Stanford eCorner

Agile Vs. Waterfall Product Engineering

Eric Ries, *Author*

September 30, 2009

Video URL: http://ecorner.stanford.edu/videos/2292/Agile-Vs-Waterfall-Product-Engineering

Author and entrepreneur Eric Ries unpacks the difference between waterfall and agile product development theories, and outlines when each are best employed. Waterfall - the linear path of product build-out - is best used when the problem and its solutions are well-understood. However, its hazard is that it can also lead to tremendous investment without guarantee of its success. Agile development, on the other hand, is a less-risky model of what can happen when the product changes with frequent user feedback and minimal waste. Without an authoritative solution clearly in sight, which is often the case of the startup, agile programming allows the growing enterprise to build-out quickly and correct itself often.

Transcript

So let's talk about Agile product development and what I want to do is try to convince you that the experience I had in Startup 1 and Startup 2 was not just my idiosyncratic experience but, in fact, represents a kind of a larger trend. And so I'll do that by showing what I went through schematically. And a point of disclaimer is that they are not everybody is interested in software startups per se. Let me just say that that is my experience but I think these principles have broader applicabilities, I said at the beginning. But I also want to say, and I don't mean to be melodramatic but I do believe that the future survival of our civilization depends on our ability to reliably ship new kinds of software. Why? Because every new product in the industrialized world is either contains software in it now or is built with software assist. So if we accept the same kinds of failure rates that we've gotten used to in a software industry, globally, then I think we're in a real trouble. I don't think we have to wait for robots to take over our civilization. We're doomed anyway. That's a digression.

It's traditional in a talk about development methodologies to beat up on the waterfall methodology. This is the kind of a traditional way people are thought to build products. This is how I was trained as an engineer, so I will. Waterfall is this idea that you take a product idea. You turn it into some kind of requirements document, then you collect specifications. You build the design for it; you implement it. You hand it off to some kind of QA function and it enters into maintenance mood. So a fundamentally linear batching queue way of building products. And although that's fun to beat up on because, of course, this is what makes achieving failure possible, because you're getting this positive feedback about how you're advancing the plan, even when you're advancing the plan off a cliff. It's actually important to understand that waterfall is appropriate methodology to use in a certain context.

The context when both the problem and the solution you are trying to solve are relatively well understood. When you can model what's going to happen in the future, you want to do this. The issue is that in almost all high-tech product development, this is not true. So if you look at the academic research on, for example, on an IT projects that are built using waterfall, something like 6 out of 10 of them fail outright. Think about that. The lucky 4 out of 10 are the ones that come in way late and

way over budget. Six out of ten never finish. They just entered the batch size death spiral and you never hear from them again, so luckily as an industry, we've been working on doing something better. It's called agile product development. And the insight of agile is that we can eliminate all kinds of waste from our development process in waterfall.

For example, when you build the specification document that goes stale or you have a meeting where you accomplish nothing, or you build extra APIs that you might need in the future but then actually while not needing them. All that's waste. And so what we want to do is build the product itself iteratively so that we change our unit of progress from just advancing to the next stage to creating a line of working code. So like the canonical Extreme Programming example: Extreme Programming is an agile methodology as diagrammed here. It's something like a big company needs a new payroll system and so when you're building payroll, you don't really have to ask what problem are we trying to solve. OK. Every company makes payroll or it's going out of business pretty soon and so what you would do in. I know traditional waterfall, even those projects fail. But under agile we would actually collapse the feedback cycle time so we would take an in-house customer, a product owner, who knows a lot about payroll. And we'd sit them to the engineers who are building the products so that if they have a question, like how does the deferred amortization work? Or how should the screen look? Or what is payroll, seriously? How does it work? They have somebody they can turn to and get authoritative answers, "Excuse me, in-house customer, can you explain this to me?" And then they have a dialogue on the spot, at the moment that the question arises.

And so under agile, we can do big IT projects substantially better because we're living in a world where the problem is known. It's the solution that's unknown. And if only startups live in this world, we'd be fine. But, of course, this is what it looks like in Startup Land, where both the problem and the solution are unknown. So let's say you want to do Extreme Programming, who are you going to sit next to the engineers, if you don't even know who the customer is? Startups faces problem that there is no authoritative answer yet to the questions they want to ask. And so what they need to do is combine an iterative process of customer development with an iterative process of agile development tied together in a company-wide feedback loop. And what's interesting about this is, it changes the unit of progress in an interesting way.