gorrion

What if I told you **not all tech debt is bad?**

Many entrepreneurs whose business depends on a software product dread the words “technical debt.” They know that it effectively means developers nagging them about fixing “critical” issues (a critical issue being, seemingly, absolutely everything), spending time and money on invisible fixes, and slowing down the release of necessary new features.

Those who don’t dread technical debt, simply don’t know what it is. **Yet.**

So how come we can so confidently tell you that technical debt isn’t necessarily a bad thing?

Because at Gorrion we know from experience of developing over 200 projects that when debt is strategic, informed, and properly managed, it is the best tool in your growth acceleration toolbox.

In this ebook we explain what technical debt is and isn’t, why it’s good and why it can be bad, how to use it strategically and stay on top of it, and finally **– what to do when you’re in too deep.**

Table of contents

What is technical debt?

- It's a loan
- It's not a surety bond

When is technical debt good?

- Throwaway prototypes
- MVPs
- New features

When and why is technical debt bad?

- The consequences
- Resource consumption
- Deadlock
- Premature optimization

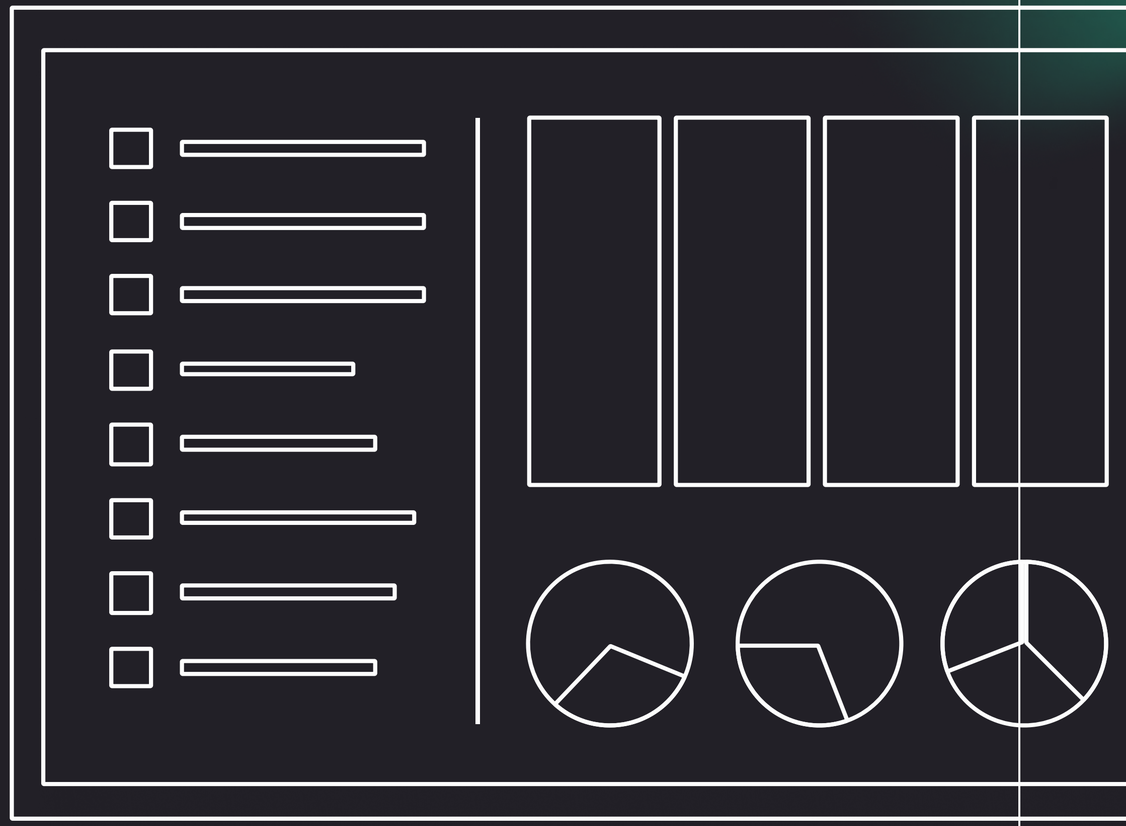
Why it’s so hard to pay off

- “Don’t fix what ain’t broken”
- That’s a you problem
- Fictitious force
- Cutting costs

How to manage technical debt – the right way

- Awareness
- Assessment
- Sustainable development
- Rotational focus

Contact



What is technical debt?

