Yes, You Can Develop Software Using Agile Methodology – Part 1

Date:	Mar 1, 2016
Title:	Yes, You Can Develop Software Using Agile Methodology – Part 1
Code:	AR-AM1
Abstract:	*Even if it is "fitting a square peg in a round hole," there's a belief that the software methodology used doesn't really have a direct impact to results. Great programmers always make great products, right? Nothing could be further from the truth. Let's look at how software development team process fits into the development of the new embedded, smart device frontier.*
Version:	1.0 (2016-03-01) Original

The Software Lifecycle Landscape

With the development of hardware systems, careful attention is given to the design and creation of highly detailed specifications that can be used to source board components. This is usually followed by a phased delivery set of milestones including: prototype, implementation, test supplier qualification and final release to production. This regimen offers the advantage of ensuring quality and functional requirements are met by the time manufacturing. Taking a *waterfall* approach to hardware development can minimize any risk of downstream issues once hardware goes into volume production.

Software development is perceived to be completely different. The "soft" part in *soft*ware implies there is an inherent ability to change along the way. Unfortunately, this can give management a false impression that software projects can easily accommodate change with little to no additional cost or schedule impact. The perception of a software developer is further glamourized in Dilbert cartoons by Scott Adams. [1] A programmer is considered to be a breed apart—working in a world combining art, technical inquisitiveness, social discomfort and sometimes even magic. All you need is creative talent, endless hours, lots of coffee, a PC, and a handful of software tools.

As hardware development projects are perceived to complete on a timely basis, software development projects are often late and cost much more than was originally planned. Just ask any project manager in the embedded industry who has worked on both hardware and software projects. Given the choice, many project managers would rather manage hardware projects.

Since both software and hardware projects may be under development at the same time, both sets of teams must collaborate. As hardware developers tend to work in a very structured and organized manner, software engineers tend to perform a lot more trial and error. This "software way of life" can be quite a challenge for management and I've seen more than one attempt in my career to convert more-disciplined hardware engineers to become programmers. This rarely works due to different skill sets and a radically different mindset.

Rather than demand that software people conform to hardware design approaches, a better way is to take lessons from enterprise and mobile software computing. We must identify the right approach for software development that marries the way software developers like to work with how work is performed.

Audit Your Software Development Lifecycle

Years back, a client's development team was having a tough time juggling incoming product management requests making project schedules difficult to achieve. At the time, any mention of changing our process methodology to agile (specifically, Scrum) was met with, "no way! That stuff is just a fad."

I found it was easier to let the team identify what doesn't work rather than force fit some newfangled agile methodology on them. So, I asked the team, "OK, what can we improve with our software development process?" Through rather intense brainstorming, we were able to compile a list of improvements to be made:

1. Decisions made throughout a project lifecycle appear ad hoc and reactive.
2. QA gets involved way too late.
3. By the time the customer sees the product, it isn't what they originally expected.
4. Some features being developed aren't getting completed on time. We usually find this out way too late to course correct.
5. Handoffs at key milestones are ill-defined and by the time the project is completed, the original specifications and design documents aren't even close to what was actually implemented. And yet, we never have time to update them.
6. Requirements are often too abstract with little to no mention of priority or guidance for validation.

Comparing Project Management Approaches

In the software development community, there are two competing schools of thought for managing the process of motivating a team to deliver quality products: waterfall and agile. Figure 1 illustrates the differences between the two extremes with waterfall being more structured and less adaptable, while agile (Scrum and eXtreme Programming shown) is less structured and more adaptable.

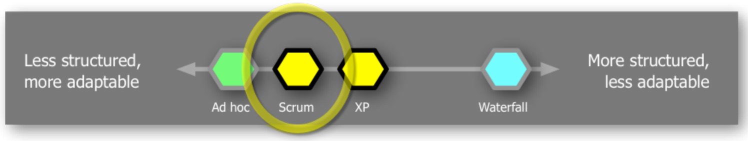
Less structured, more adaptable
Ad hoc
Scrum
XP
Waterfall
More structured, less adaptable

Figure 1: *Agile versus traditional waterfall*

As mentioned earlier regarding hardware projects, waterfall is a step-by-step, phased delivery approach. Completing one step should advance you to the next step as shown in figure 2.

