From: The Elements of Scrum

Sims, Chris; Johnson, Hillary Louise (2011-02-15). Dymaxicon.

In The Beginning: The Waterfall Method

In 1901, a 63 year-old daredevil named Annie Edson Taylor decided to go over Niagara Falls in a barrel for no obvious reason whatsoever. She survived with only a few bruises and gashes and declared, upon emerging, “I would sooner walk up to the mouth of a cannon, knowing it was going to blow me to pieces, than make another trip over the falls.”

If you’ve ever worked on a big, messy enterprise-level software project that used the “waterfall” method, you probably know exactly how Annie felt. Surprisingly, however, the term waterfall does not owe itself to frustrated developers’ identification with Annie’s misadventure. Winston W. Royce first presented what is now known as the traditional waterfall method in a 1970 paper delivered at IEEE WestCom, an engineering conference. Royce didn’t use the term waterfall, but he did describe a sequential process wherein each phase is completed before the next is begun. What you might not know is that Royce offered up this model as an example of how not to do software development!

Royce went on to say that of course one would never want to run a software project this way— and he next described an iterative process, much like today’s agile methodologies, which he declared to be categorically superior. Still, somehow, the description of what would come to be known as waterfall clicked with his audience and became widely talked about.

The event that cemented waterfall’s status as the trusted model for all enterprise-scale software development projects was the US Department of Defense’s adoption, in 1985, of the waterfall method as the official standard for all projects carried out on the DoD’s dime, whether by government agencies or independent defense contractors.

By the turn of the 21st Century, even the government had begun to get an inkling that the waterfall method might be flawed. An official 2005 NASA document describing the method noted that, “The standard waterfall model is associated with the failure or cancellation of a number of large systems. It can also be very expensive.” The document went on to mention that something called “eXtreme Programming” looked quite promising.

Four years later, NASA gave a flurry of press interviews to announce that their engineers had devised their very own agile methodology called “Extreme Programming Maestro Style.” We know, it sounds like something you would order with a side of fries at In-N-Out Burger, but NASA used it to write the software to control the Mars lander robot. Pretty cool, eh?

Waterfall Defined

The waterfall method for developing and delivering enterprise software projects breaks the process into discrete stages like:

1. requirements-gathering
2. design
3. coding
4. testing

In a waterfall process, each step must be completed before moving on to the next, and all steps in the process must be completed before any value is delivered to the customer. The development process literally flows from one stage to the next, moving the project inexorably downhill (often in every sense of the word).

Proponents of the waterfall method do, of course, have a rationale for the way they like to do things. For starters, waterfall lends itself to scheduling and reporting, allowing CEOs, CFOs, corporate attorneys and other stakeholders to use familiar tools and processes when it comes to writing contracts and allocating budgets. Make no mistake, getting these constituents to embrace change can be a lot more challenging than getting the most entrenched project managers and developers to come around to a more agile point of view.

On the design side, waterfall proponents cleave to a philosophy known as big design up front (BDUF), which is common to many plan-driven software development methodologies (The phrase and acronym are most often used by BDUF's detractors, spoken with a bit of a lip-curl, much the way some students use the phrase Big Man On Campus, or BMOC, to describe the "dumb jocks" at their school).

A common argument in favor of BDUF is that by "perfecting" the design before moving on to implementation, one can catch errors and bugs early, which reduces costs over the life of the project.

The fly in the ointment is that rather unrealistic word: perfecting. Now, if you are manufacturing an automobile, then a good case can indeed be made for getting your tooling right before moving into production. It is easy enough to ensure that your fenders will line up with your body panels when everything is still on paper, thus avoiding having to re-cast expensive dies and hold up the entire production process when you discover too late that they don't fit.

BDUF is predicated on the notion that it is possible to "perfect" a product's design before moving into production. And that may be true when it comes to car fenders... but software products are complex systems, not static objects, and systems designed in the absence of any real experiential data are famous for generating a muddle of unintended consequences— before they fail, leaving you with a big mess to clean up. "Communism is like Prohibition, it's a good idea but it won't work," Will Rogers famously said, and you could say much the same about BDUF.

In software development terms, this means that you can sit at the drawing board all day long, creating breathtakingly elegant theories that are a delight to behold— but the moment you begin putting it into practice, Whoa Nelly!— all kinds of unexpected consequences and complications begin to emerge. Worse yet, down the road your customer may just end up doing battle with the software equivalent of the Russian mafia.

Enter the Agilistas

Right now it's only a notion, but I think I can get the money to make it into a concept, and later turn it into an idea. ~ Woody Allen

In 2001, seventeen super-geeks gathered at the Snowbird ski resort in Utah to explore a shared hunch about the future of software development. They included proponents of nascent methodologies like scrum, extreme programming, crystal, feature-driven development, and "others sympathetic to the need for an alternative to documentation driven, heavyweight software development processes," according to Jim Highsmith, who set down the events in writing for posterity. He fondly pointed out that "a bigger gathering of organizational anarchists would be hard to find."

