Agile Product Ownership

A software development guide for non-technical leaders.

BYRON SOMMARDAHL

CO-FOUNDER OF ACKLEN AVENUE

Agile Product Ownership

A software development manual for non-technical leaders.

Byron Sommardahl

This book is for sale at http://leanpub.com/agileproductownership

This version was published on 2016-09-23

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To my lovely wife, Pamela. Thank you for giving me the time and space to work on this book. Te amo, mi princesa.

Contents

Introduction

Software is Hard

Software Project Failures

The national and international failure rates for software projects are difficult to justify. Not because they aren't true, but because it's hard to explain how such horrible failure rates can be allowed to exist year after year. One study shows that as much as 80% of software projects either failed to be delivered or came in almost 200% of the original estimate of cost. With such staggering failure rates, it really puts the pressure on product owners to do whatever they can to ensure that their project doesn't become part of next years statistics. The good thing about these statistics is that there's no mystery as to why failure rates are so high. Well, there might be a mystery as to why the software industry allows things like that to happen. But we do know what causes these failure rates. And we know how to prevent them.

Software project failure is completely preventable. Experiencing a software project failure in this decade is similar to dying from a Papercut.

There are so many reasons why software project could fail, which is likely the reason why so many do.

How do we know or define project failure?

Project failure can be defined as one or a combination of cost overruns, late deliveries, poor quality, and/or developing a product that does not get used.

It's important to understand what goes into those statistics. The obvious ways a project can fail is to never reach the users for whom it was intended. Other forms of failure could be extreme cost overruns, late deliveries that damage the software's value, or poor quality that results in the inability to maintain, extend, or fix later on.

Failure to Benefit

A solution without a problem is not much of a solution at all. It provides no benefit or value. Software that can't realize it's value is useless and a complete waste and should be considered a failure.

Software realizes it's value by making someone's life better, increasing revenue, automating some process, or providing some benefit. Even the canonical "Fart App" on your mobile device solves a problem, albeit trivial (apparently we needed a laugh).

Too many software products become failures because they are unable to solve even the most trivial problems. There are a few ways to miss the mark, here. For example, perhaps the software product reaches the users as designed, but it doesn't solve the right problem or the software doesn't solve the problem in the right way. Either way, the users reject the product and it becomes a failure. Another example is a product that tries to solve a problem that is a moving target. There's a great danger in

the software attempting to solve a problem that is no longer solvable or no longer need solving. Yet another example comes from a development team who fails to properly implement the solution to a perfectly good problem. Software that doesn't do what it should is a failure.

Failure to Deliver on Time

Failure to Stay on Budget

Software is hard to estimate correctly and almost always misses the estimate mark. But sometimes software development projects are so grossly under estimated that the project spends all of the companies reserves just to finish the project and recuperate what's left of the investment. Projects like this deserve to be called failures because they were so poorly planned an estimated that they spend most of The value that the software might have provided. Projects like this cause the business to switch to recovery mode instead of being able to enjoy the values of the software brings.

Failure to Build a Maintainable Product

Software can be delivered in pristine condition and satisfy all of the requirements and live up to its design. They can solve the problems of the world and make everyone's life's better. But that's just talking about delivery day. What happens to software that, under the hood has low levels of quality, when it's time to fix a bug?

1) long lived technical debt 2) poor project management practices 3) lack of understanding of the final product on the developers part and on the product owners part 4) Why do software projects go over budget?

Failure

The Death of My Company

It happened on an cold, rainy afternoon in late autumn of 2010. I will never forget that day. The decision I made that day left a deep mark on my soul that will surely never fade away completely. It was the right decision, but a bit too late for "right decisions." Damage done.

In the late 2000's, just before startups became "cool", a friend and I had an idea for a neighborhood-based social network. We were sure it would be a huge hit and make a lasting social impact, especially in America's neighborhoods. We decided to invest in the idea and devote ourselves to development. Ben, my partner, handled the business side of everything. I handled the tech and development. Ben and I were a great team.

Go Speed Racer, Go!

Over the course of 12 months, we developed an amazing amount of functionality. Ben and I were generating ideas left and right. A team of developers and I set out to turn ideas into code. We spent a lot of time on news aggregation systems that would go out to different sources and collect local news. We created a system that allowed neighbors to band together and petition government officials. We created a crime mapping solution to alert neighbors when bad people were doing bad things. We spent time on messaging, chat, groups, profiles, coupons, advertisements, and an impressive list of features, all meant to make people fall in love with our product. Demos were impressive, especially when an idea came up during a demo and a developer had it implemented before the demo was over! We were making progress like a speeding freight train. Amazing times.

