startuplessonslearned.com

Lessons Learned: Five Whys

10-13 minutes

Taiichi Ohno was one of the inventors of the Toyota Production System. His book Toyota Production System: Beyond Large-Scale Production is a fascinating read, even though it's decidedly non-practical. After reading it, you might not even realize that there are cars involved in Toyota's business. Yet there is one specific technique that I learned most clearly from this book: asking why five times. When something goes wrong, we tend to see it as a crisis and seek to blame. A better way is to see it as a learning opportunity. Not in the existential sense of general self-improvement. Instead, we can use the technique of asking why five times to get to the root cause of the problem.

Here's how it works. Let's say you notice that your website is down. Obviously, your first priority is to get it back up. But as soon as the crisis is past, you have the discipline to have a post-mortem in which you start asking why:

1. why was the website down? The CPU utilization on all our front-end servers went to 100%
2. why did the CPU usage spike? A new bit of code contained

an infinite loop!

3. why did that code get written? So-and-so made a mistake
4. why did his mistake get checked in? He didn't write a unit test for the feature
5. why didn't he write a unit test? He's a new employee, and he was not properly trained in TDD

So far, this isn't much different from the kind of analysis any competent operations team would conduct for a site outage. The next step is this: you have to commit to make a proportional investment in corrective action at every level of the analysis. So, in the example above, we'd have to take five corrective actions:

1. bring the site back up
2. remove the bad code
3. help so-and-so understand why his code doesn't work as written
4. train so-and-so in the principles of TDD
5. change the new engineer orientation to include TDD

I have come to believe that this technique should be used for all kinds of defects, not just site outages. Each time, we use the defect as an opportunity to find out what's wrong with our process, and make a small adjustment. By continuously adjusting, we eventually build up a robust series of defenses that prevent problems from happening. This approach is a the heart of breaking down the "time/quality/cost pick two" paradox, because these small

investments cause the team to go faster over time.
I'd like to point out something else about the example above. What started as a technical problem actually turned out to be a human and process problem. This is completely typical. Our bias as technologists is to over-focus on the product part of the problem, and five whys tends to counteract that tendency. It's why, at my previous job, we were able to get a new engineer completely productive on their first day. We had a great on-boarding process, complete with a mentoring program and a syllabus of key ideas to be covered. Most engineers would ship code to production on their first day. We didn't start with a great program like that, nor did we spend a lot of time all at once investing in it. Instead, five whys kept leading to problems caused by an improperly trained new employee, and we'd make a small adjustment. Before we knew it, we stopped having those kinds of problems altogether.

It's important to remember the proportional investment part of the rule above. It's easy to decide that when something goes wrong, a complete ground-up rewrite is needed. It's part of our tendency to over-focus on the technical and to over-react to problems. Five whys helps us keep our cool. If you have a severe problem, like a site outage, that costs your company tons of money or causes lots of person-hours of debugging, go ahead and allocate about that same number of person-hours or dollars to the solution. But always have a maximum, and always have a minimum. For small problems, just move the ball forward a little bit. Don't over-invest. If the problem recurs, that will give you a little more budget to move the ball forward some more.

How do you get started with five whys? I recommend that you start with a specific team and a specific class of problems. For my first time, it was scalability problems and our operations team. But there is no right answer - I've run this process for many different teams. Start by having a single person be the five whys master. This person will run the post mortem whenever anyone on the team identifies a problem. Don't let them do it by themselves; it's important to get everyone who was involved with the problem (including those who diagnosed or debugged it) into a room together. Have the five why master lead the discussion, but they should have the power to assign responsibility for the solution to anyone in the room.

Once that responsibility has been assigned, have that new person email the whole company with the results of the analysis. This last step is difficult, but I think it's very helpful. Five whys should read like plain English. If they don't, you're probably obfuscating the real problem. The advantage of sharing this information widely is that it gives everyone insight into the kinds of problems the team is facing, but also insight into how those problems are being tackled. And if the analysis is airtight, it makes it pretty easy for everyone to understand why the team is taking some time out to invest in problem prevention instead of new features. If, on the other hand, it ignites a firestorm - that's good news too. Now you know you have a problem: either the analysis is not airtight, and you need to do it over again, or your company doesn't understand why what you're doing is important. Figure out which of these situations you're in, and fix it.

