THE OVERFLOW

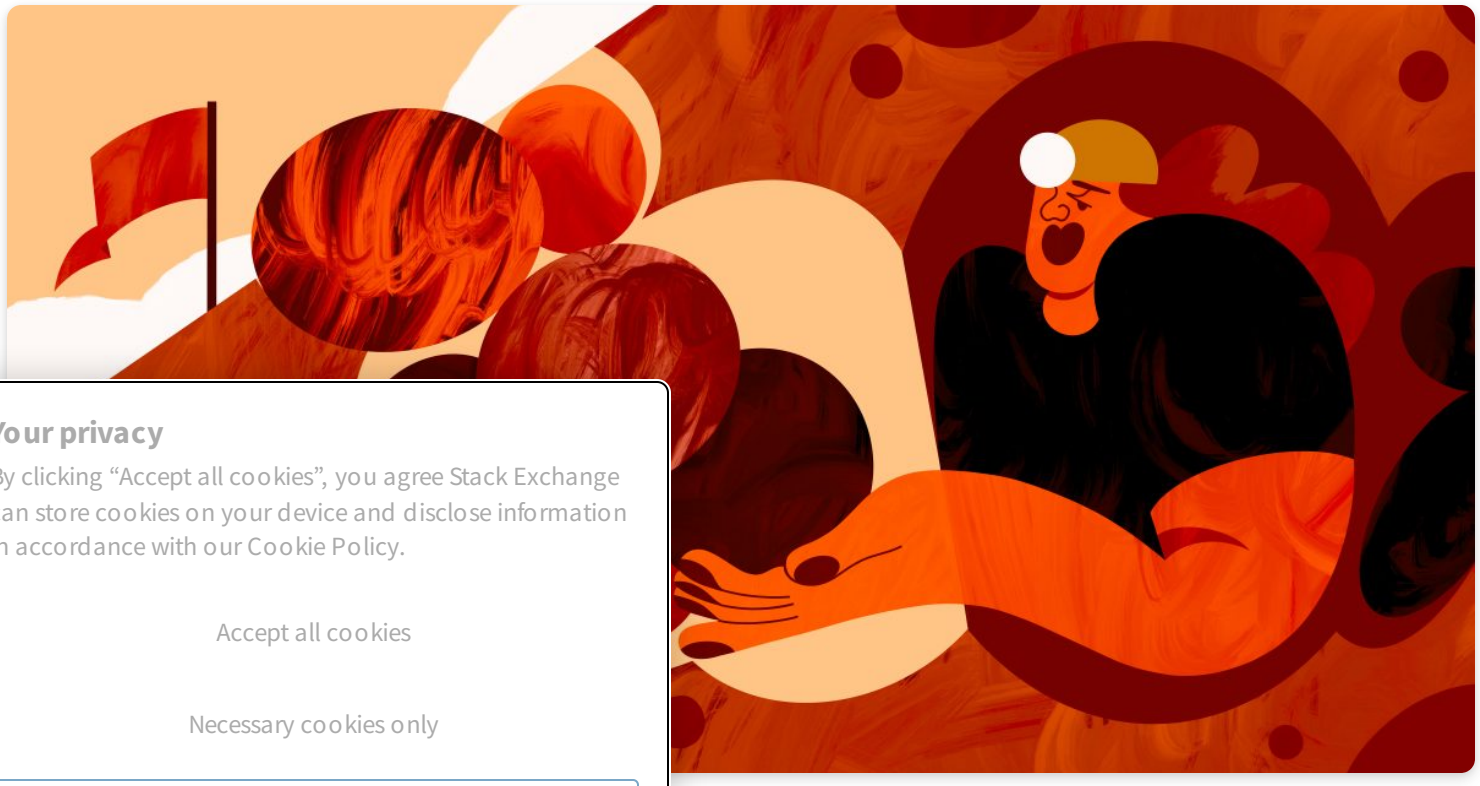
Essays, opinions, and advice on the act of computer programming from Stack Overflow.

code-for-a-living FEBRUARY 27, 2023

Stop saying “technical debt”

Everyone who says "tech debt" assumes they know what we’re all talking about, but their individual pictures differ quite a bit.