Our Chief Innovation Officer, Dominik Guzy, explains technical debt this way:

> Tech debt happens when you make shortcuts and rushed decisions in the code, which builds up to the point where you must return to it.

Imagine that you can develop a feature in your app in two ways: one is “quick and dirty,” the other one is “proper” (and more time consuming). If you choose the first route, you have just incurred technical debt. The feature will work now, but there’s no guarantee it will work in the future or that it won’t interfere with any further deployments. If it breaks or affects your product in any other way, you will have to fix it, i.e. pay off your debt.

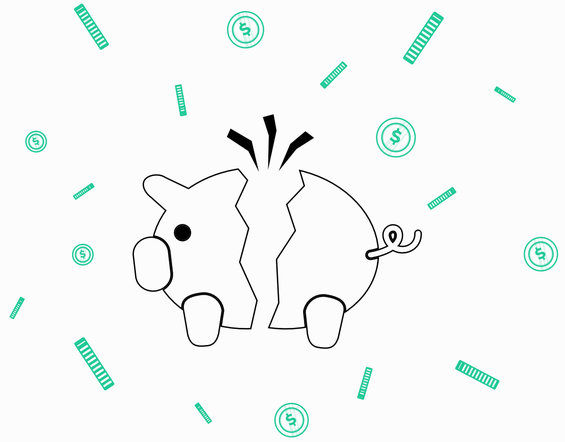
Kent Beck – the creator of extreme programming
– defines three stages of software development:

make it **run** → make it **fast** → make it **right**

Making software **run** means simply writing code that works, i.e. can be read by computers.
Making it **right** means writing code that is stable and polished to an acceptable standard.
Making it **fast** means optimizing it to a point that it performs better and scales
(this encompasses testability and extensibility of the code).

2024

It's a loan



You may have already picked up on financial lingo and banking metaphors. That was the intention of Ward Cunningham, who coined this term. The easiest way to think of technical debt and to manage it is by looking at it as if it were financial debt.

Yes, many people will say “all debt is bad,” but even more people will say it’s a great way to leverage yourself, and some will say it’s the only way to dig themselves out of a hole. Everyone knows debts must be paid off and everyone understands that they come with an interest. We can probably also all agree that taking more debt to pay off a previous debt is a bad idea and must be avoided at all cost. All these things are true both when we speak of money and when we speak of code. The difference is, with technical debt you’re effectively borrowing time and not money itself.

It's not a surety bond

Without going into philosophical debates, it's important to note what technical debt is not. **Technical debt is not simply "bad code."**

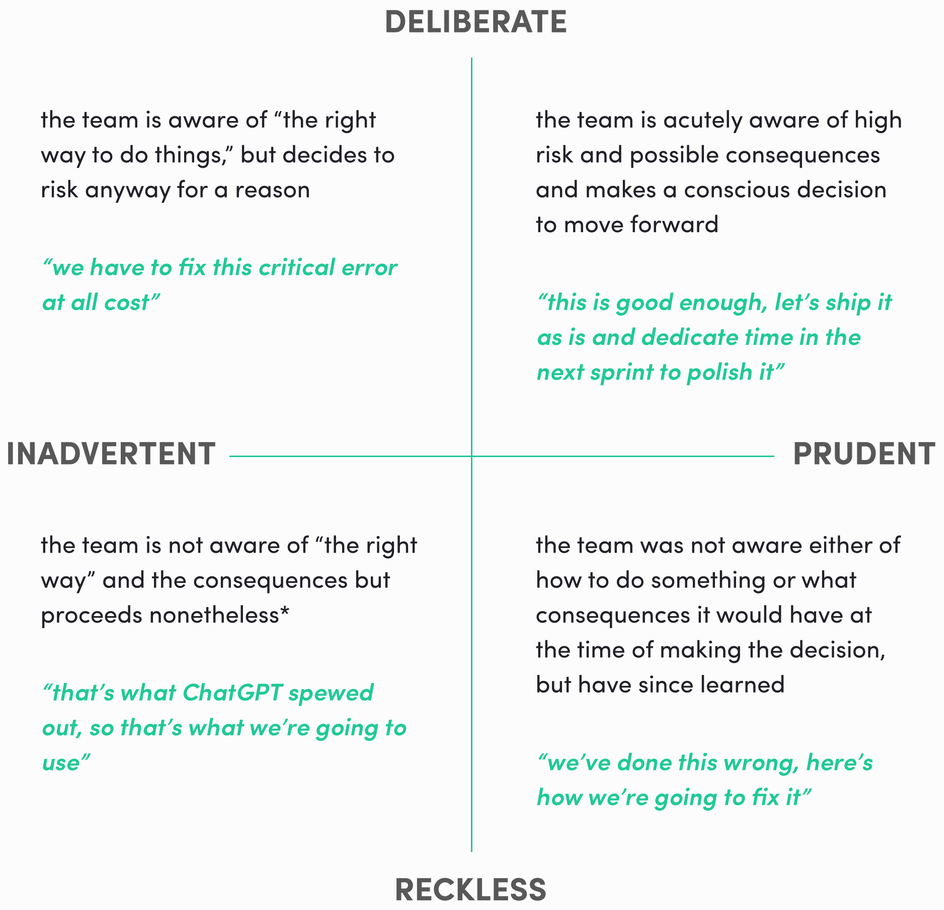
It's a different thing when developers consciously make short term sacrifices, take shortcuts, or find workarounds due to project constraints, but they are aware of the consequences and intend to fix their code in the future. Imagine your app going down due to a critical error that must be fixed immediately by any means necessary so that the software can be operational again. The fix is dirty, but it works for now. You've just taken a loan to stay afloat.

