What is agile methodology? Modern software development explained

Enterprises need software competency to deliver winning digital experiences. Agile development is how enterprises get there

By Isaac Sacolick
Contributing Editor, InfoWorld
FEB 25, 2020 3:00 AM PST

Table of Contents

Every technology organization today seems to practice the agile methodology for software development, or a version of it. Or at least they believe they do. Whether you are new to agile application development or you learned software development decades ago using the waterfall software development methodology, today your work is at least influenced by the agile methodology.

But what is agile methodology, and how should it be practiced in software development? How does agile development differ from waterfall in practice? What is the agile software development lifecycle, or agile SDLC? And what is scrum agile versus Kanban and other agile models?

[DevSecOps: How to bring security into agile development and CI/CD]

Agile was formally launched in 2001 when 17 technologists drafted the Agile Manifesto. They wrote four major principles for agile project management, with the goal of developing better software:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Before agile: The era of waterfall methodology

Old hands like me remember the days when the waterfall methodology was the gold standard for software development. The software development process required a ton of documentation up front before any coding started. Someone, usually the business analyst, first wrote a business requirements document that captured everything the business needed in the application. These business requirement documents were long, detailing everything: overall strategy, comprehensive functional specifications, and visual user interface designs.

Technologists took the business requirement document and developed their own technical requirements document. This document defined the application's architecture, data structures, object-oriented functional designs, user interfaces, and other nonfunctional requirements.

This waterfall software development process would finally kick off coding, then integration, and finally testing before an application was deemed production ready. The whole process could easily take a couple of years.

We developers were expected to know "the spec," as the complete documentation was called, just as well as the documents' authors did, and we were often chastised if we forgot to properly implement a key detail outlined on page 77 of a 200-page document.

[Learn Java from beginning concepts to advanced design patterns in this comprehensive 12-part course!]

Back then, software development itself also wasn't easy. Many development tools required specialized training, and there wasn't anywhere near the open source or commercial software components, APIs, and web services that exist today. We had to develop the low-level stuff such as opening database connections and multithreading our data processing.