Chelsea Troy



We were supposed to release this feature three weeks ago.

One developer got caught in a framework update. Another got stuck reorganizing the feature flags. A third needed to spelunk a long-abandoned repository to initiate the database changes. The team is underwater. Every feature release will feel like this until we get a few weeks to dig ourselves out of tech debt. We have no idea how to even get the business to consider that.

Does this sound familiar? It's a disheartening conversation.

But we often predispose ourselves to this situation. How? We try to get businesspeople, designers, product managers, and engineers onto the same page by using the phrase "tech debt." But that phrase puts us on completely different pages.

Ask someone in tech if they've heard of tech debt, and they're likely to respond with a knowing sigh. Now ask them what it is. Do this ten times, I dare you. How many different answers do you get? Three? Four? Seven?

Everybody associates the term with a *feeling*—frustration, usually—but they don't have a precise idea of where that feeling comes from. So they slap the term onto whatever happens to bother or frighten them. Designers say it means the design can't look the way they planned it. Product folks lament how it means they lose three weeks and get no features out of the deal. Engineers? Their answers vary the most, but often they've got something to say about "bad code." We'll return to why "tech debt equals bad code" is such a scourge, but first we have to address the effect of a bunch of different people defining the same term differently in the first place.

Here's the effect: the minute we trot out the term "tech debt," everyone is upset but no one is listening. Each conversant assumes they know what we're all talking about, but their individual pictures differ quite a bit. It sounds to the business like the engineers are asking for three weeks free from the obligation to release any features. They remember the last time they granted those weeks: within a month the team was underwater again. When businesspeople don't want to grant a "tech debt week" because they saw with their own eyeballs that the last one improved the team's capacity zero percent, *how can we expect* them to grant us another one with alacrity?

logy. And we can find that terminology by dissecting

bad code

into traps. It allows us to assume that the prior
charitable, but fine, until we realize that there was
s constraint explains the loathsome characteristics of this
n genius solution.

I once worked on a team that complained ad infinitum that customer information required a query that drew from two different tables. The team assumed that the structure remained in place because of inertia or because changing the database structure had backward compatibility implications. After spending a non-negligible amount of time bashing the database design and dreaming up ways to fix it, the team learned that their plan was…illegal. For privacy reasons in their industry, it's illegal to store these two particular pieces of personally identifying data in the same table. Luckily, a product manager happened to mention the situation to a lawyer at the company before the engineering team got very far, or it might have been a showstopping compliance issue.

Equating tech debt to bad code also allows us to believe that if we just write good enough code, we won't have tech debt. So we don't spend time eliminating any. There's no need to revisit, document, or test this code; it's just *that good*. A year later, we're back where we started. Whoops.

Equating tech debt to bad code also allows us to conflate "this code doesn't match my personal preferences" with "this code is a problem"—which, again, is fine, until we're under a time constraint. We spend "tech debt week" doing our pet refactors instead of actually fixing anything. Engineers love tech debt week because they get to chase down their personal bugaboos. The thing is, those bugaboos rarely intersect with the code's most pressing maintenance challenges. So when each engineer finishes their gang-of-four-fueled refactoring bender, the code is no easier to work in than it was before: it's just different, so no one besides the refactorer knows it as well anymore. Fantastic. A+. No notes.

In all seriousness, this is a *huge* reason that spending three weeks paying down tech debt, carte blanche, often does little or nothing for the team's velocity after those weeks have ended.

To fix these problems, choose something measurable to evaluate the quality of the system. My recommendation: maintenance load. How much time and effort are developers spending on tasks that *are not* adding features or removing features? We can talk to folks outside the engineering team about that number. If we have six developers but half of our work is maintenance work, then our feature plan can only assume three developers. Business people think of engineers as expensive, so this framing motivates them to help us decrease our maintenance load.