It's an entirely different thing when developers don't know (or don't care) what they're doing, but they're doing it anyway. Imagine you hire a team of unskilled developers who don't have experience in your field or technology of choice, and they just trudge ahead creating shoddy software that looks like it's working for now. **That's not a loan** – you've just signed a surety bond for a very unreliable debtor and there will come a time that you have to pay it, whether you like it or not.

We mention this because, historically, Gorrion has occasionally taken over projects where this happened. The code was so bad that it was no longer feasible to add new features without performing near-impossible **“circus tricks”** that would take a whole lot of unnecessary time and money. There was nothing else to do, but to rewrite the code from scratch In instances such as these, it’s difficult to speak of **“technical debt”** and not a **“botched job.”**

It’s important to note, though, that not everyone would agree. **Martin Fowler – an expert on the subject of technical debt and a software developer** himself – came up with the idea of “technical debt quadrants” that classify debt based on its intentionality and rationality.

It looks something like this:



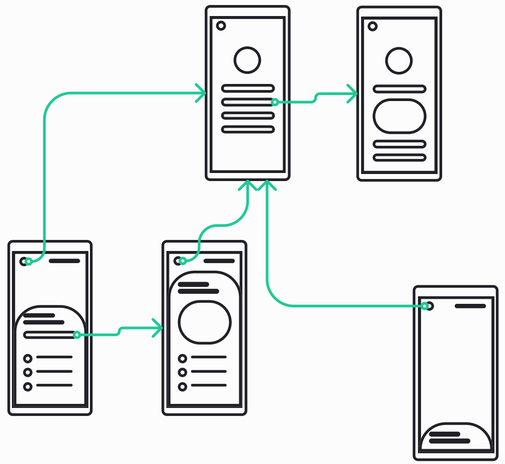
*We will not be covering instances of inadvertent and reckless debt in this ebook

When is technical debt good?

Now that we're all on the same page as to what technical debt is, we can finally answer this seemingly rhetorical question: can technical debt be good?

And we at Gorrion say: **yes, it very much can!** There are instances where you'd want to welcome it as a friend.
This is especially true for projects (or rather: stages of projects) where you need to prioritize speed over quality.

Throwaway prototypes

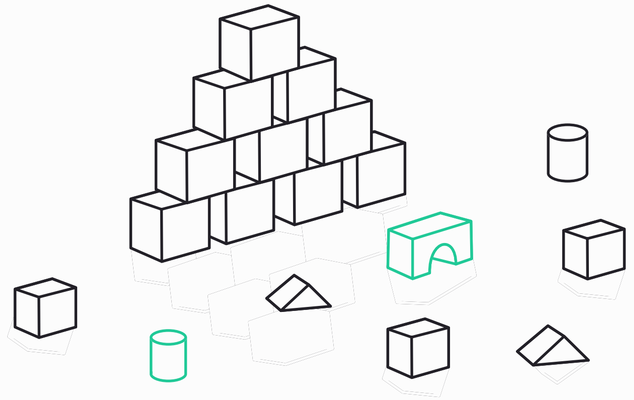


You create a prototype or a PoC (proof of concept) to find a solution that simply shows a problem can be solved – nothing less, nothing more. It doesn't need the best performance, well-architected code, or massive test code coverage.

