The #1 Reason Your Software

is sucking the life out of your development team and 3 common mistakes to avoid

Helen Abell

3 common mistakes to avoid

Introduction

Are you constantly being asked to deliver more with less?

Are you juggling a lot of requirements with tight deadlines for multiple customers?

Are you struggling to get your team on board with all this?

If so, you're not alone.

Customers want more, bigger, better, faster, more secure features in your products, and your company wants to give it to them, at a profit. But how do you achieve this when your team is struggling with what's already on their plate and your projects are being derailed by changing priorities and unexpected critical bugs.

Are your developers context switching all the time trying to keep on top of urgent bugs and changing requirements?

Are your testers spending a lot of time manually regression testing the same features release after release?

Do new features take increasingly longer to develop because unexpected bugs or performance issues arise whenever your developers make changes to your software?

If your software company is successfully selling your product to customers, who are constantly coming up with new things they need it to do, and your application is more than 5 years old, and the number of people who have worked on it over that time has grown and changed, then you will have started to experience some growing pains within your software.

You will probably have noticed that you make a development plan or product roadmap for the year ahead, but less than one month later you need to change it completely because of the urgent issues reported by customers this month that have derailed your plan completely.

You've tried to get estimates from the developers for how long the issues will take to fix, but they either say something like it could be 10 minutes or 3 weeks and if they spend the time finding out what the problem is, which they need to do before they can tell you how long it will take, they might as well fix it while they're in there, or, if they're savvy and know the business will try to hold them to whatever they guess at, they will refuse to give you an answer in numerical form at all. In this environment it can feel impossible to plan or to know what commitments to make to the business and customers.

What Does Not Work

Incentivising Your Team With Metrics.

Since I first started programming in the 1980's managers have been trying to measure developer performance using metrics, inspired by the quote "If you can't measure it, you can't manage it." commonly mis-attributed to Peter Drucker [1].

Most development managers these days now know that the original bad metric - measuring **lines of code** to measure developer productivity is not just stupid, it's dangerous. Better code is often less code. And lines of code are polluted with noise that has no bearing on developer productivity such as comments. Incentivising the production of volume of code is a fast route to massive technical debt, which will eventually slow your team down to a crawl.

Commit count is less noisy to be sure, but still, incentivising your development team for large numbers of commits is going to encourage developers to check code changes in before adequately testing, and before finishing a piece. This is likely to have an adverse effect on other team members who are working with a constantly moving target code-base that will have a higher number of errors in it. Number of commits is clearly a pointless metric for incentivising developers as it tells you nothing about the content or quality of the code being committed.

Shipping Velocity or Issues Resolved, is at least vaguely connected to a meaningful outcome, but what does it incentivise? Well for a start it penalises tackling difficult problems that are going to take longer to solve, in preference for simple or trivial issues. It encourages quick and dirty fixes without regard for side effects, because you get points for each time you fix a side effect. It discourages well thought through design, impact analysis and the creation of unit tests that slow things down in the short term but speed things up in the long term.

Code Churn minimization is hard to measure consistently. This measures changes to a local area of code, but the biggest causes of code churn include unclear requirements, indecisive stakeholders, so what does it incentivise? Not starting until the requirements are 100% clear and known. Which, as anyone who remembers the bad old days of waterfall development will know, is never.

As well as sloppy coding, high code churn is caused by useful things, like prototyping, tackling difficult problems, polishing, refactoring, commenting, standardizing, generally making your codebase easier to work with next time. So, if you discourage it with your metrics, you will see your code base degenerate to unworkability pretty quickly.

Summary

Software engineering is a creative process, much like art or sculpture - programmers create software out of thought-stuff, implemented with code.

Would you measure the performance of Picasso by counting brush strokes?

How do you think Picasso would feel if you did?

Would he be more motivated to create awesome art?

Bad incentives lead to bad outcomes. All these metrics are game-able, and will incentivise unfortunate outcomes for your software. If you try to use them as a carrot & stick to motivate your developers, they will be gamed, to the detriment of your software.

The good news is that most programmers are naturally incentivised to create amazing software, and they are happiest when they are most productive according to research by Dr Tom Zimmerman, a senior researcher at Microsoft Research in Redmond [3].

You need to help them do that by removing the things that are slowing them down. In Picasso's case he needed the right materials, the right environment, and plenty of uninterrupted time to focus. Your developers are not so different.