We can also track that number and determine how fast it grows over time. The faster our maintenance
t. Zero growth means that we can always maintain the
ering team.

With good code stewardship practices. We rarely reward,
hat we do feature development skills. But code
vering context from code, and designing for future
hat hums along for a decade or more and a team that
ankruptcy, rewrites, and despair.

The Holy Grail? *Negative* maintenance load growth: the kind of growth that makes our code *more* maintainable over time instead of less. The Grail requires even more of the team than a healthy quotidian code stewardship routine. It requires us to look at individual maintenance tasks, track their origins, and address those problems at the source. These chores, backed by empirical evidence, give us something concrete to discuss in meetings about tech debt.

Are we performing lots of routine library or framework updates right now? Maybe we need to explicitly set aside time on a recurring basis to complete those. The more these pile up, the harder it becomes to debug the interactions between releases of different libraries. And the less programmers perform these, the more out of practice they remain—which makes the update rockier and more painful at the last possible second, when the update becomes mandatory.

Are we reaching into abandoned repositories and figuring out how to make a change? Maybe we need to devote effort to recapturing information about how those repositories work. It's common for development to become much harder after some seminal team member leaves because it turns out they knew a lot of critical information that wasn't written down or organized anywhere. I call this a context loss event, and we have no idea how maintainable a code base really is until it survives one of these. After a context loss event, developers need to proactively assess and repair the damage done to the team's shared knowledge before unfamiliar parts of the code base become dark and scary.

Are we constantly working around an architectural choice from the past based on assumptions about our domain that are no longer true? Maybe we need to create and prioritize a ticket for rearchitecting that. A resilient code design considers what types of changes the team expects to make in the future, and it allocates flexibility in those parts of the code. As with any task that involves predicting the future, sometimes we get it wrong. In those cases, we might need to change our design. That may require a dedicated effort.

How do we identify and prioritize chores like these? I have a whole self-paced online course about that, but even focusing on maintenance load in units of specific chores, rather than a unitary towering thundercloud of "tech debt," gives us a better place to start addressing it.

We *want* feature development to feel smooth and effortless. The longer we put off maintenance work, nent will be. Rather than sweep all of those tasks under a for time to deal with it as one unit, we can track what velopment to take longer, measure them in terms of the nd then negotiate their completion as individual tasks y. We're no longer framing them as an opaque and early circumscribed *investments* in our ability to produce s on the same page. It also increases the likelihood that:

do the maintenance work

oing the maintenance work

- The maintenance work, having been selected from the real reasons that feature development slowed down, will actually improve the feature development experience for the future

And that makes the conversation about tech debt a lot less disheartening; It might even make it hopeful.

Tags: code maintenance, technical debt

The Stack Overflow Podcast is a weekly conversation about working in software development, learning to code, and the art and culture of computer programming.

Related

code-for-a-living FEBRUARY 16, 2023

Monitoring debt builds up faster than software teams can pay it off

Today, it’s easier than ever for a team to monitor software in production. But it's also easy to build up a lot of tech debt around monitoring.

Jean Yang

20 Comments

Jason C 27 Feb 23 at 10:51

I’ve never heard or seen this phrase before this blog article. I think I’m going to start saying it now.

Reply

Aaron Newman 27 Feb 23 at 1:40

I’ve always used ‘Technical Debt’ to mean dependencies on technology that are obsolete or deprecated. I’ve never heard it to mean ‘bad code’. It could mean old code doesn’t handle new use cases. Maybe some good practices can minimize the amount of rework required to update to new technologies. Eliminating it altogether seems akin to predicting the future.

Handling technical debt is required for complex systems that work over long periods of time, if those systems are not to become obsolete themselves. You can stop using the phrase but that won’t fix any problems on the ground.

Reply

27 Feb 23 at 5:24

. and stop using it when talking to non programmers, the
OOP and asynchronous functions.

Kevin 28 Feb 23 at 1:16

It’s called technical debt because it’s analogous to financial debt. Business owners need to understand and reckon with it.