The idea behind prototypes and PoCs is that they should be as fast and as inexpensive as possible. They make for a very basic scaffolding to the final product so if you have to knock them down and start again, there's very little harm done.

Tech debt in throwaway prototypes is like taking a zero-interest loan – doesn't really cost you a thing (as long as you pay it off on time) but it's a quick way to get what you want.

MVPs



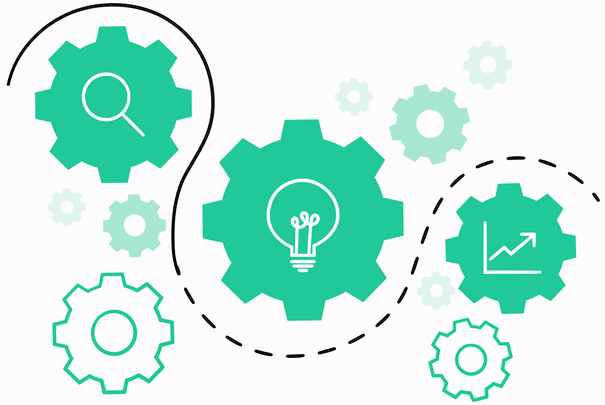
Then we have the case of startups and MVPs (minimum viable products). The situation here is somewhat similar: an MVP is meant to prove something (i.e. market demand) and, as the name suggests, it provides the bare minimum.

Oftentimes coming up with an MVP requires some shortcuts. This can either be because of scarce resources – a gripe startups only know so well! – or time pressure. In any case, tech debt here is like a mortgage. It's frequently a "necessary evil" to get things going and a long-term burden, however – if taken responsibly and paid off wisely – it's a great leverage for a lifetime investment.

Take Uber as an example.

Before it took over the world by storm, it was just a startup. In 2016 they already had an MVP and a successful presence in several cities in the US. They moved fast and validated their idea but they were also conscious of their product's limitations. That's when they decided to hire Gergely Orosz, the author of The Software Engineer's Guidebook, to "rewrite the Uber app." They found the right time and the right people to pay the debt, which allowed them to scale and grow into a global phenomenon.

New features



Not only new businesses are allowed to go into hock, though! Even mature organizations can – and perhaps even should – run up some technical debt in certain situations.

Imagine you have a mature product, but you're looking into new niches or business opportunities. There's already some traction there, perhaps even some competition, so speed is critical – to be successful, you need to be the first to the market. **However, you don't really want to put all of your chips on the table** – you don't know if there will be any interest in this new feature of yours or you're not sure this market is really right for you.

In this case, even though you have a mature product, you treat the new feature very much like an MVP. In order to maintain speed and validate your idea quickly, you use technical debt as a leverage. If it pays off – great, if it doesn't – no harm done!

When and why is technical debt bad?

If technical debt is so good then why does it get such a bad rep?

Why isn’t everybody talking about it?

2024

Perhaps you should think of it the other way round.

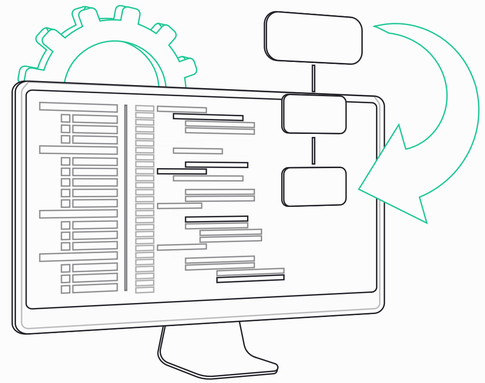
Everyone has at least some degree of technical debt, the thing is – not in a good sense, like in the examples above. Technical debt has become a sad reality of all software development, unavoidable in all projects due to pressures from the business, inadequate resources, and other factors. Companies will simply push for new features to generate more revenue or they will “optimize” resources to squeeze out more from the revenue they already have. **Engineers are then forced to find shortcuts to deliver on the expectations.**

It's therefore important for businesses to understand the consequences of technical debt and how it affects not just the code, not just the product, **but the business as a whole.**

The consequences

If you've read this far, congratulations, we've just reached perhaps the most important part. Whether good or bad, intentional or not, tech debt always has consequences. To use Martin Fowler's terminology, being unaware of those would be imprudent.