We started to get some attention from outside investors and large tech companies like Microsoft, AOL, and Yahoo. That brush with "fate" stoked our fire and made us walk a little taller. We added some more features to the product and did some more demos. We also added some "shareholders" to the company to help us get to the next level. Feeling great.

Quicksand Rising

And then we started slowing down. New features took longer to implement. New developers took longer to spin up. The bug list was growing faster than we could patch. Code that went in super fast took 10 times longer to modify, breaking countless disparate parts of the system along the way. It was like trying to run in quicksand. Our once nimble team had been reduced to a pack of sad sloths. After 6 months of death march, we made the decision to start over with a new code base and new technology.

And just like that, we were fast again! That's how it felt to the developers, at least. We were given the chance to "get it right this time." But 6 months into our new shiny code base, we started to feel the quicksand rising again. What was it we were doing here again? What are we trying to build? Who is this for, anyways? We had lost our way.

The idea that once woke me up early in the morning and fueled my 18-hour coding binges got lost somewhere along the way. My passion for my company and what we were building had been replaced with a feeling of helplessness, anxiety, and despair. So, on a cold, rainy afternoon in late autumn of 2010, I finally threw up my hands and stopped coding. Development stopped, meetings stopped, momentum bottomed out, and our company drowned mid-river.

In 2010, according to the Standish Group's 2011 Chaos Report, "37% of all projects [succeeded] (delivered on time, on budget, with required features and functions)." I didn't realize it at the time, but I and my company had just become a permanent part of the 63% that fail.

Lessons Learned

- Release Something—Did I ever say anything about our users? Nope, because we had none. We just kept on cranking out feature after feature without regard for the people who could be using our product. Don't get me wrong. We were definitely focused on our users, working hard to develop and perfect our functionality so that the product would absolutely delight the neighborhood-dwellers of the United States. We kept telling ourselves that the 30 some odd features we had weren't enough. Therefore, we must keep developing. In our quest for the proverbial kitchen sink, we never released. If we had known what we know now, we would have developed the most compelling/valuable features first and then released them for use. Then, we would have listened to the feedback from our users to make sure we were building the right pieces and solving the right problems. Even if it was incomplete, it would have made the difference between life and death for our company if we had just released something.
- Fast Code is not Good Code—At the beginning of each phase of development, we were hauling tail! Coding and producing at a rapid pace is a rush that's hard to describe for non-programmers. But even non-technical folks enjoy seeing features flying out the door at breakneck speeds. The truth is, though, that fast code is not necessarily good code. Our fast code, over time, ended up paralyzing our development team. That's not fast at all! Knowing what I know now, we would have used proven engineering techniques and practices to ensure high quality across the board and over time. We would have worked in tight collaboration with one another and with the business. We would have sacrificed our hubris and accepted strong code instead of speed. We would have listened to experts like Robert Martin and Kent Beck when they tell us that a clean, maintainable code base makes us faster in the end and protects the project from failure. Fast code is great for prototypes and throw away ad campaigns on Facebook, but it's can be dangerous for anything else. Fast code is not always good code and can lead to the demise of your project (or your company).
- Developers Should Not Be SME's—For this next one, I take full responsibility. I never intentionally made myself into a subject matter expert for our project. It just happened. Yes,

programmers are experts. But when a programmer is an expert over a section of code, a piece of functionality or a system, he becomes so important that he is irreplaceable. My knowledge of the entire system made me too important to lose. That feeling of importance is hard to resist. I can say that I never maliciously accepted so much importance, but I can also say I didn't turn it away. There were several times when I felt like I wanted to get away from the project, especially as things started to slow down and I felt frustrated. But I realized that, if I left, the project would come to an end. And when I finally left, it did. If I could go back in time, I would've shared my knowledge with other developers. I would have encouraged techniques like pair programming to prevent myself from becoming a silo of knowledge. Still, I might have left the project, but at least my departure wouldn't have meant the downfall of the company.

This Time We'll Get It Right!

My favorite line from "Batman Begins" is when Thomas Wayne asked his son, "Why do we fall, Bruce?" Bruce answered, "So we can learn to pick ourselves up." I like success just like you. But life comes with this guarantee: WE WILL FALL. We will not always be successful, and we may even hurt ourselves or others on the way to failure. But you and I will not stay down. We will get up, brush ourselves off, and forge ahead with the determination to get it right the next time. I hope the lessons I learned from my own failures can serve to help you lead your own software development projects to long-term success.