Reply

Giorgio Migliaccio 28 Feb 23 at 2:29

Exactly, technical debt, whatever name you would give it, actually exists, and ‘bad code’ perhaps is a phrase we shouldn’t use, but everything ever developed is circumstantial and contextual.
For urgent release reasons, some shortcuts in the code were willingly added, and ‘should be addressed’ in the future for example. Or, as you mentioned, quite some dependencies on libraries are pretty outdated and more often than not, this might bring its fair share of effort needed to put in updating your code.
Twice I did and upgrade of ‘outdated’ .NET 4.5 code to the newer .NET Core 3. for a huge business application this took us 4 FTE’s more than 6 months before it was working fine. But also sometimes a ‘simple’ dependency, and in its slipstream a whole lot transitive dependencies might bring a whole lot of reworking.
But this can’t be prevented in any thinkable way, this is just part of the job and technology.
If this should be addressed in a ‘3-week tech debt’ period on the other hand, is also something more mature tech companies don’t tend to do (anymore).
They just identify these ‘problem areas’ and place it as stories on the backlog, so it can be estimated and planned in.
They’re just new work items needing to be addressed, nothing to be ashamed of, nothing to be disappointed in, just part of the nature of development.

Reply

Michael P 28 Feb 23 at 6:27

ance generally nor outdated dependencies. It’s the
nce over sustainable design:

27 Feb 23 at 3:15

of a conversation, not the end of one.

I am reminded of the phrase “premature optimization,” sometimes used as an excuse for refusing to consider performance in your software designs. Like “technical debt,” this phrase is meant to start a conversation, not end it. Despite these phrases having the potential for abuse, there are genuine reasons why you might want to invoke them.

The considerate software engineer, having used one of these phrases in a conversation, must then explain specifically what they mean, starting with the legitimate premise that the conditions described by either phrase can cost the organization time and money if they are not managed properly.

The phrase “code smell” doesn’t have any technical meaning at all, but everyone who uses it knows what it means: a sense of uneasiness. The problems arise when inexperienced developers ascribe more meaning to these phrases than they should, or confer more authority to these phrases than they really have. There’s no “official” basis for a “code smell,” any more than you can definitively identify “technical debt” or “premature optimization” without more analysis.

Reply

Stephen Boesch 27 Feb 23 at 4:15

“Technical debt” has fairly clear meaning in established software / data engineering teams. *Weak testing *Weak CI/CD *Limited commenting *Limited or no design/other technical docs *Refactoring should be done (eg violating DRY). Apparently in other front-end or other types of teams it is more nebulous.

Reply

Andrew Cowenhoven 27 Feb 23 at 4:38

No. Don’t stop using the term. It is not a meaningless, hackneyed phrase. It is a technical term. Just, as you always should, consider your audience before you use any technical term. And, of course, understand it yourself. http://wiki.c2.com/?WardExplainsDebtMetaphor

Reply

27 Feb 23 at 4:57

oblems”.

ing during races because your shoes are untied, you’re ie your shoes, ideally before the next race begins. 🙂

et peeve, or just a misunderstanding of how the code is bout it, is a whole different discussion, and that hether you continue to use the term “technical debt” or

Reply

Daniel Craig 27 Feb 23 at 5:11

Saying “Technical Debt” considered harmful

Reply

Joshua 27 Feb 23 at 5:22

Better idea: stop playing name games.

Reply

Victor Williams Stafusa da Silva 27 Feb 23 at 7:08

Technical debts are things the TO DOs scattered around the code. If there is no TO DO and the intent is just to make it more elegant, then it is seldom worth the trouble. However, most of times, those TO DOs are things that slow down the development team, limits the growth of the business, rises costs and some of them are even timed-bomb waiting to be either defused or detonated.

This frequently have to do with bad code, but it is much more than that, and it is often to growth pains or technology changes and requirements changes. Although technical debts frequently involves bad code, equating them is just incorrect.