Those assembled agreed upon a name for their movement: "agile." They dubbed themselves the Agile Alliance, and drafted an Agile Manifesto: a brief set of statements that would serve as the new movement's Declaration of Independence, Constitution and Bill of Rights all rolled into one. This napkin-sized document maps the common philosophical ground the Alliance members discovered over that weekend. What the members did not do was seek to codify any one set of practices or methods.

"The agile movement is not anti-methodology," Highsmith wrote, "in fact, many of us want to restore credibility to the word methodology. We want to restore a balance. We embrace modeling, but not in order to file some diagram in a dusty corporate repository."

None of this happened in a vacuum. The Agile Alliance was a reaction to the way software projects were commonly managed: development processes like waterfall that break planning, design, development and testing into a set of discrete steps, one after the other— down which development flowed freely and smoothly like water over Niagara… until crashing into the rocks at the bottom.

The times were ripe for change. In 1995 the Standish Group's annual "Chaos" report had detailed the shocking failure of traditional software development methods to deliver. According to the report, only 16% of traditionally-run enterprise software projects came in on time and on budget; 31% of projects would be cancelled, while 53% would run 189% over their original budgets. When surveyed as to why their projects failed so hard and so often, the number one cause IT managers cited was "lack of user involvement," with "incomplete requirements" a close second. That's right, even BDUF was not able to

provide adequate requirements-gathering, despite the procedural emphasis placed on that phase of development.

Our founding Alliance members, despite their perhaps romantic penchant for referring to themselves as "anarchists," came from the ranks of those disgruntled IT managers who had seen and experienced the waterfall method's failings in action. They were experienced hands, not theorists, and they knew what worked, and what didn't.

Alistair Cockburn, a British IT strategist residing in Salt Lake City, had been working on a new methodology he called "crystal," based on his observation that the problem with rational, linear methodologies is that human beings are essentially non-linear— and all software development is done by humans. "We have been designing complex systems whose active components are variable and highly non-linear components called people, without characterizing these components or their effect on the system being designed," Cockburn told an audience of systems scientists and other technologists at a conference in 1999. "Upon reflection, this seems absurd, but remarkably few people in our field have devoted serious energy to understanding how these things called people affect software development."

At Chrysler Corporation, Kent Beck and Ron Jeffries had been collaborating with people like Ward Cunningham, the inventor of the wiki, on a developer-centric methodology known as "extreme programming," which included practices like test-driven development and pair programming.

And Jeff Sutherland, John Scumniotales, Jeff McKenna and Ken Schwaber had all been developing yet another iterative methodology they called "scrum."

These and other early agile theorists were all present at Snowbird, and they had all come to believe independently that iterative methodologies were the future.

The Iterative Method

One key problem with BDUF is that it assumes perfect knowledge of the future. But anyone who has worked on an enterprise-scale software project knows that the only thing you can count on is change. Agile processes of all kinds share one thing: they embrace change, approaching it as an opportunity for growth, rather than an obstacle.

Agile teams do the same development work that waterfall teams do, but they do it very differently. The agile development cycle employs the same functions as the waterfall method: requirements-gathering, design, coding and testing.

The simple view into how agile development differs from waterfall development is this: an agile team, instead of completing each step before moving on to the next one never to return, does a little bit of requirements gathering, a little bit of design, coding and testing, and delivers a little bit of value to the customer. Then the team does it all over again… and again, refining and tweaking processes as work progresses, until the project is complete.

But incremental, iterative development changes not just when you do things, but how you do them. Agile iterations (called “sprints” in scrum) are not miniature waterfalls; in agile processes, there really are no steps. Agile development is a holistic process, meaning that testing, design, coding and requirements gathering are fully integrated, interdependent processes. Testing, for example, is folded into the design process. Requirements aren’t simply gathered; instead, a deep, shared understanding of them is cultivated through constant communication between the team, the product owner and the customer.

But what does this look like in practice? How do you do agile development? Whether you adopt scrum, lean, extreme programming, or create your own melange of several agile methodologies, you will:

Test as you go, not at the end— a bug fixed now is cheaper than one that has had a chance to propagate through a system for months.

Deliver product early and often, as only by demonstrating working software to your customer can you find out what they really want. Because agile processes include constant feedback from customers, projects stay relevant and on track, and because each increment is complete upon delivery, agile development serves to mitigate risk: should a project be cancelled, then the customer may still use the software delivered to date.

Document as you go, and only as needed. When you bake the documentation into your process, you only write documentation that is relevant and useful.

Build cross-functional teams to break down silos, so that no individual or department can become a process or information bottleneck.

The main idea behind the agile approach is to deliver business value immediately, in the form of working software, and to continue delivering value in regular increments. As we’ll see in the next chapter, the benefits a business can realize from doing development work iteratively are both immediate and cumulative.