Exercise 1

What metrics do you use to measure developer productivity, and what have been your results?

Adding More Development Resource to a Late Project

Brook's law says that adding more developers to a late project makes it later [2]. This law was made famous by Fred Brooks in the Mythical Man Month, yet, it still happens in almost every company I've worked with. The project is late, and the immediate response is to try to put more resources on it, as if developers were commodities that could be dropped into a project like hamsters on a wheel to make it go faster... Just add 3 X javascript developers + 2 X backend developers and a couple more testers... Don't do this. The hamsters will get in each other's way, scrambling over each other and pushing each other off the wheel. Don't do it in software projects either.

One reason it doesn't work is that there's a limit to how much you can break down a project to assign to many developers before they start tripping over each other. The thing to remember is that developers generally spend on average 10 X the amount of time reading code as writing it, in order to figure out where to safely make their change without breaking anything else. The bigger the codebase, the more extreme this factor becomes. But imagine what happens when the code you just read changes each time you go back to it? You're trying to keep your place in a large and complex maze in short term memory, and it's changing as you work, like a set of Hogwarts staircases. This is going to both slow you down and increase your error rate.

Each time you add a new person to a project, you also add one more node into the communication matrix. That new person needs to be brought up to speed with what the team is trying to do, and where the project is up to, and who is working on what right now. The time it's going to take the rest of the team bringing the new person up to speed is not inconsiderable. This means the entire team will slow down before it starts to speed up. The later in the project that you do this, the worse the effect will be.

By definition the new developers will not be up to speed on the business domain requirements of the change, and possibly not of the underlying system either, which is often the biggest problem. Therefore the existing team will need to take time away from their work to bring the new team members up to speed. This will take a lot longer than expected. It isn't enough for the new team members to read through the specifications for the project, or even for the piece of the project they are to undertake. 80% of what they need to know will not be in the specification, or the user story. The existing behaviour of the part of the application being changed is the hard part, which the original team have learned by spending months working on it. There's no shortcut for this.

You can think of it in these terms - if you need to dig a hole faster, the limiting factor isn't the number of people, it's the number of spades. And the physical space around the hole.

If your project is late, it's late. You're going to have to deal with that. The best people to help you make it less late, is the team you already have. The good news is that in most software companies there are quite a few things you can do to help them with that.

Exercise 2

Have you ever tried adding development resources to a late project? What was the result?

Adding more progress meetings

Some managers and project managers believe that the reason projects are late is because the developers don't understand the importance of the deadlines, and that putting more pressure on them in the form of more progress meetings, will make them work faster. They add more and more progress meetings with developers to try and understand the problems, in an attempt to micromanage the problem.

The first issue with this approach is obvious. If a developer is sitting in a meeting with you explaining why a task is taking more time than expected, she's not working on her task.

The second issue is that software development is a creative process that requires focus. On the code, not on the deadlines, not on the fact that you're stressed or the fact that your sales people are getting upset because they're going to have to explain to your customers why they can't have their new features when they promised for the price they promised. It is hard to focus on complex problem solving when you're stressed and everyone around you is flapping. If a programmer is constantly thinking I should have finished this by now, oh no I'm behind, I've only got 2 hours left to do this 10 hour task, and we've got another progress meeting in an hour, they're not thinking about the task in hand, and they're more likely to get the equivalent of writer's block, or worse, lose their train of thought and have to begin reasoning through the problem all over again.

The third issue is that a meeting is a context switch, where you will lose all the information you have in your short term memory. If you are in the middle of unravelling a particularly complex piece of code that is misbehaving, then having to stop and go to a progress meeting in the middle of the morning, is going to set you back about half an hour while you refill your short term memory with the details of what you were debugging. Some issues need several uninterrupted hours to work through.

If you must have progress meetings every day, then schedule them for the time the whole team actually starts work so you're not interrupting anyone, or, if everyone is starting at different times, then schedule them for just before lunch. This tends to keep them from overrunning as people are eating into their natural break time. Don't put them at 10am or 3pm or you'll kill 3 to 4 hours productivity with your half hour meeting.

Exercise 3

What regular meetings do you require your development team to attend? What time are these meetings scheduled for? If they are in the middle of the morning, or the middle of the afternoon, try moving them to just before lunch and record the results.

