28

Harm

1. I will not produce harmful code.

The first promise of the software professional is: DO NO HARM! And that means that your code must not harm your users, your employers, your managers, or your fellow programmers.

You must know what your code does. You must know that it works. And you must know that it is clean.

Sometime back, it was discovered that some programmers at Volkswagen wrote some code that purposely thwarted EPA emissions tests. Those programmers wrote harmful code.

It was harmful because it was deceitful. That code fooled the EPA into allowing cars to be sold that emitted 20 times the amount of harmful nitrous oxides as the EPA deemed safe. Therefore, that code potentially harmed the health of everyone living where those cars were driven.

What should happen to those programmers? Did they know the purpose of that code? Should they have known?

I'd fire them and prosecute them. Because whether they knew or not, they should have known. Hiding behind requirements written by others is no excuse. It's our fingers on the keyboard. It's our code. We must know what it does!

That's a tough one, isn't it? We write the code that makes our machines work. And those machines are often in positions to do tremendous harm. Since we will be held responsible for any harm our code does, we must be responsible for knowing what our code will do.

Each programmer should be held accountable based on their level of experience and responsibility. As you advance in experience and position, your responsibility for your actions and the actions of those under your charge increases.

Clearly, we can't hold junior programmers as responsible as team leads. We can't hold team leads as responsible as senior developers. But those senior people ought to be held to a very high standard, and be ultimately responsible for those whom they direct.

That doesn't mean that all the blame goes to the senior developers or the managers. Every programmer is responsible for knowing what the code does, to the level of their maturity and understanding. Every programmer is responsible for the harm their code does.

No Harm to Society

First, you will do no harm to the society in which you live.

This is the rule that the VW programmers broke. Their software might have benefited their employer—Volkswagen. However, it harmed society in general. And we, programmers, must never do that.

But how do you know if you are harming society or not? For example, is building software that controls weapons systems harmful to society? What about gambling software? What about violent or sexist video games? What about pornography?

If your software is within the law, might it still be harmful to society?

Frankly, that's a matter for your own judgment. You'll just have to make the best call you can. Your conscience is going to have to be your guide. Choose wisely!

Another example of harm to society was Healthcare.gov, the website that was mandated by the Affordable Care Act. Although in this case the harm was unintentional.

The Affordable Care Act was a bill passed by the Congress of the United States and signed into law by the president. Among its many directives was the demand that a website be created and activated on October 1, 2013.

Now, never mind the insanity of specifying, by law, a date by which a whole new massive software system must be activated. The real problem was that on October 1, 2013, they actually turned it on.

Do you think, maybe, there were some programmers hiding under their desks that day?

Oh man, I think they turned it on.

Yeah, yeah, they really shouldn't be doing that.

Oh, my poor mother. Whatever will she do?

I won't bore you with a laundry list of the problems that ensued. Suffice it to say that the website did not work—at all—for months. This is a case where a technical screwup put a huge new public policy at risk. The law was very nearly overturned because of these failures. And no matter what you think about the politics of the situation, that was harm to the society.

Who was responsible for that harm? Every programmer, team lead, manager, and director who knew that system wasn't ready and who nevertheless remained silent.

That harm upon society was perpetrated by every software developer who maintained a passive-aggressive attitude toward their management. Everyone who said: "I'm just doing my job—it's their problem."

Every software developer who knew something was wrong, and yet did nothing to stop the deployment of that system, shares part of the blame.

Because here's the thing. One of the prime reasons you were hired as a programmer was because you know when things are going wrong. You have the knowledge to identify trouble before it happens. And therefore, you have the responsibility to speak up before something terrible happens.

Harm to Function

You must KNOW that your code works. You must KNOW that the functioning of your code will not do harm to your company, your users, or your fellow programmers.

On August 1, 2012, some technicians at Knight Capital loaded their servers with new software. Unfortunately, they loaded only seven of the eight servers, leaving the eighth running with the older version.

Why they made this mistake is anybody's guess. Somebody got sloppy.

Knight Capital ran a trading system. They traded stocks on the NYSE. Part of their operation was to take large "parent" trades and break them up into many smaller "child" trades in order to prevent other traders from seeing the size of the initial parent trade and adjusting prices accordingly.

