SUBJECT7
QUALITY IS A CULTURE

The Value of Frequent Releases

By Matthew Heuser for Subject7

The Agile Manifesto has twelve principles, that include "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."

For its time, the Manifesto was revolutionary. Today, that speed seems quaint. Agile Machismo has taken over, with one week sprints better than two, and continuous deployment, where every change request goes to production as soon as it passes a battery of tests, is best of all. But are shorter deployments actually better, and if so, why? Wouldn't you just spend a lot more time regression testing?

That's a pretty complex and nuanced question. Today I'd like to try and answer that question. To that, I'll break it down into three issues: The obvious reality of value to the customer (without the fancy calculus), how that fundamentally changes the structure of software work, and how that *could* impact software testing.

Here goes.

Value to Customer

This one is pretty straightforward. If you ship every two weeks, then in two weeks your customers can start using the new features. If the company across the street delivers code to production every six months, then it takes six months. Say in the faster release universe that you deliver ten story points of value in the sprint, and those story points have some real value. Assuming one day of customer value per point, then in five and a half months, when the other team finally ships, you've delivered 1,210 day-points of value. (22 business days in a month, 10 points, 5.5 months.) You can do similar math on sprint two, on sprint three, and so on. Best of all, you can prioritize the delivery of value, so you deliver the most important things first.

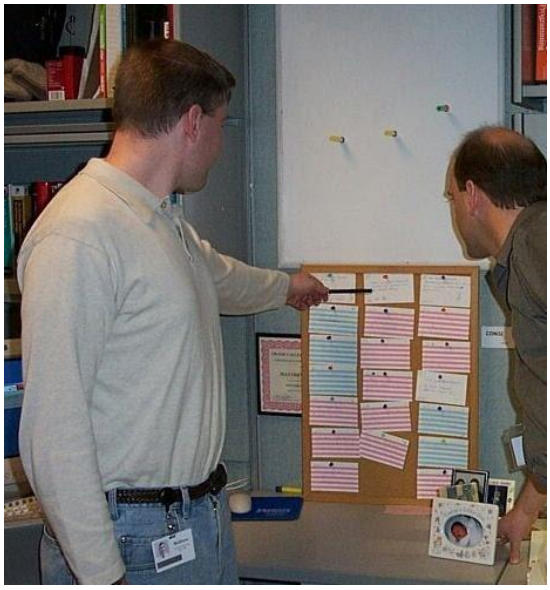
There are a few organizations where this doesn't make a ton of sense. Car and television makers will say they ship firmware, and once a year is their cadence. ERP and first generation online product companies will say their product is either done or it is not; incremental delivery doesn't make sense in their world. Yet Tesla pioneered online updates for cars. I worked on an ERP conversion that cut over January first, and we didn't need the monthly reports until February 1st; we didn't need the quarterly reports to be converted until April. Basecamp, a software project management tool with one hundred thousand paying customers, initially released without the ability to bill customers. The product had a thirty-day free trial, so the owners figured they could release on, say, October 1st, then put the ability to bill customers into the next sprint. That way, they could release the product two weeks earlier, and collect revenue earlier. With $25 million a year in sales, that means each year the company is about a month

ahead of where it would be otherwise. In accounting terms, a month earlier means a month more revenue.

Would you like your product to generate an extra month of revenue by the end of the year? That example is one of the "hard" ones, the "initial release" that "can't be split."

Changes to the Way Software is Developed

Many of my formative years in software were spent at an insurance company. That time straddled the line between pre-and-current Agile. We started with the waterfall method, with a large number of projects going on at the same time that were mostly delayed by this or that. Around 2004, I worked on the first project at the company to use story cards as index cards; that is when this picture was taken.



The portfolio management process at the time was … interesting. Internal customers would advocate for their projects, and, perhaps every year or two, their department would "get one." Because projects were so rare, the customers often took what I would call a "kitchen sink" approach to requirements. Basically, they wanted to jam as much as possible in the project, because it would not come back again anytime soon.

I took the big, thick requirements document filled out using the template and broke it down into a dozen cards. I brought the cards to the customer, asked them to sort the cards, and told them they would have their first new functionality in two weeks, not six months. Once they used that functionality, they could order the cards or create new cards. The only catch was that in twelve weeks, the music stops and so does development. This was totally new for them; it turned planning into a game. The only thing the manager did not like was the twelve-week limit.

Many people are familiar with the idea of the 80/20 rule. It sounds great, in theory, until your customer tells you they really do need all the things - these are *requirements*, Matt, not *desires*.

Yet we finished that project in five two-week sprints, for a total of ten weeks. We finished early.

Think about what it would mean to deliver that 80% of value, then switch to something else. Do a bunch of stories that deliver $2,000 a month and, when you get to the ones that would only delivery $1,000, cut over to another project delivering $2,000. Rinse and repeat. The world, as they say, becomes your oyster.

The Impact on Testing

This is the hard part. It's easy enough to scale down development - just change one field, one button. Serious testers intuit that for most programmers, a whole lot needs to be retested. Not just that button. Scaling up testing frequency is a lot harder than scaling up development. Done poorly, you'd certainly get less software out the door in a given month by trying to regression test twenty-two times, at once per day, than once per month. And with multiple deploys per day … fuhgetaboutit.

The longer, harder road is to improve engineering discipline and skill to reduce regressions. That includes architecture work, fast deploys with feature flags to enable rollback, plus the development of programming skill.

Or you could just get really good at testing.

As it turns out, testing is a bit like building a muscle. If it hurts, and you do more of it, and do that well, you get stronger. It is always possible to lift heavy incorrectly, and do serious damage. Form matters. The same is true in testing. Most of the organizations I work with can usually cut their testing time in half with a few simple tweaks. In some cases, it is a 90% reduction in test time. Shrinking the cycle time and rejiggering the process could mean you throw away old, outdated ways of doing work.

I am not suggesting you go to continuous deployment overnight. Instead, get better at testing, and engineering, intentionally. Call each step forward a win.

Conclusions

Ship more often, deliver value. Ship more often, deliver less waste. Ship more often, become better at programming and testing.

Ship more often should be feared, but in the same sense you fear going to the gym or changing your diet. If you do the work, in six months, or a year or two, you'll be, well, shredded. You'll look back at what you were and see something you hardly recognize. Of course, if you want to go to the gym or not that's on you. Right now, I've had other priorities in my life. The software gym? It costs you almost nothing, and offers you so much.

Ship. More. Often.