Moreover, having to always pay the technical debt and having the impression that we eventually are all back to square one is part of the development process. It is disheartening, but it is unavoidable and even necessary. As I say, “Technical debts are billed with high interest fees” and refusing to pay them only makes them growing larger and larger until they eventually become unpayable.

If the business people don’t understand that things left incomplete, undone or no longer satisfactory needs to be maintained, then the business itself is doomed. I already saw several business go bankrupt e because the focus was just to add feature over feature stly done. The reason is because eventually the technical added without introducing a lot of bugs that degrades all ncluded. Eventually the system enters into a state that and be productive with it anymore and neither the ed. All the effort goes into trying to kill hydra-bugs (cut he business people only complains that no feature ever lapse. All of this could be prevented if people understood he technical debts and that new features can’t be lled.

Releasing new features without a solid code base to support them is the same as building another store in the building without reinforcing the basement and the pillars of all the stores below. Business people who can't understand this simply are not worth of their business.

Everyone will always work underwater because the only dry land is a desert wasteland. And if being underwater is bad, once you are in dry land, you are out of the business. Business people and developers must know that there are only three options: to swim, to drown or to die in the desert wasteland. Three weeks of refactor isn't enough and will never be. No amount of time and effort is enough, but this is not a reason to not invest any effort or time, quite the opposite actually.

Technical debts even have something to do with the second law of thermodynamics: The entropy of an isolated system only grows. To keep the entropy under control, the only way is to transfer it to somewhere else, and that somewhere else is the work, effort and time of the development team.

Reply

J. B. Rainsberger 28 Feb 23 at 5:24

I think "maintenance load" provides an interesting way to preserve the original intent of Technical Debt (per Ward Cunningham) without obscuring the meaning too much. Part of the problem seems to lie in this: people forget that Technical Debt is an unavoidable consequence of the choice to build and deliver features incrementally: we allow design to be simpler now in order to defer the part of the investment in design that just isn't needed until we build that feature later. I see Technical Debt as a choice and a strategy, but "maintenance load" seems to include both Technical Debt and Cutting Corners.

Sadly, the Euphemism Treadmill always wins, so if "maintenance load" becomes popular, someone will write an article about why we need to stop saying it…. I predict around mid-2027. Maybe earlier.

Same as it ever was. Don't give up!

Reply

Martin Ba 28 Feb 23 at 5:59

bt, e.g.: https://youtu.be/piUesxuZkIQ?t=163 (Refactoring
2022)

28 Feb 23 at 9:01

a previous team to remove the stigma from the phrase.
choice between a tactical solution and a slower but more
ice based on a number of factors, including time or
commercial pressures, technical knowledge, value placed on stability/performance/security/etc.

Everyone owns this trade-off, and if the tactical option is chosen, it's on the promise to repay this debt. There's a responsibility to be "better lenders" and not always choose the tactical option, but that responsibility never sits solely with the technical members of the team.

Reply

Felix Rabinovich 28 Feb 23 at 11:46

Interesting… most of the people I talk to have pretty similar definition of technical debt. For example, we would like to refactor the design of the database to take advantage of new capabilities (say, many-to-many relationship in Entity Framework or Time Series tables in SQL Server) – but we have interfaces or reports that will make it prohibitively expensive.

But there is one statement here that makes me question credibility of the author. "For privacy reasons in their industry, it's illegal to store these two particular pieces of personally identifying data in the same table". I worked with many standards (HIPPAA, GDPR, most recently, CJIS) – and I am yet to see any that specifies anything about the *tables*. When you make up stuff like that, it makes me question other premises of the post!

Reply

Kevin 28 Feb 23 at 1:14

Stop trying to normalize mediocrity.

Reply

Me 28 Feb 23 at 2:15

technical debt is mostly used from people who are not earning their living with programming but with telling the bosses that everything is wrong and they could help to make it better. But never ever by doing it better.

d the code is, just to sell.

he gift from heaven to the world of programming.

28 Feb 23 at 3:02

dge Limit" — i.e., the number of times I'd make a quick
d it needed to be re-written properly. I think that's a

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Post Comment