Eight years earlier, a simple version of this parent–child algorithm, named Power-Peg, was disabled and replaced with something much better, called SMARS. Oddly, however, the old Power-Peg code was not removed from the system. It was simply disabled with a flag.

That flag had been used to regulate the parent–child process. When the flag was on, child trades were made. When enough child trades had been made to satisfy the parent trade, the flag was turned off.

They disabled the Power-Peg code by simply leaving the flag off.

Unfortunately, the new software update that made it onto only seven of the eight servers repurposed that flag. They turned it on. And that eighth server started making child trades in a high-speed infinite loop.

They knew something had gone wrong, but they didn't know exactly what. It took them 45 minutes to shut down that errant server; 45 minutes while it was making bad trades in an infinite loop.

The bottom line is that, in those first 45 minutes of trading, Knight Capital had unintentionally purchased over $7 billion worth of stock that it did not want, and had to sell it off at a $460 million loss. Worse, they only had $360M in cash. They were bankrupt.

Forty-five minutes; one dumb mistake; $460 million.

And what was that mistake? The mistake was that they did not KNOW what their system was going to do.

At this point, you may be worried that I'm demanding that programmers have perfect knowledge about the behavior of their code. Of course, perfect knowledge is not achievable. There will always be a knowledge deficit of some kind.

The issue isn't the perfection of knowledge. Rather, the issue is to KNOW that there will be no harm.

Those poor guys at Knight Capital had a knowledge deficit that was horribly harmful—and given what was at stake, it was one they should not have allowed to exist.

Or let's take the case of Toyota, and the software system that made cars accelerate uncontrollably.

As many as 89 people were killed by that software. And more were injured.

Imagine that you are driving in a crowded downtown business district. Imagine that your car suddenly begins to accelerate, and your brakes stop working. Within seconds, you are rocketing through stop lights and crosswalks, with no way to stop.

That's what investigators found that the Toyota software could do—and probably had done.

The software was killing people.

The programmers who wrote that code did not KNOW that their code would not kill. Notice the double negative. They did not KNOW that their code would NOT kill. And they should have known that. They should have known that their code would NOT kill.

Now again, this is all about risk. When the stakes are high, you want to drive your knowledge as near to perfection as you can get it. If lives are at stake, you have to KNOW that your code won't kill anyone. If fortunes are at stake, you have to KNOW that your code won't lose them.

On the other hand, if you are writing a chat application or a simple shopping cart website, neither fortunes nor lives are at stake …

… or are they?

What if someone using your chat application starts having a medical emergency and types: "HELP ME. CALL 911" on your app? What if your app malfunctions and drops that message?

What if your website leaks personal information to hackers who use it to steal identities?

And what if the poor functioning of your code drives customers away from your employer and to a competitor?

The point is that it's easy to underestimate the harm that can be done by software. It's comforting to think that your software can't harm anyone, because it's not important enough.

But you forget that software is very expensive to write, and at the very least, the money spent on its development is at stake, not to mention whatever the users put at stake by depending on it. The bottom line is that there's almost always more at stake than you think.

No Harm to Structure

You must not harm the structure of the code. You must keep the code clean and well organized.

Ask yourself why the programmers at Knight Capital did not KNOW that their code could be harmful.

I think the answer is pretty obvious. They had forgotten that the Power-Peg software was still in the system. They had forgotten that the flag they repurposed would activate it. And they assumed that all servers would always have the same software running.

The reason they did not know their system was harmful is because of the harm they had done to the structure of the system by leaving that dead code in place.

And that’s one of the big reasons why code structure and cleanliness are so important. The more tangled the structure, the more difficulty in knowing what the code will do. The bigger the mess, the more the uncertainty.

Take the Toyota case, for example. Why didn’t they know their software could kill people? Do you think the fact that they had thousands of global variables might have been a factor?

Making a mess in the software undermines your ability to know what the software does, and therefore your ability to prevent harm.

Messy software is harmful software.

Now, some of you might object by saying that sometimes a quick-and-dirty patch is necessary to fix a nasty production bug.

Sure. Of course. If you can fix a production crisis with a quick-and-dirty patch, then you should do it. No question.

A stupid idea that works is not a stupid idea.

However, you can’t leave that quick-and-dirty patch in place without causing harm. The longer that patch remains in the code, the more harm it can do.