Technical debt

When you go to the cash register to pay for your goods, you have to offer something of value before they will let you leave the store. If you pay with cash, then the deal is sealed. Now the store has your $100, and you have their goods. I prefer not to carry much cash, so I will almost always pay with my credit card. After all, I might need that cash at my next stop and my credit card gives me points that I can use on a flight next month. I get certain benefits from using my credit card. For example, if my paycheck hasn't been deposited yet, I can still buy things. Also, I don't have to physically go to the bank to take out some cash. And there's safety… I live in a dangerous part of the world, so it's a bad idea to carry around a lot of moolah. So, when I pay with plastic, I get to reap the benefits of a small loan.

If I pay off my credit cards by the end of the month, the benefits I get from those little loans end up costing me nothing. What a deal! But, if I become undisciplined and I let the month go by without paying, then my small loans turn into "debt". Credit cards have notoriously high interest rates and costly penalties for debt. Get on bad terms with your credit cards, and you won't even be able to to remember the benefits you might have had.

When building software, development teams are constantly taking loans and, potentially, incurring debt. Of course, I'm not talking about financial debt, but about a concept known as "technical debt". The term "technical debt" was first coined in 1992 by Ward Cunningham as he was trying to describe the challenges in maintaining a large, long-lived software project. His metaphor of technical debt effectively explains how short-term loans can empower developers to deliver features faster, and how long-term debt can suffocate a development team.

You can think of technical debt much like any other type of debt. Many businesses depend on debt to grow or survive. Over time, small loans turn into much larger debt. If loans are paid back quickly, they don't accumulate interest. Sometimes payback is delayed for one reason of another. The longer a debt is allowed to exist, the more interest it accumulates and the more it costs. Left unchecked, debts and their cost can grow out of control and may lead to the debtor's discomfort or even downfall.

Short-term Technical Loans

A technical loan is any beneficial software development activity that is postponed. A natural occurrence in any software development project, small short-term technical loans can be taken in the form of shortcuts, temporarily messy code, or some experimentation. Developers need to think on their toes and take advantage of their creativity while coding. When developers allow themselves to explore and experiment, they can often arrive at a working solution faster than if they had been held to a high engineering standard throughout the process. Even well-known software engineering techniques like test-driven development encourage short-lived technical loans as part of the "red-green-refactor" cycle (see TDD). The red and green stages of that cycle are for experimentation and

creating something that “works”, taking a small technical loan in the process. The refactor stage is for paying off the technical loan by cleaning up the code. Technical loans offer development teams great benefits, much like my credit cards. Take small, manageable loans, reap the benefits, and pay them off before they are able to incur interest and penalties.

Technical Debt

When technical loans go unpaid, they convert into debt and incur interest and penalties. Technical debt is created in the same way a development team creates technical loans: shortcuts, experimentation, and “temporarily” messy code. However, when that code is left behind and the team allows that code to stay in the code base, it becomes a first-class citizen of the software. The cost started out as a useful tool to help developers innovate. But, once it gets left behind, it converts into something that actually stands in the way of future innovation. Future features are inevitably built on top of that technical debt which further deepens its grasp on the software. Technical debt accrues interest and penalties, which get paid through velocity reduction, missed deadlines, inability to deliver, endless bug lists, overhauls, re-writes and other consequences.

Technical debt includes but is not limited to…

- complex code
- untested code
- brittle tests
- re-invented wheels
- out-of-date code comments
- out-of-date documentation
- hardcoded values
- nested structures
- large conditional structures
- temporary hacks
- unused code
- poorly-named code structures
- unused database tables and columns
- or anything in design or code that must or should be done in a different way or could stand in the way of future modification or innovation.

Technical Debt as a Feature

Have you ever used a credit card for a large purchase or expense knowing that you would not pay it off by the end of the month? I wouldn’t say it is wise, but it might be the only option you have, especially if you have to pay for an emergency surgery or repair to your house. The benefit is that you can cover the emergency expense. The consequence is that you will end up paying much more

down the road. For some situations, it makes sense to intentionally incur technical debt for the good of the company or the product.

