

2009

PACIFIC NW SOFTWARE
QUALITY
CONFERENCE



MOVING
QUALITY
FORWARD

OCTOBER 27-28, 2009

Conference Paper Excerpt

From the

CONFERENCE
PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

My Experience at Adopting an Agile Software Development Approach

John Bartholomew
Nethra Imaging, Inc., Beaverton, OR
bart@nethra.us.com

Abstract

This paper presents an overview of the software development process used at Ambric Inc., a fabless semiconductor startup company in Beaverton OR, from roughly 2006 to 2008. During that time, an agile development process was initiated and refined over the course of several major software releases, yielding significant improvements in both product quality and in reliability of release delivery dates. An analysis of the direct benefits of this approach and additional insights gained are included.

Our team of ten software tool developers and four quality assurance engineers produced six major releases in roughly two years. In addition, a smaller application development team and their associated quality assurance engineers also used an agile approach in delivering their releases. Only the tool team's efforts will be covered in this paper. For both teams, the resulting products supported Ambric's massively parallel processor array (MPPA) chip, which allowed our customers to create and debug designs to program the chip's 344 processors and on-chip memory. Customer and internally produced applications were in the areas of video encode/decode, encryption/decryption, and image and signal processing in several domains.

Biography

John has worked in the EDA (Electronic Design Automation) and semiconductor software tools industries for over 20 years in the Portland OR area, working primarily on simulation and co-simulation toolsets. He has also served as an adjunct faculty member at the Oregon Institute of Technology, and Portland Community College.

While the company described in this paper, Ambric, closed in November 2008, its intellectual property was purchased by Nethra Imaging of Santa Clara CA in early 2009. John and several former Ambric employees are now continuing to develop the Ambric multi-processor array chip technology and software toolset at Nethra. John is currently the Senior Software Architect for the the Ambric tool set at Nethra Imaging.

John has an M.S and B.S in Electrical Engineering/Computer Science from the Massachusetts Institute of Technology.

The Scrum Methodology

The development model we adopted was based heavily on the agile programming practice known as “Scrum”. [1] Scrum can be considered a lighter weight alternative to the more radical XP (Extreme Programming) approach. [2] Where XP focuses more on the details of programming aspects of the development process, Scrum is concerned with the methodology and framework of that process – i.e., how your team effort is organized and executed, not the tactics of the daily work flow.

A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an empirical approach—accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly and respond to emerging requirements. [3]

In the Scrum methodology, a *Product Owner* ensures that the development team has the right input from the company's business perspective. For us, this meant that management (including sales/marketing) negotiated the general content of a small number of releases into the future, with software team leads quickly estimating the effort required to design, implement and test new features. The result was a short prioritized list of planned new features per quarterly release, based on actual and expected customer need. This list is referred to as the *product backlog*. These new features were not set in stone at this point, but served as the prioritized initial road map for the development team when starting a new release cycle. If the priority of a scheduled feature changes during the course of the release, the Product Owner must bring this fact to the attention of the team. The team will then alter their development plans accordingly. This flexibility is one of the principle benefits of an agile approach; the iteration planning process must be thorough but also lightweight enough to easily accommodate such in-flight changes. The role of Product Owner was split between our VP of Software Engineering and our marketing Product Manager.

In Scrum, the basic unit of team effort is called an *iteration* or a *sprint*. Each iteration is a few weeks in length, often in the two to four week range. One of the main goals of Scrum is that each iteration adds well tested increments to the overall product functionality, as shown in Figure 1, resulting in a new shippable product release at the end of each iteration.

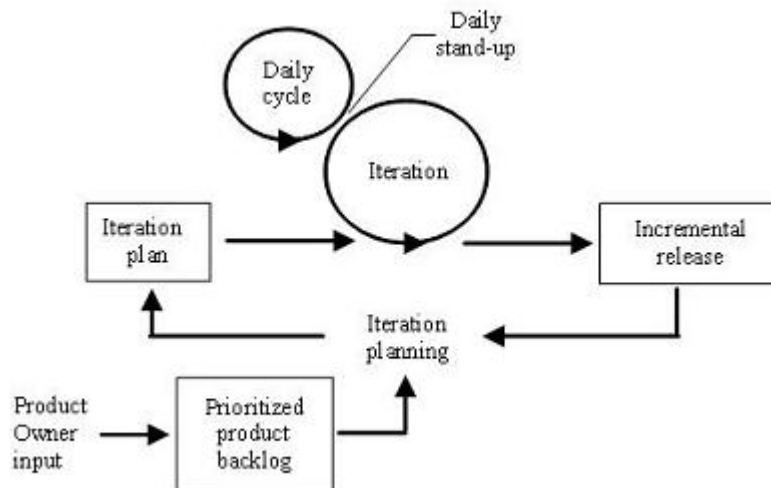


Fig. 1 The Scrum process [1]

