Building Real Software

Developing and Maintaining Secure and Reliable Software in the Real World

Monday, February 13, 2012

Technical Debt - How much is it Really Costing you?

The idea behind the technical debt metaphor is that there is a cost to taking short cuts (intentional technical debt) or making mistakes (unintentional technical debt) and that the cost of not dealing with these short cuts and mistakes will increase over time.

The problem with this metaphor is that with financial debt, we know how much it would cost to pay off a debt off today and we can calculate how much interest we will have to pay in the future. Technical debt though is much fuzzier. We don't really know how much debt we have taken on – you may have taken on a lot of unintentional technical debt – and you may still be taking it on without knowing it. And we can't quantify how much it is really costing us – how much interest we have paid so far, what the total cost may be in the future if we don't take care of it today.

Some people have tried to put technical debt into concrete financial terms. For example, according to CAST Software's CRASH report

> "applications carry on average $3.61 of technical debt per line of code".

For some reason, the average cost of Java apps was even higher: $5.42 per line of code. These numbers are calculated from running static structural analysis on their customers' code.

Sonar, an Open Source dashboard for managing code quality, also tries to calculate a technical debt cost for a code base, again using static analysis findings like code coverage of automated tests, code complexity, duplication, violations of coding practices, comment density.

Thinking of technical debt in this way is interesting, but let's stop pretending that these are hard numbers that we can use to make trade-off decisions. Although the numbers appear precise, they're arbitrary, guesses. And they assume that technical debt can be calculated by a tool looking at the structure of the code. Unfortunately, dealing with technical debt is not that straightforward.

But if debt is too fuzzy to be measured in detailed cost terms, how do you know what kind of debt is hurting you the most, how do you know when you have too much? Let's look at different kinds of technical debt, and how much they might cost you, using a fuzzier approach.

$$$ Making a fundamental mistake in architecture or the platform technology – you don't find out until too late, until you have real customers using the system, that a key piece of technology like the database or messaging fabric doesn't scale or isn't reliable, or that you can't scale out your architecture like you need to because of core dependency problems, or you you made some fundamentally incorrect assumptions on how the system is supposed to work or how customers will use it. Now you have no choice but to start again or at least rewrite big chunks of the system to get it to work or to keep it working, and you don't have the time to do this properly.

$$-$$$ Error-prone code – the 20% of the code where 80% of bugs are found. Capers Jones says that all big systems have a small number of routines where bugs and problems cluster, code that is hard to understand and expensive and dangerous to change because it was done badly in the first place or it went to hell over time because of the accumulation of short-sighted fixes and changes. Not rewriting this code is one of the most expensive mistakes that developers make.

$-$$ The system can't be easily tested – because you don't have good automated tests, or the tests are brittle and slow and keep falling apart when you change the code. Testing costs can make up more than half of the cost of making any change or fix – sometimes testing can take much more time and cost much more than making the fix – and testing costs tend to go up over time as you write more code, as the system adds more interfaces and options.

$-$$ Not taking care of packaging and release and deployment. Relying too much on manual steps and manual checks, leading to mistakes and problems in production, late nights. Like testing, release and deployment costs don't go away, they just keep adding up incrementally.

$-$$ Code that mysteriously works, but nobody is sure how or why – usually performance-critical or

About Me

Jim Bird

I am an experienced software development manager, project manager and CTO focused on hard problems in software development and maintenance, software quality and security. For the last 15 years I have managed teams building and operating high-performance financial systems. My special interest is how small teams can be most effective in building real software: high-quality, secure systems at the extreme limits of reliability, performance, and adaptability. Software that has to work, that is built right, and built to last. I use this blog to explore ideas and problems in software development that are important to me. To reflect and to find new answers.

View my complete profile

Disclaimer

This is my personal blog - it represents my views and not those of my employer.

SANS Application Security

I also blog on application security issues on the SANS Application Security Street Fighter blog. You can follow my posts here.

Top Posts

safety-critical low-level plumbing code written by a wizard who has long since left the company. It might be beautiful code, but if nobody on the team understands it, it's a time bomb – someday, somebody is going to have to change it or fix it, or try to.