Remember that the Knight Capital debacle would not have happened if the old Power-Peg code had been removed from the codebase. It was that old, defunct code that actually made the bad trades.

So, what do we mean by “Harm to structure?”

Clearly, thousands of global variables is a structural flaw. So is dead code left in the codebase.

Structural harm is harm to the organization and content of the source code. It is anything that makes that source code hard to read, hard to understand, hard to change, or hard to reuse.

It is the responsibility of every professional software developer to know the disciplines and standards of good software structure. They should know how to refactor, how to write tests, how to recognize bad code, and how to decouple designs and create appropriate architectural boundaries. They should know and apply the principles of low- and high-level design. And it is the responsibility of every senior developer to make sure that younger developers learn these things, and satisfy them in the code that they write.

Soft

The first word is software is *SOFT*. Software is supposed to be SOFT. It’s supposed to be easy to change. If we didn’t want it to be easy to change, we’d have called it HARDware.

It’s important to remember why software exists at all. We invented it as a means to make the behavior of machines easy to change. To the extent that our software is hard to change, we have thwarted the very reason that software exists.

Remember that software has two values. There’s the value of its behavior; and then there’s the value of its “softness.” Our customers and users expect us to be able to change that behavior easily and without high cost.

Which of these two values is greater? Which value should we prioritize above the other? We can answer that question with a simple thought experiment.

Imagine two programs: One works perfectly but is impossible to change; the other does nothing correctly but is easy to change.

Which is more valuable?

I hate to be the one to tell you this, but in case you hadn’t noticed it, software requirements tend to change, and when they change, that first software will become useless—forever.

On the other hand, that second software can be made to work, because it’s easy to change. It may take some time and money to get it to work initially, but after that, it’ll continue to work forever with minimal effort.

And therefore, it is the second of the two values that should be given priority in all but the most urgent of situations.

And what do I mean by urgent? I mean a production disaster that’s losing the company $10 million per minute. That’s urgent.

I do not mean a software startup. A startup is not an urgent situation requiring you to create inflexible software. Indeed, the opposite is true. The one thing that is absolutely certain in a startup is that you are creating the wrong product.

No product survives contact with the user. As soon as you start to put the product into users’ hands, you will discover that the product you’ve built is wrong in a hundred different ways. And if you can’t change it without making a mess, you’re doomed.

Indeed, this is one of the biggest problems with software startups. They believe they are in an urgent situation requiring them to throw out all the rules and dash to the finish line, leaving a huge mess in their wake.

Most of the time that huge mess starts to slow them down long before they do their first deployment. They’ll go faster, and better, and survive a lot longer, if they keep the structure of the software from harm.

When it comes to software, it never pays to rush

—Brian Marick

Tests

If you follow a comprehensive testing discipline, like TDD, test & commit || revert (TCR), or small bundles, you will KNOW, to a high degree of certainty, that the code works.

How can you prevent harm to the behavior of your code if you don’t have the tests that demonstrate that it works?

How can you prevent harm to the structure of your code if you don’t have the tests that allow you to clean it?

Is a testing discipline really a prerequisite to professionalism? Am I really suggesting that you can’t be a professional software developer unless you practice such a discipline?

Yes, I think that’s true. Or rather, it is becoming true. It is true for some of us; and with time it is becoming true for more and more of us. I think the time will come, and relatively soon, when the majority of programmers will agree that practicing a testing discipline is part of the minimum set of disciplines and behaviors that mark a professional developer.

Why do I believe that?

Because, as I said earlier, we rule the world! We write the rules that make the whole world work.

In our society, nothing gets bought or sold without software. Nearly all correspondence is through software. Nearly all documents are written with software. Laws don’t get passed or enforced without it. There is virtually no activity of daily life that does not involve software.

Without software, our society does not function. Software has become the most important component in the infrastructure of our civilization.

Society does not understand this yet. We programmers don’t really understand it yet either. But the realization is dawning that the software we write is critical. The realization is dawning that too many lives and fortunes depend upon our software. And the realization is dawning that software is being written by people who do not profess a minimum level of discipline.

So, yes, I think TDD, or some discipline very much like it, will eventually be considered a minimum standard behavior for professional software developers. I think our customers and our users will insist upon it.