Over time, here's my experience with what happens. People get used to the rhythm of five whys, and it becomes completely normal to make incremental investments. Most of the time, you invest in things that otherwise would have taken tons of meetings to decide to do. And you'll start to see people from all over the company chime in with interesting suggestions for how you could make things better. Now, everyone is learning together - about your product, process, and team. Each five whys email is a teaching document.

Let me show you what this looked like after a few years of practicing five whys in the operations and engineering teams at IMVU. We had made so many improvements to our tools and processes for deployment, that it was pretty hard to take the site down. We had five strong levels of defense:

1. Each engineer had his/her own sandbox which mimicked production as close as possible (whenever it diverged, we'd inevitably find out in a five whys shortly thereafter).
2. We had a comprehensive set of unit, acceptance, functional, and performance tests, and practiced TDD across the whole team. Our engineers built a series of test tags, so you could quickly run a subset of tests in your sandbox that you thought were relevant to your current

project or feature.

3. 100% of those tests ran, via a continuous integration cluster, after every checkin. When a test failed, it would prevent that revision from being deployed.
4. When someone wanted to do a deployment, we had a completely automated system that we called the cluster immune system. This would deploy the change incrementally, one machine at a time. That process would continually monitor the health of those machines, as well as the cluster as a whole, to see if the change was causing problems. If it didn't like what was going on, it would reject the change, do a fast revert, and lock deployments until someone investigated what went wrong.
5. We had a comprehensive set of nagios alerts, that would trigger a pager in operations if anything went wrong. Because five whys kept turning up a few key metrics that were hard to set static thresholds for, we even had a dynamic prediction algorithm that would make forecasts based on past data, and fire alerts if the metric ever went out of its normal bounds. (You can even read a cool paper one of our engineers wrote on this approach).

So if you had been able to sneak into the desk of any of our engineers, log into their machine, and secretly check in an infinite loop on some highly-trafficked page, here's what would have happened. Somewhere between 10 and 20 minutes later, they would have received an email with a message more-or-less like this: "Dear so-and-so, thank you so much for attempting to check in revision 1234.

Unfortunately, that is a terrible idea, and your change has been reverted. We've also alerted the whole team to what's happened, and look forward to you figuring out what went wrong. Best of luck, Your Software." (OK, that's not exactly what it said. But you get the idea)
Having this series of defenses was helpful for doing five whys. If a bad change got to production, we'd have a built-in set of questions to ask: why didn't the automated tests catch it? why didn't the cluster immune system reject it? why didn't operations get paged? and so forth. And each and every time, we'd make a few more improvements to each layer of defense. Eventually, this let us do deployments to production dozens of times every day, without significant downtime or bug regressions.

One last comment. When I tell this story to entrepreneurs and big-company types alike, I sometimes get this response: "well, sure, if you start out with all those great tools, processes and TDD from the beginning, that's easy! But my team is saddled with zillions of lines of legacy code and ... and ..." So let me say for the record: we didn't start with any of this at IMVU. We didn't even practice TDD across our whole team. We'd never heard of five whys, and we had plenty of "agile skeptics" on the team. By the time we started doing continuous integration, we had tens of thousands of lines of code, all not under test coverage. But the great thing about five whys is that it has a pareto principle built right in. Because the most common problems keep recurring, your prevention efforts are automatically focused on the 20% of your product that needs the most help. That's also the same 20% that causes you to waste the most time. So five whys pays for itself awfully fast, and it makes life noticeably better almost right away. All you have to do is get started.

So thank you, Taiichi Ohno. I think you would have liked seeing all the waste we've been able to drive out of our systems and processes, all in an industry that didn't exist when you started your journey at Toyota. And I especially thank you for proving that this technique can work in one of the most difficult and slow-moving industries on earth: automobiles. You've made it hard for any of us to use the most pathetic excuse of all: surely, that can't work in my business, right? If it can work for cars, it can work for you.

What are you waiting for?