$-$$ Forward and backward compatibility adapters and compromises. This is necessary, short-term debt. But the cost rises the longer that you have to maintain these compromises.

$-$$ Out of date libraries and middleware stack – you've fallen behind on applying patches and upgrades. Even if the code that you have now is stable, you run some risk of unpatched security vulnerabilities. The longer that this goes on, the further behind you are, the higher the risk – at some point if the software is no longer supported or supportable, and your hand is called.

$-$$ Duplicate, copy-and-paste code. This is one of the bugaboos of technical debt and static analysis tools. Almost everybody has it. But how bad is it, really? The cost depends on how many clones developers have made, how often they need to be changed, how many subtle differences there are between the different copies, and how easily you can find the copies and keep track of them. If the developer who made the copies is still on the team and does a good job of keeping track of all of them, it doesn't cost much if anything.

$-$$ Known, outstanding bugs in code and unresolved static analysis findings. The cost and risk depends on how many bugs and warnings you have, and how nasty they are. But if they are real problems, they should have been fixed by now. Is a bug really a bug if it isn't bugging anyone?

$-$$ Inefficient design or implementation, "throwing hardware at it", using too much memory or network bandwidth or CPU. Hardware is cheap, but these costs can add up a lot as you scale out.

$ Inconsistent use of programming idioms and patterns – developers either didn't understand the existing patterns, or didn't like them and introduced new ones, or didn't care and just wanted to get their change done. It's ugly, and it can be frustrating for developers. But the real cost of living with the situation is often less than trying to clean it all up.

$ Missing or poor error handling and exception handling. It will constantly bite you in the ass in production, but it won't cost a lot to at least get it mostly right.

$0.01 Hard coding, magic numbers, code that isn't standards compliant, poor element naming, missing comments, and code that needs tidying. This is a pain in the ass, but it's the kind of thing that is easy to clean up as part of standard refactoring work.

$0.01 Out of date documentation – another issue that is commonly considered in technical debt. But let's be honest, most programmers don't read documentation anyways. If nobody is using it, get rid of it. If people are using it, why isn't it up to date?

$0.00 Hand-rolled code that could have and should have been done using built-in language features or libraries, or existing framework or common services. It's disappointing when somebody recognizes it, but unless this hand-rolled code has lots of bugs in it, it's a sunk cost, not a cost that is increasing over time.

There are different kinds of debt, with different costs. Figuring out where your real costs are, and what to do about it, isn't easy.

Posted by Jim Bird at 1:24 PM

Labels: quality, static analysis, technical debt

21 comments:

Anonymous said...

Thank you for the insightful post. Please do keep on writing.

February 15, 2012 8:05 AM

Brian M said...

Unfortunately nothing new here (meant in a nice way!)

Wearing most of those T shirts :(

Thanks for putting it in an article - compulsory reading for all our developers now :)
:)

Grail

Developers just don't go to Security Conferences

You can't be Agile in Maintenance?

Devops has made Release and Deployment Cool

Everything I can find about Software Maintenance

Has Static Analysis reached its Limits?

Diminishing returns in software development and maintenance

Labels

agile development (44)

books (8)

Construx (8)

Continuous Deployment (8)

devops (13)

incremental development (7)

iterative development (5)

Kanban (7)

leadership (3)

maintenance (16)

OpenSAMM (4)

OWASP (17)

project management (15)

quality (35)

reliability (8)

risk management (12)

SANS (17)

Scrum (10)

security (54)

software development (19)

static analysis (7)

teams (1)

testing (4)

XP (10)

Search This Blog

Search

February 17, 2012 3:18 AM

Russell Allen said...

I was very surprised to see "poor element naming" so low on your list? If this includes method names and class names, my personal experience suggests this can be a very expensive form of debt, and one which is relatively cheap to pay back with instant dividends. (Ctrl+R+R anyone?)

February 17, 2012 4:05 AM

D&E Photography said...

My "day job" is information security, and you blog is one of my regular reads. Frankly most of your posts are more relevant to security that those from the security world that I read! :)

Why? Because quality software processes matter, should be part of any SDLC and affect the integrity and availability of every enterprise IT ecosystem.

