK to come in a noon and work until midnight, play loud music in your office and eat junk food all day, maybe producing junky sleep-deprived software isn't such a surprise.

Maybe the problem is that programmers do what is expected of them. Bosses tolerate bad habits because programmers are smart and can wrangle code: and bosses can't. Maybe part of the problem is that bosses have no idea how to quantify what is going on in programming departments. There are just these acres and acres of modules with some unknowable number of defects. All non-trivial programs have at least one undiscovered bug. A sufficient case for program triviality is that a program has no bugs. This is sadly nearly always the case.

In fact the rumors of 60,000 bugs in Windows-2000 is bizarre. We know there aren't really that many, or the thing wouldn't work at all. On the other hand, we suspect the number is still large. One of the main issues in Windows NT was the infamous Blue Screen of Death. Windows-2000 is supposed to be much more stable, but it has 13 pages in the small installation guide on how to deal with the blue screen when it strikes. Ah, progress. It's not a bug, it's a feature. I read that the next version will provide these screens in a choice of decorator colors!

Microsoft is well known for working its programmers outrageously long hours with the promise of eventual stock options if you can last it out. Does this lead to more quality or more potato chips and Twinkies.

And lest you think that Microsoft is the only vendor with problems in programming management, I spent 2 days looking for a bug in an established and unchanged system, which failed, it turned out because I'd installed the release version of Java 1.2.2. Installing the *beta* of 1.3 (er, that is a release candidate) fixed the problem. This does not lead us to great confidence in the stability of anybody's commercial software.

And how about installing the Preview Version of Netscape 6? It installs Java 1.3 (preview?!) and thereafter, running any JVM gives you the reassuring error message

```
Version should be 1.2 but registry is 1.3
```

Uh-oh.

One place where they are spending a lot of time on code robustness is the IBM Centre for Java Development in Hursley. (see www.ibm.com/java for details).They currently provide a version of the 1.1.8 JVM that is both extremely robust and has very high performance, including IBM's own JIT compiler. Is it at Java 2 level? No, but it is very stable. In fact, I found that using this Java 1.1.8 JVM solved several database access instabilities that had been driving me nuts. Take your pick.

Maybe one thing is that programmers don't make anything very concrete, and that they are dealing in wishes and ideals more than hard products. If all you have to do is make some code that does something, who is to say what the standards for completeness and correctness really are?

Maybe the problem is that software companies have discovered that customers are not really willing to pay enough for quality. It's better to ship the beta. This is rather like an instrument company I once worked for, where the engineers carefully built a model of the product the assemblers were supposed to replicate for customers. Then what did management do? They shipped the model to the first customer and let the assembly department scramble.

What Price Quality?

So what do we have to pay for quality? Two or three or ten times as much? Are we still writing tangles of pasta in the C language? Can Java help us out of this bind? Can the OO disciplines of Design Pattern use, UML design, or simple things like unit testing help? What about more mature programmers and managers? How about XML? Trouble is nobody can read it and nobody can write it.

OK, maybe the scraggly clothes crowd isn't at fault. Maybe they only respond to what the pointy-haired boss asks for. Does this mean that licensing software engineers would do away with this problem?

How can you license software engineers? Software engineering isn't a mature enough discipline yet. Yeah, I heard that before. Booga booga. We'll just scare the bugs away. Call the Orkin man!

Where does this screed of excrescence at software defects come out?

Maybe it comes out in writing code in better languages like Java. Java might be part of the solution as we move to using it more pervasively. It is embedded in robots, in servers and even in the devices we once thought would have just a few bytes of ROM. You can build really elegant systems in Java that scale pretty well and don't come crashing down. That is, if the JVM doesn't come crashing down.

So we can write much better code in Java just because it *makes* us use OO techniques. If we spend some time thinking about the objects that make up our system, we can write pretty good code. If we take the time to recognize the desirability of throwing away the first version or two of the object design before we get one that really works, we can write code that is pretty robust and extensible. If we are allowed to take the time to write code thoughtfully, it will be better code.

Does this mean more mind-numbing walk-throughs and code-inspections? I hope not. They just waste my time and yours. One discipline that has gotten a bit of attention lately is Extreme Programming. With a name like that, you think of in-line skates, elbow pads and parachutes. But, it is really incredibly simple minded at its heart. Every time you add a feature to a program, you write a test case to prove it works. This sounds great, but coming up with these test cases is not as simple as you'd think. You have to warp your mind in new ways to make sure you're testing the right thing. And while you're at it, why not write the tests before you write the code?

And what do test cases have to do with objects? How about objects that can test themselves? Now there's an Extreme Idea, and not one that you dismiss out of hand. If we design a system as a set of objects, can we design objects that test the system or themselves? Why not? Do you want software that works or just software that's done? When we subclass the objects, we need to subclass the tests, but that is now easier than writing them again.

Some thoughts on building objects

If you are determined to design better software using good OO techniques, here are a few suggestions to remind you of how you can do it. Many of them are suggestions by Riel (see reference #1).

1. Define a set of classes that are significant. Each class should have a real role to play, and should not just be a broken off piece of some monolithic design.
2. Classes should hide their data. You shouldn't ever need to know much about what is inside a class.
3. Data and the actions on that data should always be in the same class.
4. Classes should not know they have descendants. Otherwise why write descendants at all?
5. Beware of the "god class." A single class should never be too much of a controller of the entire program. This can easily become a problem when writing Mediator patterns.
6. Classes should not be actions. If your class hierarchy has classes whose names are verbs (like "Print") try again.
7. Classes should not be single minded. Maybe it is an action disguised as a class. Try again.
8. Classes should be significant. Why does a class exist? See #7.

How do you write quality into your software?

Do managers think that allocating time for testing during the heat of deadline-driven development is a waste of time? Do programmers think so? How about you? Does your software suck?

If I haven't offended you yet, it wasn't for lack of trying. What do you think explains the sorry state of software quality? And how do you solve the problem in your work, or in general? Let me know. I'll tell you what I hear.

Some Useful References

1. Arthur J. Riel, *Object Oriented Design Heuristics*, Addison-Wesley, 1996.
2. Kent Beck, *Extreme Programming Explained*, Addison-Wesley, 2000
3. James Cooper, *Java Design Patterns: A Tutorial*, Addison-Wesley, 2000.
4. Steve McConnell, *After the Gold Rush*, Microsoft Press, 1999.
5. Martin Fowler, *Refactoring- Improving the Design of Existing Code*, Addison-Wesley, 1999.