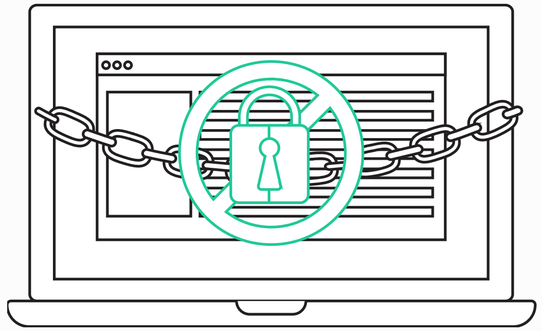
Resource consumption



First and foremost, paying off tech debt takes time and resources. After all, if the right solution was the easiest and the quickest to develop, it would have been implemented from the start. Trey Huffine even defined technical debt as “any code added now that will take more work to fix at a later time.” Going back to fix things requires the team to take a step back, which will slow down or even halt new development entirely.

From a business perspective, it’s important to remember you’re slowing down so that you can speed up later. We see this regularly – clients are hesitant to allocate time to fixes, prioritize maintenance over development, do any work that doesn’t show anything new. This is a shortsighted mistake. While it does take time and resources to pay off technical debt, this will provide returns in easier time for developers, better product, and more growth opportunities in the future. Not to mention consequences of neglecting technical debt, which are far worse and more symptomatic than temporary slowdowns.

Deadlock

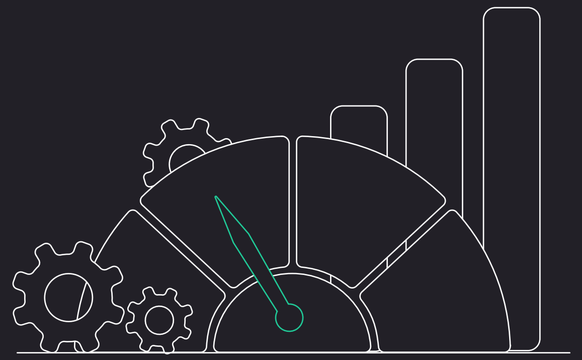


As we've mentioned before, all debts must be paid. With technical debt, if you don't do it yourself, your code will do it for you. If left unattended, tech debt builds up to a point where it causes deadlock.

A "deadlock" is a technical term for when two (or more) processes are running but neither is able to finish because they need more resources that are blocked by the other process. This creates an endless loop or, in other words, a deadlock. While you needn't understand the technicalities, it's easy to understand that, without optimization, at some point the code will become too complex to scale or even to perform at all.

At this point tech debt is not just a problem for developers, it's a problem for the entire business, as the application can't grow anymore and it may in fact slow down, show an increased number of bugs, or shut down altogether.

Premature optimization



It may seem incredulous but technical debt can be a problem even when there isn't any technical debt. This is called premature optimization and it occurs when the code is optimized too early, when priorities are misplaced or when complexity and abstractions are addressed over necessities.

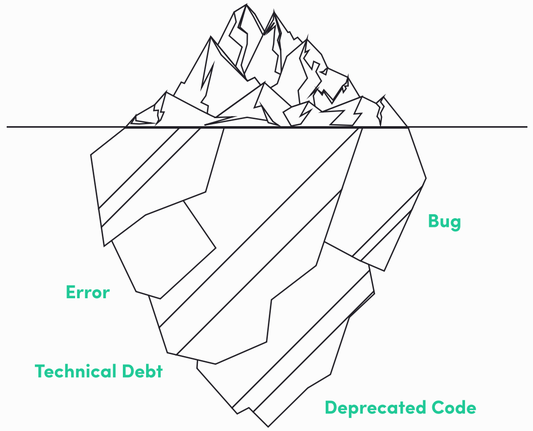
Premature optimization isn't necessarily a problem for developers. In fact, some particularly opinionated developers argue that all tech debt is bad and "no debt" should be the default state. However, it's a huge problem for the business. It means that resources were poorly allocated (and therefore lost), priorities weren't communicated clearly, and the company lost profit and/or opportunities as a consequence.