There are plenty of examples of software projects that have tight deadlines and need to meet certain goals before a trade show or a important stakeholder demo. Still other software projects are focused on beating the competition to market. In these examples, the feature set is not really negotiable. Scope is set based on market research and usability studies. That date I mentioned is immovable and everything absolutely must come off without a hitch for that day. Without speeding up efforts and incurring technical debt, the product or company might not survive.

The canonical example is Twitter. The software that ran Twitter was thrown together as fast as possible in order to get to market at the right time. When Twitter made it to users desktops around the world it was an instant success. Since it's first release, Twitter has become a marker in our culture, not just in the United States where Twitter started, but also around the world. But none of the world's new microbloggers realized that Twitter was sitting on top of a back-end hastily built out of sticks and chewing gum. Twitter worked, and that's all that mattered. The benefits of technical debt allowed the makers of Twitter to get something valuable to market at break-neck speeds. Technical debt is probably why Twitter is now a household name. Without the intentional use of technical debt, Twitter might not exist today.

Unwelcome Technical Debt

Technical debt is only acceptable when the product depends on it to succeed or survive. Benefits like speed and developer innovation are huge. But constant, unbridled speed is not always the best way to reach your goal. Let's be honest. Most deadlines are imaginary or not nearly as important as we make them out to be. Are we really racing the clock, or do we just want to apply some healthy pressure? If your software is so important to your business, and you have time to build it with care, then "care" that's the standard you should accept from your development team. Technical loans are expected and should be tolerated so long as they are paid off quickly. When your software doesn't depend on speed to market to succeed or survive, then technical debt should not be a part of your team's expectations or standards.

Consequences of Technical Debt

As mentioned, the benefits of short-lived technical loans are increased innovation and speed. But, once the loans begin to accrue interest, they convert to technical debt. The consequences of technical debt are decreased speed, innovation, stakeholder confidence, and team engagment.

Speed

Technical debt usually results in slower development speeds. Many times we compare technical debt to "mud". As you can imagine, walking in mud is much slower than walking on a firm surface. Your feet get stuck, sometimes you lose a shoe, you're concerned about getting your pants dirty, and you

end up having to clean off later on. On the other hand, on a solid surface, you're able to walk, run, skip, flip or jump at top speeds. You can go as fast as you want. Development speed at the beginning of a project is very much like running on a solid surface. It feels great. Everything seems smooth and easy. In contrast, working in a project with heavy technical debt is like trying to run in a muddy swamp. Everything takes way too long. What used to take an hour now takes days or weeks.

Innovation

Writing software is a creative process. Consider an artist painting a tapestry. The painter has paint brushes, various colors and a clean canvas with which she can realize the image in her mind. Artists innovate as they create. They depend on quick feedback loops and responsiveness of their tools and their medium in order to maintain the flow of innovation. Any break down in the process threatens the completion of the tapestry. Getting slowed down by mistakes (technical debt) that have accumulated over time is something that can rob a project of its innovative potential. Software that can no longer flex, bend and respond to innovation quickly is software that is marked for death.

Stakeholder Confidence

Software projects exist because stakeholders want them to exist. As long as they have confidence in the team and development process, the development team has breathing room and freedom which is beneficial to productivity. When technical debt slows down improvements, fixes, and innovation, stakeholders inevitably (and rightfully) lose confidence in the development team. Micromanagement ensues and freedom is replaced with slavery.

Team Engagement

An engaged development team one understands and believes in the vision of the product they are building. Engagement drives better questions which lead to greater ownership and superior decisions. An engaged team is more productive and produces a higher-quality product. As technical debt builds, the software project becomes less and less enjoyable to work with. Developers find friction at every corner. Team leads become frustrated with missed deadlines. Pressure increases and, as a result, the team becomes less engaged. Eventually, teams that deal with technical debt on a daily basis experience high turn-over, which affects engagement even more because of the need for constant training and internal product evangelism.

Recognizing Technical Debt

Unfortunately, technical debt is hard to spot. In fact, technical debt it mostly invisible to users. In many cases, for non-developers, technical debt is like the wind. You can see the effects of the wind, but you can't see the actual wind. And worse, many times we can't see even the effects of technical debt until the project is already paying the consequences of it.

You might have technical debt if…

- Your once-performant development team is now releasing features and fixes at a snail's pace.
- Your bug backlog has been your priority for weeks.
- Your developers are pushing for overhauls and re-writes.
- You are hiring new team members to replace the ones who you can't seem to keep.