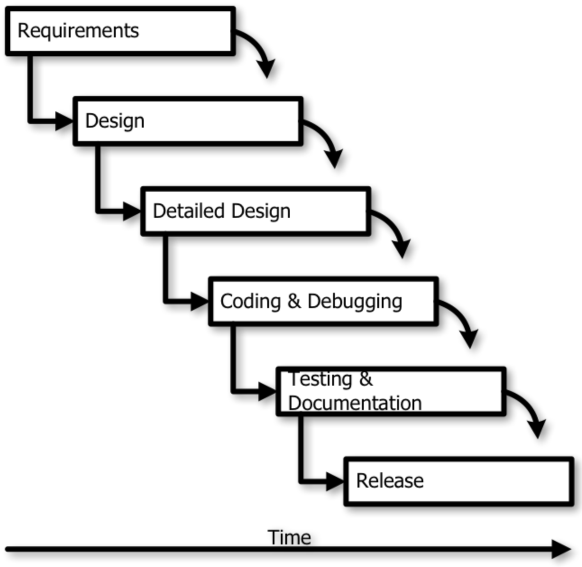

Figure 2: *A waterfall overview*

Figure 2 is not exactly linear—instead the phases overlap between milestones. Developers may start the implementation even before the detailed design document has been signed off. Work may spill over to the next milestone work; hence, the term waterfall is appropriate.

For software development, a sequence of phased milestones makes logical sense but, in practice, this approach may not be as intuitive as you might think. According to *Agile & Iterative Development*, the waterfall method has become unattractive to software developers because: [2]

- Users aren't always sure what they want and once they see the work, they want it changed.

- Details usually come out *during* implementation so committing to upfront schedules is not possible.
- Approved specs are rarely accurate by the time a project is completed.
- Disconnected long series of steps with handoffs are typically subjective.
- Success seems far, far away and, in practice, schedules aren't very predictable. This results in the team not seeing any work completion for possibly months. And then the problem begins when QA gets their hands on the code.

To compensate for these issues, the Agile Alliance [3] was formed to focus on time-driven, customer efficiency for the software development industry. The Agile Alliance defined core values from W. Edwards Deming's research on achieving quality and productivity improvements. [4] According to Deming, work can be more efficiently completed with small cross-functional teams focused on immediate quality results.

Each mini-projects takes on work by the team to be completed in fixed-length *sprints*. I've found that two-week sprints work best for major product upgrades and one-week sprints work best for product updates or research projects. Throughout each sprint, the team is focused on planning, doing, studying, and assessing (PDSA) activities.

The software community at large has embraced this approach most notably with Scrum shown in figure 3. Most software development organizations that I know have selected Scrum as our agile methodology to use.

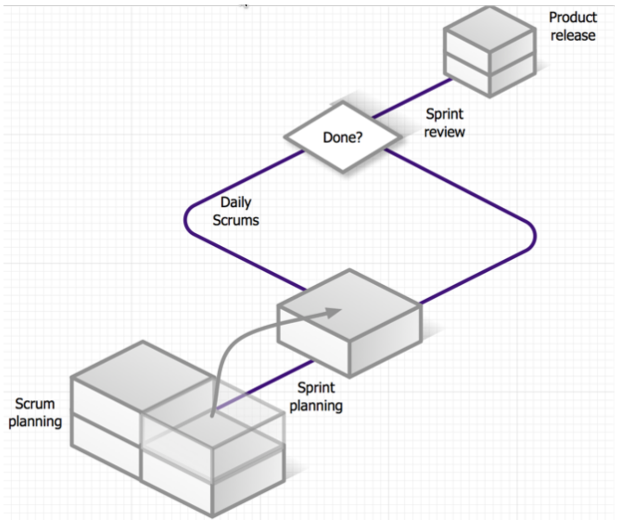

Figure 3: *A Scrum process overview*

Work is defined in T-shirt-sized estimates (small, medium, and all the way up to extra large) at the beginning of a project during a Scrum planning kickoff meeting. The team agrees on resources needed, sprint duration, features (called *product backlogs*) and project vision. And the most important work is taken off the remaining work stack in the first sprint planning meeting. The team breaks the work into tasks and assignments so that by the time the sprint ends, the work should have been assigned, designed, implemented, and validated. To mitigate risks, the team meets every day to review what work has been accomplished, what is left to complete, and identify where assistance may be needed.