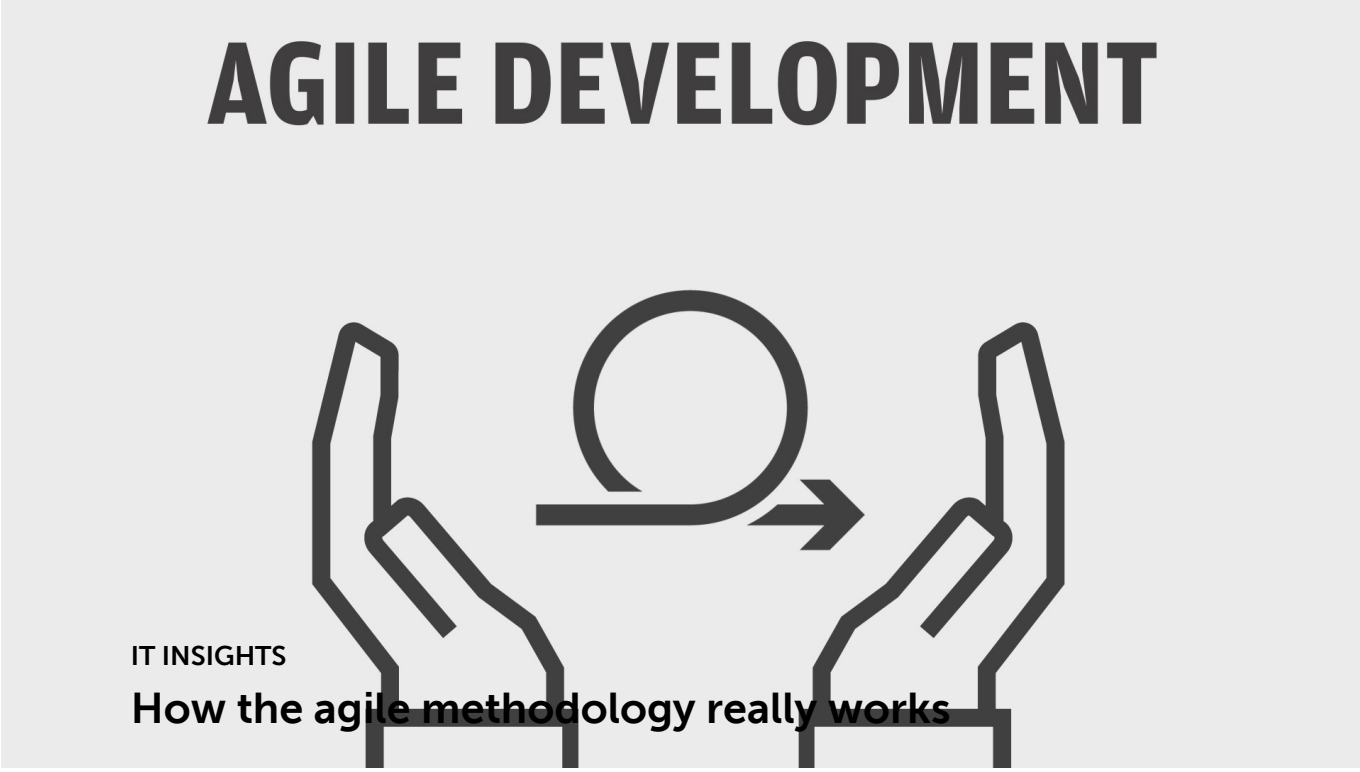
For even basic applications, teams were large and communication tools were limited. Our technical specifications were what aligned us, and we leveraged them like the Bible. If a requirement changed, we'd put the business leaders through a long process of review and sign off because communicating changes across the team and fixing code was expensive.

Because the software was developed based on the technical architecture, lower-level artifacts were developed first and dependent artifacts afterward. Tasks were assigned by skill, and it was common for database engineers to construct the tables and other database artifacts first, followed by the application developers coding the functionality and business logic, and then finally the user interface was overlaid. It took months before anyone saw the application working and by then, the stakeholders were getting antsy and often smarter about what they really wanted. No wonder implementing changes was so expensive!

Not everything that you put in front of users worked as expected. Sometimes, users wouldn't use a feature at all. Other times, a capability was widely successful but required reengineering to support the necessary scalability and performance. In the waterfall world, you only learned these things after the software was deployed, after a long development cycle.

Related video: How the agile methodology really works

Everyone seems to be talking about agile software development, but many organizations don't have a firm grasp on how the process works. Watch this five-minute video to get up to speed fast.



The pivot to agile software development

Invented in 1970, the waterfall methodology was revolutionary because it brought discipline to software development to ensure that there was a clear spec to follow. It was based on the waterfall manufacturing method derived from Henry Ford's 1913 assembly line innovations, which provided certainty as to each step in the production process to guarantee that the final product matched what was spec'd in the first place.

[Also on InfoWorld: 3 ways to kick off a devops program]

When the waterfall methodology came to the software world, computing systems and their applications were typically complex and monolithic, requiring a discipline and clear outcome to deliver. Requirements also changed slowly compared to today, so large-scale efforts were less

problematic. In fact, systems were built under the assumption they would not change but would be perpetual battleships. Multiyear timeframes were common not only in software development but also in manufacturing and other enterprise activities. But waterfall's rigidity became an Achilles heel in the internet era, where speed and flexibility were required.

Software development methodology began to change when developers began working on internet applications. A lot of the early work was done at startups where teams were smaller, were colocated, and often did not have traditional computer science backgrounds. There were financial and competitive pressures to bring websites, applications, and new capabilities to market faster. The development tools and platforms changed rapidly in response.