Why it’s so hard to pay off

We’ve already mentioned the struggle we sometimes experience personally when working with clients that is, frankly, prevalent in software development everywhere – the resistance from business stakeholders.
Let’s face it, to non-technical people technical debt is much less visible than actual financial debt that they know and understand could sink their company. And technical teams demanding technical debt be paid off look far less threatening than creditors and debt collectors. And so, technical debt is deprioritized despite being as dangerous as (if not more dangerous than) financial debt.

From our experience, these are some of the most common “blinkers” businesses wear when it comes to technical debt.

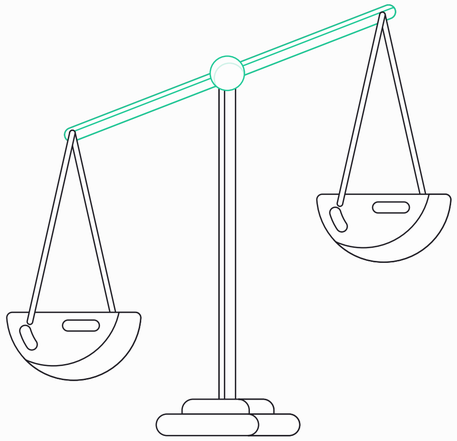
“Don’t fix what ain’t broken”



If the software is working, surely it means everything’s ok with the code as well, right…? **Well, wrong.**

Businesses are often ignorant about existing or potential issues with their product simply because of a poor understanding of the more technical aspects. And, admittedly, it is hard to be worried about things we can’t see and/or understand, like climate change and technical debt. So, following the old adage of “don’t fix what ain’t broken,” they aren’t really invested in improving the code unless it is actually necessary (i.e. the app breaks).

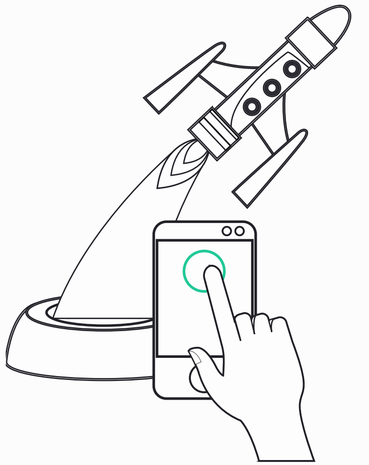
That's a *you* problem



So we've established that business stakeholders rarely understand the implications of technical debt, but it's equally important to point out they are also rarely affected by it. Conversely, developers both understand it very well and find it significantly more painful. Code that's not optimized is harder to maintain and to integrate, resulting in more workload and harder times for developers.

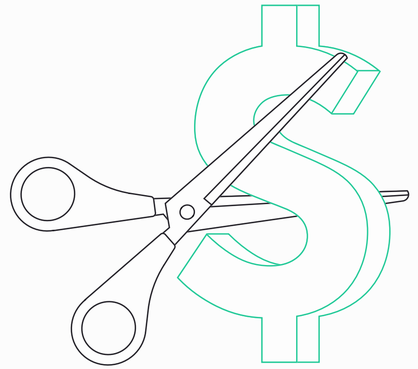
Without certain "empathy" for their work (or at least for how their wages affect the payroll) it's hard to agree what goals to tackle next and to prioritize things that would make that workload easier, even if – ultimately – it's for the benefit of the entire business.

Fictitious force



Let’s face it, “we’ve developed this new thing” will always sound more appealing to business leaders than “we’ve fixed this old thing you didn’t know was broken.” That’s somewhat understandable, because new features mean new opportunities to sell, and this means more revenue and more opportunities to grow. Which is why, when the product starts to get some traction on the market, it’s very hard for the business to slow down (much less stop altogether!) and potentially disturb that growing trend. This “fictitious force” that drives business leaders and the willingness to please them that motivates other stakeholders leads them to neglect technical debt in pursuit of novelty.

Cutting costs



Finally, there's an old Polish saying that goes something like this: if you don't understand what the motive is then the motive is always money. So if you don't know why businesses don't pay off technical debt, it's simply because it's cheaper that way. After all, as we've already established, fixing issues takes a significant amount of time and effort and affects progress along the way.