Daily meetings may appear to be overkill, but if a daily 15-minute scrum can mitigate project risks, the time is well worth it.

This drastically different approach assumes close-knit collaboration and effective communication. Unfortunately, software developers aren't usually very proactive at communicating and that's where a ScrumMaster steps in. A ScrumMaster must facilitate open discussion, communication and closure. This can take lots of extra time to prepare and communicate to the appropriate stakeholders.

A unique characteristic of Scrumming software development projects is that a product owner and the customer should be represented at Scrum planning, sprint planning, and sprint reviews. During a sprint review, also called a sprint retrospective, the customer has the opportunity to review progress and the team can adjust to that feedback in the next sprint.

In traditional waterfall projects as QA involvement takes place near the end of a project lifecycle adding significant risk. Until all of the critical bugs are found and fixed, the long hours of testing and fixing pushes any schedule out. Instinctively, Scrum projects integrate QA testers as core development team members from the start.

Figure 3 demonstrates the impact of involving QA too late in the process. On waterfall projects, the risk to deliver accelerates, causing long hours and schedule slips.

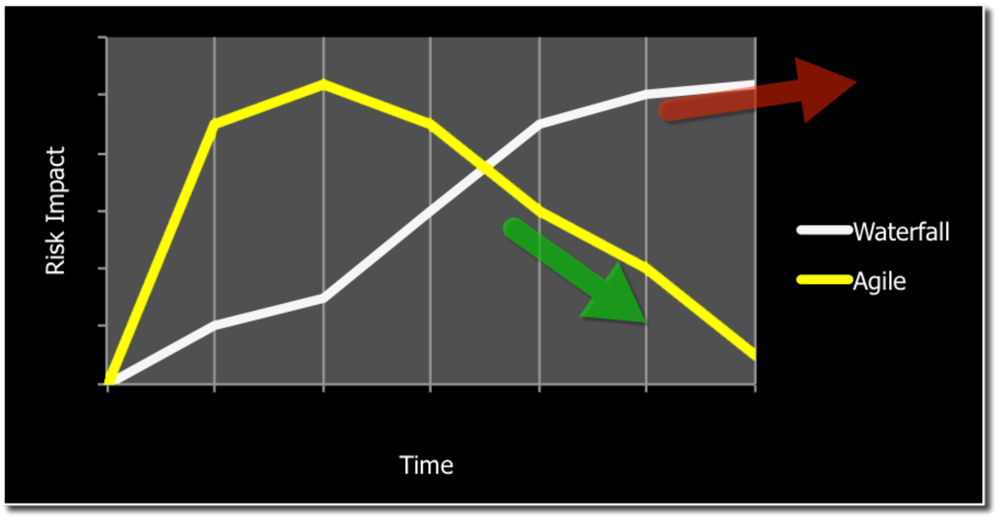

Figure 4: *Comparing schedule risk during the product development lifecycle*

In agile (Scrum), the risk of late schedules due to quality issues should reduce over time. As developers are writing code, the QA testers are busy designing, creating and running tests. I highly recommend investing in test automation to allow code to be tested in nightly builds without human intervention. The use of test automation is critical with Scrum projects because of the need to constantly test. As you might expect, a software project is not released until all tests pass.

Attacking Waterfall Liabilities with Agile

Adopting agile into our software development lifecycle has mitigated many of the liabilities we experienced years ago.

Liability	Agile Benefit
Users aren't always sure what they want … and once they see the work, they'll want it changed.	The customer is involved throughout the process so that adjustments can be made. For embedded projects, code is usually "burned in" so there isn't much chance to update it in the field. It better be right.
Details usually come out during the work.	I've heard this proclamation from software engineers, "I can't estimate the work until I do it." The great thing about agile is that status should be discussed and reported on a daily basis.
Approved specs are rarely accurate by the time a project is completed.	One of the Agile Alliance core values is that "working code is more important than outdated documentation." As a point of fact, how often does your team pull out the documentation after a product has been released? Document what is absolutely needed to design the code and what is required for ongoing support, but not more. An exception is your code going after MISRA C or DO-178C certification.
Disconnected long series of steps with handoffs are typically subjective.	Many years ago when I led software development for process control systems, I couldn't believe how milestones and roles in a project lifecycle would be misinterpreted. What engineering called an alpha milestone ("it sorta works") didn't match expectations from QA ("all features are implemented and unit tested"). Scrum removes these ambiguities by focusing the team to complete work as if the product needs to be released by the end of each sprint.
Success seems far, far away and, in practice, schedules aren't very predictable.	A project team needs to feel some success along the way. The Scrum approach emphasizes interim completion of tasks by the end of each sprint. The added benefit is that these completed features are tested along the way, making early product releases possible.