This led many of us working in startups to question waterfall methodology and to look for ways to be more efficient. We couldn't afford to do all of the detailed documentation up front, and we needed a more iterative and collaborative process. We still debated changes to the requirements, but we were more open to experimentation and to adapting to end-user needs. Our organizations were less structured and our applications were less complex than enterprise legacy systems, so we were much more open to building versus buying applications. More importantly, we were trying to grow businesses, so when our users told us something wasn't working, we more often than not chose to listen to them.

Our skills and our abilities to innovate became strategically important. You could raise all the money you wanted, but you couldn't attract talented software developers able to work with rapidly changing internet technologies if you were going to treat them as subservient coders slavishly following "the spec." We rejected project managers coming in with end-to-end schedules describing what we should develop, when applications should ship, and sometimes even how to structure the code. We were terrible at hitting the three-month and six-month schedules that the waterfall project managers drafted and unceasingly updated.

Instead, we started to tell them how internet applications needed to be engineered, and we delivered results on a schedule that we drew up iteratively. It turns out we weren't that bad at delivering what we said we would when we committed to it in small, one-week to four-week intervals.

In 2001, a group of experienced software developers got together and realized that they were collectively practicing software development differently from the classical waterfall methodology. And they weren't all in startups. This group, which included technology luminaries Kent Beck, Martin Fowler, Ron Jeffries, Ken Schwaber, and Jeff Sutherland, came up with the Agile Manifesto that documented their shared beliefs in how a modern software development process should operate. They stressed collaboration over documentation, self- organization rather than rigid management practices, and the ability to manage to constant change rather than lock yourself to a rigid waterfall development process.

From those principles was born the agile methodology for software development.

The roles in the agile methodology

An agile software development process always starts by defining the users and documenting a vision statement on a scope of problems, opportunities, and values to be addressed. The product owner captures this vision and works with a multidisciplinary team (or teams) to deliver on this vision. Here are the roles in that process.

User

Agile processes always begin with the user or customer in mind. Today, we often define them with user personas to illustrate different roles in a workflow the software is supporting or different types of customer needs and behaviors.

Product owner

The agile development process itself begins with someone who is required to be the voice of the customer, including any internal stakeholders. That person distills all the insights, ideas, and feedback to create a product vision. These product visions are often short and straightforward, but they nonetheless paint a picture of who the customer is, what values are being addressed, and a strategy on how to address them. I can imagine Google's original vision looked something like "Let's make it easy for anyone with internet access to find relevant websites and webpages with a simple, keyword-driven interface and an algorithm that ranks reputable sources higher in the search results."

We call this person the *product owner*. His or her responsibility is to define this vision and then work with a development team to make it real.

To work with the development team, the product owner breaks down the product vision into a series of user stories that spell out in more detail who the target user is, what problem is being solved for them, why the solution is important for them, and what constraints and acceptance criteria define the solution. These user stories are prioritized by the product owner, reviewed by the team to ensure they have a shared understanding on what is being asked of them.

Software development team

In agile, the development team and its members' responsibilities differ from those in traditional software development.

Teams are multidisciplinary, composed of a diverse group of people with the skills to get the job done. Because the focus is on delivering working software, the team has to complete end-to-end functioning applications. So the database, business logic, and user interface of *part* of the application is developed and then demoed—not the whole application. To do this, the

team members have to collaborate. They meet frequently to make sure everyone is aligned on what they are building, on who is doing what, and on exactly how the software is being developed.

In addition to developers, software development teams can include quality assurance (QA) engineers, other engineers (such as for databases and back-end systems), designers, and analysts, depending on the type of software project.

[Also on InfoWorld: How to improve CI/CD with shift-left testing]

Scrum, Kanban, and other agile frameworks

Many agile frameworks that provide specifics on development processes and agile development practices, aligned to a software development life cycle.