Though it is nearly impossible to detect technical debt from a user's or stakeholder's perspective, we can certainly lift the hood and see what's going on from a more technical vantage point. There's a lot that the code can say about itself as to the level of technical debt it carries. By running your codebase through tools like static analyzers, compilers, linters, and more, you can get a window into a wealth of information about the health of the stuff under the hood. Of course, the merits of each calculated metric can be argued, so it's best to view them with a grain of salt. Here are a few potential metrics that could help reveal some warning signs:

- Lack of Unit Test coverage
- High cyclomatic complexity
- High Halstead complexity
- High number of lines (compared to similar projects)
- High coupling
- Low cohesion
- High occurrance of duplicate code
- Compiler/linter warnings
- Style & consistency rule warnings

All of these metrics are explained in more detail in later chapters.

Who Decides?

As destructive and costly as technical debt can be, wouldn't you think that the decision to incur it falls on the business or product owner? It's true, at times the costs of technical debt can be justified because it allows the team to get to market faster. This results in a business decision, premeditated or inferred, to lay quality aside and push the team to race to the finish line. However, such demands for extreme speed are not the norm. And, if you ask most product owners if they would rather high or low quality, they would choose high quality. So, it seems like it should be a simple business decision, doesn't it? Sadly, most technical debt it incurred by accident. It starts as a small loan in the form of a shortcut, but is left behind and forgotten. Call it lack of skill, lack of experience, or lack of engagement, but technical debt is almost always unintentional. But don't go blaming the programmers. The decision to tolerate technical debt was made before the project began by a lack of communication between the product owner and the team about expectations of quality.

Technical Debt is Avoidable

It's probably impossible to avoid all technical debt, especially considering larger systems. Software projects that dissolve into muddy swamps of technical debt are far too common, but should not be lifted up as the standard. For systems that include quality as a feature, there's no reason why technical debt should go unchecked and be allowed to grow out-of-control. Constant awareness of technical debt and a team-wide commitment to fighting it will yield a system that is flexible and agile for years to come.

Product Ownership

Agile Development Concepts

Standards

ISO/IEC 12207 is the standard on software lifecycle processes. this standard is well-thought-out and heavily scrutinized and potentially provided the sound roadmap to a quality product. It is compatible with both waterfall and frameworks, but standards like this are seldom followed or required.

At the time of this writing software development is regulated in only a few domains such as medical devices, pharmaceutical, financial, nuclear systems, automotive, and avionic sectors. outside of those domains, software development is comparable to the wild west.

In almost every industry where billions of dollars are spent, there is regulation and standards. Standards are documented and armada law. Those that step outside of those standards are not considered professionals. They can even be held responsible and punished for their Miss deeds. For example in the medical profession, an entire industry has been built on the concept of medical malpractice. Doctors have an oath that says that they will do no harm. When they do harm, there are consequences for that doctor. Accountants must use double entry accounting. Accounting errors introduced by accountants have consequences for that accountant. Construction workers must adhere to government standards and codes before the House or building can pass inspections and be inhabited. What I'm trying to say is that all these industries have standards and rules and people, more or less, follow these rules because they know that they are the right thing to do.

The software industry has never had any such standards. Maybe it's still too new, but software developers are allowed to build systems that may or may not be sound. Software can literally kill people in the right context. More software can cost businesses billions of dollars. Other software can completely ruin and destroy businesses. Isn't it strange that the software industry is not held to any kind of standards or regulation?

No one likes regulation and rules especially when people can experience so much creative freedom. I started programming when I was eight years old. I made my first money making software at age 14. I could have never done that, if the creation of software was governed by rules and regulations. Kids around the world are able to throw together applications and get their app into the mobile App Store. It's incredible and a lot of fun.

Since there are no universally excepted standards and since there is no government regulation, the software industry is highly fragmented. There are those who value speed so much that quality is unknown and something that we don't really care about. On the other side there are those that care so much about quality that a piece of software may never reach the market because of drastic delays caused by over engineering. And in the middle, there are thousands of software development professionals who are doing the very best they can to try to meet both demand. But since there are

no universally excepted standards, there's nothing that ties developers together on a common goal of reasonable speed and reasonable quality.

There are a few strong voices crying out in the wilderness, and what they say could be used as standards easily. But they are preaching to the choir or really. Software developers understand what it takes to get the job done right. But even with such influential voices, their words, methods, principles, and practices are rarely heated as a viable standards.

Agile Culture

Working with Teams

Agile Product Design

Agile Product Delivery

Develop Products Together