The Number One Software Team Killer

I promised to tell you the number one reason your software is sucking the life out of your development team, so here it is.

Rushed software development.

The primary cause of all your team's problems is rushed deadlines.

Rushed deadlines are the primary cause of Technical Debt in code, which you and your developers will be paying for in every future software change for years, unless you invest the time to re-do the work properly. Essentially you will be paying interest every day until you pay off the debt. This is what most people have forgotten. The rate of progress will gradually slow down as your technical debt levels increase. If you've been rushing software releases out for a few years now, you'll be starting to feel the pain.

Navigating a codebase riddled with technical debt is slow, error prone and painful, and knowing it will never get better because there's always a more urgent problem, is a primary cause of dissatisfaction in software developers, who will slowly lose motivation and become cynical, depressed and unproductive.

Rushed deadlines encourage the skipping of tests because there isn't time to create them, which in turn causes code that you can't easily test to slip into production.

Rushing software development encourages insufficient design thought, or discussion of the specification with the business and the testers, because they need to get cracking with the code, which leads to more misunderstandings of the detail of the requirements.

Rushed deadlines cause bad code that doesn't fully consider its impacts on the rest of the codebase, that doesn't scale or perform well, and that causes more bugs in the rest of the software, either immediately, or for years in the future when other changes are made to that area of the software. Let that hack into production now, and you'll be paying the cost in every subsequent software change for years.

Bad code accumulates, and makes software hard to understand, hard to change safely and hard to write automated tests for.

If you leave *"doing it properly"* until there's a better time, that time will never happen, and your team knows this. Knowing this is a major demotivating factor for developers.

Picasso isn't going to create great art on the back of a paper bag from the supermarket with your kids' crayons because you don't have time to wait for Amazon to deliver the oil paint and canvas. He'll get fed up and go and work for someone else.

Better may be the enemy of done, but rushed software is a recipe for technical debt which is the enemy of the next change, which will be slower than the last change until your software becomes totally unmaintainable, and your company goes under.

And if, like so many others, your team has been working to rushed deadlines for a good few years, your codebase will now be getting very difficult to work with. Your developers will be getting frustrated with working on bad code every day, without ever having the time and

As a software development leader, you need to advocate for your team, as well as for the business. You need to make sure that your project timescales include sufficient contingency to allow for new features to be developed better, and ensure that some resource is invested strategically in repair work to make the existing code base easier to work with.

If you can show your team that you take their issues seriously, and that you are able to advocate for them and negotiate resources to strategically manage the technical debt in your product you will soon start to see productivity and motivation increase within your team.

What Next?

I hope this book has given you a better idea of what works and what doesn't when it comes to improving software development productivity in your team.

If you want to go deeper into the subject of Technical Debt, you can find a link to our video course here on our website:

https://www.chaosontoast.co.uk/courses/become-a-technical-debt-detective/

We have also developed a Technical Debt Strategy Session which is designed to last only 30 minutes focussed on helping you clarify your strategy.

There are a limited number of sessions available per week, due to the time it takes.

This is delivered by our CEO or one of our experienced consultants via a web conference using Zoom.

During this discovery call we will take a dive into your specific software development team situation, and look at how Technical Debt may be impacting your team's productivity.

Specifically we will:

1. Review your business goals for your team and your software, and where you are on the product validation timeline.

2. Take the temperature of your technical debt, and the specific pain points that are affecting your software company.

3. Discuss whether your current strategy for managing technical debt is at the right level for your specific situation, or whether you need to dial up or down your investment in your strategic technical debt management.

At the end of this call, if we think there's a fit, and only if you wish to take the discussion further, the consultant may, with your permission, outline our technical debt management process with you, and how you could incorporate this into your processes for your team.

Our goal is to help you figure out where you're having the most trouble with your software team, and then help you figure out what the best plan of action is.

Please contact us if you would like to book a discovery call.

- helen@chaosontoast.co.uk
- Linkedin Page
- Facebook Page
- @chaosontoast

Bibliography

1. https://www.drucker.institute/thedx/measurement-myopia/

2. The mythical man month by Fred Brooks.

3. https://www.microsoft.com/en-us/research/podcast /the-productive-software-engineer-with-dr-tom-zimmermann/

The Number 1 Reason Your Software Is Sucking The Life Out Of Your Development Team