Great blog, and I very much hope you will continue into the foreseeable future.

February 17, 2012 4:22 AM

Philip Oakley said...

"problem with this metaphor is that with financial debt.." The flip side to the coin (sic) is the way most folk misunderstand financial debt - Choose your favourite bank - financial debt is just as misunderstood!

February 17, 2012 5:58 AM

RobKraft said...

An Excellent post. Creating a list of the specific sources of technical debt was well done. Adding the relative cost of those forms of debt is masterful.

Rob Kraft
My SoftDev blog

February 17, 2012 6:42 AM

Jim Bird said...

@Russell,

You make a good point - poor element naming can be more expensive. A badly named class or method or variable at least delays someone's understanding of what the code is really doing. These delays are waste and can add up over time. Worse, it can cause people to make incorrect assumptions, overlook a problem or make a mistake especially when they are dealing with older code that they're not familiar with - code that has changed over time. We do look out for this in code reviews and spend time coming up with good names (which can be an art in itself). As you point out, it's cheap and easy to fix when you recognize it.

@Philip,I still think there is a fundamental difference between technical debt and financial debt. It's the difference between not knowing and not doing anything about something. Governments and companies and people who take on too much debt know it and can measure it - they just choose to pretend otherwise. But the worst kinds of technical debt are the kinds that teams take on without knowing it: fundamental and unintentional mistakes in design or platform technology selection or making mistakes in implementation because they don't know the language or the design problems yet. Even when you're trying to do everything carefully, when you're trying not to cut corners, you can still be taking on debt. And the costs of these mistakes can be huge.

February 17, 2012 7:11 AM

Jeremy M said...

Documentation looks like it's one of those things you can do without, but its abscence is a bomb with a long fuse. As long as there are still enough people on the team who were there when the code was written not having any documentation does not really hurt you because there's always someone who can explain what the code is meant to be doing. But if your team expriences a high degree of turn-over / wastage you can suddenly find yourself in a position where the business is crying out for something to be changed and there's no way to change it safely because no one understands what the application actually does.

February 17, 2012 7:46 AM

Graham Harris said...

Another form of debt is not explicitly adding code units to a project but relying on project search paths to retrieve the code unit at compile time.

February 17, 2012 7:55 AM

Jim Bird said...

@Jeremy,

I'm becoming more skeptical about documentation and the value of documentation over time. Even with teams that have high turnover. Because documentation, even if there is any, is almost always incomplete and out of date. You can't trust it. So you have to do what programmers have always had to do: you get somebody to show you how the system works, you find the code, and you start walking through it.

February 17, 2012 9:59 AM

Anonymous said...

Disagree with:

$0.00 Hand-rolled code that could have and should have been done using built-in language features or libraries, or existing framework or common services. It's disappointing when somebody recognizes it, but unless this hand-rolled code has lots of bugs in it, it's a sunk cost, not a cost that is increasing over time.

Every line of code has to be maintained in some fashion. Even if it is solid... someone has to keep an eye on it.

February 17, 2012 10:11 PM

Dodgy_Coder said...

Yeah I like your point on inline documentation (if noone is reading it then take it out). Personally, if I see a section of code with a lot of documentation/comments then its sort of a warning sign that the code may not be clearly written, or not obvious what is happening. With a well written piece of code or module, it should be clear what it is doing without looking at the comments.

February 18, 2012 5:23 AM

Dicky said...

Good list of technical debt. I hope that all the stakeholders including both developers and non developers would remember that there is a cost that we have to pay each time we take a short cut to complete a new features

February 19, 2012 11:14 AM

Anonymous said...

Everyone is so worried about phrases and the latest trends - just write software people. Too much time spent on meaningless details while other teams are banging out software hits one after the other.

February 19, 2012 3:07 PM

David N said...

Listing these out is a tedious, yet necessary task, in many environments, it seems. I've recently had a bout with a tech-debt-ridden project.

Fortunately, it will be degraded to a POC. I thank the almighty (and our management) for the hind-sight.