Table 1: *Addressing typical software development liabilities in an agile way*

Does Agile Guarantee Success?

Adopting a process, any process, never results in instant success. I cringe every time I hear a fellow VP boast, “Yep, we’re going agile and I bet we reduce expenses and meet schedules as a result.” The transition for this team wasn’t an overnight success either. It took well over a year to get the hang of it. Here are my top gotchas to avoid:

- ***Avoid any project restarts by defining and verifying product architecture changes up front.*** The danger about agile projects is that if dramatic structural changes are left until the end to implement, iterative development cycles may delay possible infrastructure issues. Always prioritize working on those tough underlying framework components, first.
- ***Separate uncertainty from focused delivery.*** The nature of many software development projects enters unchartered territory. In other words, the implementation required to build specific features may take a lot of trial and error to figure out. Attempting to keep a team focused *while* investigating implementations is asking for trouble. Instead, why not define a short-term research project with short iterations? This mini-project’s sole mission would be to resolve unknown implementation issues before beginning the actual project.

  I can think of one example years back where a client was planning performance improvements to a client’s file system. Under Linux, there was a need to better integrate the file system into the Linux cache. We determined there were three different approaches we could take. Rather than tie up the entire team, a single engineer worked on a three-week research scrum with one-week sprints to prototype each approach. The last approach he tried worked beautifully and the research project was closed. The team started the actual project the following week.
- ***Performing a home-grown methodology.*** Many software organizations that are unable to commit to agile tend to pick and choose parts of competing methodologies into their own secret sauce. By combining Scrum and waterfall (hence, *scrummerfall*) into a kinda-agile approach never works. It eventually confuses the team and avoids the benefits that a true agile-run project enjoys.
- ***Not handling people issues immediately.*** As a software manager, I spend as much time working with people issues as I do dealing with technical and process activities. One common characteristic of software developers is the inability to get to done-ness (100 percent completion of tasks). At the daily standups, the team witnesses first hand if estimated tasks are not being completed. Take that opportunity to work with the folks who can’t seem to complete work on time. That guidance can help motivate the individual to improve their performance. The worst outcome is not attending to the problem proactively. Who suffers? The team does as they have to pick up the slack to keep the project on schedule.
- ***Lack of coordination between software and hardware engineers.*** If the hardware team’s implementation schedules are out of sync with the software team’s agile schedule, one will be waiting for the other. This is especially true with embedded, IoT, or mobile projects. Your job is to ensure both groups have coordinated milestones so that the software supports the hardware (and vice versa).

Software Development in an Agile World

Scrumming has proved to be a great motivator to any software development team by including QA early, breaking up long projects into small subprojects, and involving the team to collectively problem solve on a daily basis. By adopting a collective way to estimate work, it has been my experience that schedules have become much more predictable.

Some of the techniques used to identify resources has allowed time to be reserved for technical support escalation which customers and sales organization appreciate. Since customers are more involved along the way, even custom services engagements use Scrum as a means to frequently communicate among project stakeholders. It is all built into the agile DNA!

What's in store in part 2? *Lots! Setting the project vision, identifying an upfront schedule range, getting buy-in by separating must haves from nice to haves, flipping the iron triangle, motivating the team to get to D-U-N-N, and team motivational techniques that work.*

References

1. Adams, Scott. *Dilbert*. www.dilbert.com.
2. Larman, Craig. *Agile and Iterative Development: A Manager's Guide.* Boston: Pearson Education, 2004.
3. Agile Alliance. *Manifesto for Agile Software Development.* (www.agilealliance.org).
4. Deming, W. Edwards. *Out of the Crisis*. Boston: The MIT Press, 2000.

Bio

Ken Whitaker, of Leading Software Maniacs, has more than 25 years of software development leadership and training experience. He has written books on leadership and is an innovator in instructional design and agile project leadership workshops. Ken is the creator of PM Chalkboard, a software company VP, and the editor for *Better Software* magazine. He is creating a unique gamification product that redefines learning.