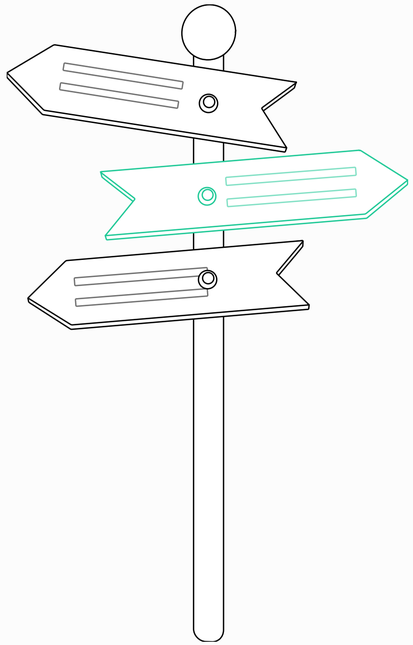
There's a flipside to this type of thinking, though, because all is fine and well as long as the code is still functional. However, without paying off technical debt, at some point it won't be so fine, and you will need to refactor the entire application. That, our friend, is not at all cheaper... to say the least.

How to manage technical debt – the right way

So too little debt is bad and too much debt is bad, and technical debt can be both good and disastrous...
If we've learned anything from this ebook, it's this: moderation in all things.

So how should you manage your technical debt so that it works in your favor and not against you?
Here are some strategies.

Awareness



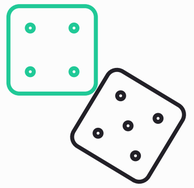
The first step towards fighting off technical debt is... realizing that it exists. As we've said before, lack of technical understanding often leads business roles to ignore the problem, which results in teams being unable to integrate new features smoothly or even at all.

To prevent that, you need to make a conscious decision to keep tabs on your technical debt and use it strategically. This requires you not just to talk to developers, but also to make them an integral part of business decision making process.

Assessment

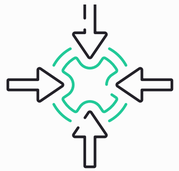
An important element of awareness is assessment. Not all technical debt was created equal, meaning – not all of it is equally painful or dangerous for your business.

A relatively easy way to assess your technical debt is based on the SQALE method (which, in itself, is very much not relatively easy). Every issue reported by the development team should be scored from 1 to 5 (1 being “not much” and 5 – “a lot”) on three scales:



probability

how likely is it that this issue will ever cause any problems?



impact

if problems do occur, to what degree will they affect the business – what’s the worst case scenario?

cost

how much time and resources will we have to invest in fixing this issue?

Issues with the highest score on all scales naturally need to be prioritized first.

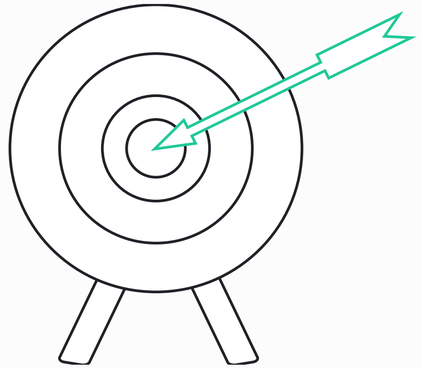
Sustainable development

Once you are aware of your tech debt and have it assessed, there are some ways to manage it without disrupting the whole development cycle.

For example, you can decide to allocate around 10-20% of each sprint to addressing code refactoring, testing, and technical debt issues. However, this time should be used for proactive maintenance rather than reacting to problems when they become critical. By regularly scheduling this maintenance, teams can keep a healthy codebase, avoid bottlenecks, and ensure that the system remains adaptable to future changes.

Importantly, the specific percentage allocated for maintenance might vary depending on the project, team's experience, product complexity, and most importantly, the amount of technical debt.

Rotational focus



Alternatively, you can decide to rotate your team's focus every few sprints and dedicate a full sprint (or cycle) exclusively to tackling technical debt and reducing system complexity.

This allows your team to periodically refocus and refine the system architecture which reduces technical debt without causing long-term development delays.

And if all else fails...?

If you're reading this because you already have an existing product and are struggling with unmanaged technical debt, but the solutions we've provided don't seem to be enough, you need to face the possibility that you're already in too deep. In these instances, there's often nothing else to do but to perform a complete refactoring of the software. The sooner, the better.

If you suspect this might be the case and you don't know how to approach this problem, we can help. We've worked with multiple products that seemed like they're beyond repair when they first arrived on our doorstep, and yet they are now operational and successful.

Let's see if we can help you, too!

gorrion

Contact us

leo.baz@gorrion.io

+48 511 535 131

www.gorrion.io

www.clutch.co/profile/gorrion

www.linkedin.com/company/gorrion