February 20, 2012 1:26 AM

Mike K said...

I'd disagree with the ordering or your priorities, it's too reminiscent of a cowboy-coding mentality. Of course "error prone-code" is a source of errors, but pointing that out is a bit like telling developers to "avoid making mistakes" without telling them how.

Almost every subsequent point is a symptom, if not a cause, of the processes that lead

to error-prone code. I'll pick out a few:

Poor Element naming: The developer didn't think too much about the code he/she was writing. Unclear element names, or elements or objects used twice for different purposes, can really confuse someone who is trying to fix a bug.

Lack of testability: What hope have you of fixing that error-prone routine if you can't test it fully.

Duplicate/Copy/Paste code: Twice as much code means roughly twice as many bugs. And when you fix one copy, are you sure you fixed the same bug in all the other copies - are you sure you didn't miss one?

Hand rolled code: Don't reinvent the wheel. Someone has probably done it better than you, and they've tested it, too, which you probably haven't. Also: more lines code usually means more bugs.

Inconsistent use of coding patterns: How can a developer hope to understand, debug and fix code if each developer uses their own standards and the developer has to wade through and understand fifteen different usage patterns?

Poor error handling: Your software will be silently failing, doing the wrong thing, and you'll never know. Very dangerous if you ever write software that has anything to do with financial transactions.

I know that some developers will happily spend days gold-plating their code, but I think this post veers too far in the other direction. The worst kind of technical debt is code that only one developer understands and can fix, and good standards and methodologies are all about avoiding that very problem.

February 20, 2012 3:44 AM

Jim Bird said...

@Mike K,

Don't get me wrong, I'm not saying that it's ok to write bad code. What I am trying to say is that most of what people think of as technical debt (copy and paste, long methods, bad element names) is the kind of stuff that is easy to see and (usually) easy to fix and it should be fixed when you have the chance. But if you're working on a large code base over a long period of time, you're going to have to put up with some of it. Sure, it adds up, but it can be paid back.

The real cost of technical debt is mostly hidden: making a fundamental mistake in technology or architecture, or not being able to fix the root cause of most of your bugs because you don't understand where they care coming from. This is the kind of debt that can take a long time to recognize, and it can be crippling.

February 20, 2012 9:13 AM

Anonymous said...

About copy/paste reuse ... Idunno, my experience is that it can get quite costly. The original programmer is almost never available when such things need fixing, and nobody keeps track of all duplications. Finding occurences based on regular expressions is at best error-prone, and usually programmers using a lot of copy and paste also write tests with a lot of holes in them, so you can't rely on existing tests to find errors in duplicated blocks of code.

I'm not so sure about inconsistent use of idioms and patterns. Unless you allow for a certain level of inconsistency, your code will never evolve. If something is consistently done in a bad way throughout the codebase, it's stupid to start changing every place at once, but IMO it's similarly stupid to enforce the same bad solution in places where changes occur or in new code.

Out of date documentation is IMO also a higher cost than you estimate, but on a different level/from a different POV. Maintaining documentation, unless you have a standard that lets you get away with very focused, slim documentation, is too costly for most projects. However, misleading javadoc- or doxygen-generated documentation is costly in that clients of the wrongly documented APIs will not only spend a lot more time than necessary for very simple issues, but also waste time of other team members, who know the code and don't need the documentation. This can IMO increase the cost by an order of magnitude.

February 23, 2012 1:26 AM

Anonymous said...

Jim, great discussion. The greatest service you have done with your ranking is to (correctly) place architectural errors as the most expensive you can make. Of course the irony here is that most organizations do not even bother to design their computer systems - they just start coding.

And now we see the true source of the stat that 80% of IT projects fail - the lack of architecture.

February 28, 2012 12:35 PM

Gustavo Chaves said...

Jeff Thalhammer agrees with the shortcomings of the *technical debt* metaphor and suggests that *technical insurance* is a better one. Worth a read.

March 8, 2012 8:54 AM

Jim Bird said...

@Gustavo Cool, thanks for the link on technical insurance, that's an interesting take on the problem

March 9, 2012 8:38 AM

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)