The most popular agile framework is called *scrum*. It focuses on a delivery cadence called a *sprint* and meeting structures that include the following:

- Planning — where sprint priorities are identified
- Commitment — where the team reviews a list or backlog of user stories and decides how much work can be done in the sprint's duration
- Daily standup meetings — so teams can communicate updates on their development status and strategies)

Sprints end with a demo meeting where the functionality is shown to the product owner, followed by a retrospective meeting where the team discusses what went well and what needs improvement in their process.

Many organizations employ scrum masters or coaches to help teams manage the scrum process.

Although scrum dominates, there are other agile frameworks:

- Kanban works as a fan-in and fan-out process where the team pulls user stories from an intake board and funnels them through a staged development process until they are completed.
- Some organizations adopt a hybrid agile and waterfall approach, using agile processes for new applications and waterfall for legacy ones.
- There are also several frameworks to enable organizations to scale the practice to multiple teams.

While agile frameworks define process and collaboration, agile development practices are specific to addressing software development tasks performed in alignment with an agile framework.

So, for example:

- Some teams adopt pair programming, where two developers code together to drive higher quality code and to enable more senior developers to mentor junior ones.
- More advanced teams adopt test-driven development and automation to ensure the underlying functionality delivers the expected results.
- Many teams also adopt technical standards so that the developer's interpretation of a user story doesn't lead to just the functionality desired but also meets security, code quality, naming conventions, and other technical standards.

Why the agile methodology is better

What is agile methodology? Modern software development explained

Enterprises need software competency to deliver winning digital experiences. Agile development is how enterprises get there

By Isaac Sacolick
Contributing Editor, InfoWorld
FEB 25, 2020 3:00 AM PST

Table of Contents

Page 2 of 2

When you take the aggregate of agile principles, implement them in an agile framework, leverage collaboration tools, and adopt agile development practices, you usually get better-quality applications, faster-developed applications, and better technical methods (aka *hygiene*).

The core reason is that agile is designed for flexibility and adaptability. You don't define all the answers up front as you do in the waterfall method, but break the problem into digestible components that you then develop and test with users. If something isn't working well or as expected, or if the effort reveals something that hadn't been considered, you can adapt the effort quickly to get back on track quickly—or even change tracks if that's what's needed. Agile lets each team member contribute to the solution, and it requires that each takes personal responsibility for his or her work.

For many problems, agile development is better because its principles, frameworks, and practices are designed around today's operating conditions. Agile frameworks and development processes that prioritize delivering working software iteratively and promote leveraging feedback to improve the application and process is more suitable to today's world of operating smarter and faster.

Product owners may *think* they know exactly how they want to develop an application that fulfills their vision, but rarely do they want to give up the ability to improve that vision as they talk to more users and see how an application actually performs for them. Likewise, development teams *think* they know how to engineer the perfect application, but they can't demonstrate it until the entire application is integrated, functionality demonstrated, and changes in requirements rationalized.

Agile development is also better because it encourages an ongoing process of improvement. Imagine if Microsoft ended Windows development after version 3.1 or Google stopped improving its search algorithms in 2002. Software is in constant need of being updated, supported, and enhanced; agile processes that are iterative in nature establish both a mindset and process for that continuous improvement.

Finally, agile development is better because people on the team are more productive and happier working with this process. Engineers have a say in how much work they are taking on, and they are proud to show their results. Product owners like seeing their vision expressed in software sooner and being able to change prioritize based on the latest insights. Users like getting software that does what they actually need, in a way that meets or enhances their processes.

Today, enterprises need a high level of software competency to deliver exceptional digital experiences in a hypercompetitive world. And they need to attract and keep great talent to build great software. Agile development helps enterprises do both.

Isaac Sacolick is the author of Driving Digital: The Leader's Guide to Business Transformation through Technology, which covers many practices such as agile, devops, and data science that are critical to successful digital transformation programs. Sacolick is a recognized top social CIO, a long-time blogger at Social, Agile and Transformation and CIO.com, and president of StarCIO.

Follow

/