In order for our product to have major releases available once each quarter, a release typically included six iterations, with the final iteration being three weeks long to allow completion of more extensive release testing. At the start of the first iteration, teams would work through the design of all new features in the release in detail with a task breakdown in order to verify that implementation and testing of these features could be completed within the scope of that release. If not, the release contents had to be renegotiated with the Product Owner or resources had to be reassigned as needed. Our software team manager filled the *Scrum Master* role, tracking progress towards team goals and performing any needed renegotiation or reassignment. This worked well within our team, due to the smaller size and lack of organizational hierarchy common in most startups. It may not be generally advisable, however, due to the possible conflicting roles of team manager (directing work, providing performance reviews) and Scrum Master (facilitating development but not directly assigning it).

A typical quarterly release structure for our product is shown in Table 1, and our development iteration's structure is shown in Table 2.

Iteration 1	Feature planning, design, initial documentation; development start
Iteration 2	Development/test
Iteration 3	Development /test
Iteration 4	Development /test
Iteration 5	Development /test
Iteration 6 [three weeks]	Release testing, final bug fixing (no new feature code), release packaging

Table 1. Typical Scrum release structure

	Monday	Tuesday	Wednesday	Thursday	Friday
Week One	Feature design; coordinate team interdependencies	Develop/test	Develop/test	Develop/test	Develop/test
Week Two	Develop/test	Develop/test	Develop/test	Qualified release build/test	Fix outstanding issues, if any

Table 2. Scrum iteration structure

Scrum-based methodologies often include a brief (15-20 minutes) daily team project status meeting, called a *stand-up meeting*. Based on several team members' prior experience, we chose to hold only two each week, held by individual teams (compiler, debugger, etc.) not the entire software team as a whole. The entire software team would gather only once per week for an overall project status review, as well as once at the start of a new iteration. This latter meeting would serve as a review of the prior iteration's results, as well as a kick-off for the new iteration about to start.

The Scrum Iteration Model

Early on, we decided that all development, including significant test development, would be performed on branches from the main code repository. In this way, both developer implementation and unit testing and QA team testing would be performed on any new feature prior to it being committed to the code repository main line. At first, several team members were unhappy with the decision to use branches; this disliked the extra overhead of branching and merging, and they had typically developed independently and checked directly into the database main line. As it turned out, each sub-team (2-4 members) working on a given feature could be isolated from other teams, yet coordinate their own work, via a common branch.

Branches were kept alive for as short a time as possible, in order to achieve a level of completion that could be tested as non-disruptive to the main line, and then be merged. Longer-lived branches would be periodically updated from the repository main line to ease merging later on.

As part of this practice, QA team members were tightly integrated with development teams, involved as early as reviewing initial documentation during new feature design. This enabled QA team members to more fully understand new features and to complete their test planning in coordination with their associated development team lead. QA team members also used their development team's branch for their test efforts, including running a full regression pass to test for integration failures. New code (features or significant bug fixes) could therefore be developed and more completely tested as a modular unit prior to being included in the code repository main line. This reinforced the Scrum goal of producing a shippable product at the end of each iteration.

If a new feature testing was not yet completed at the feature commitment cutoff point (iteration week 2, Wed. eve), then the responsible developer had to hold off merging any of their changes from their branch to the repository main line until that iteration's qualified build was tested and declared good. This practice tended to eliminate surprises due to the inclusion of buggy, partially implemented/partially tested code checked in to the repository at the last minute. Instead, the slightly-behind-schedule developer would finish the work on their branch over the next day or two. Once the current iteration qualified release was declared completed, the code repository main line was opened once again for committing new code. In the meantime, in any time remaining prior to the start of the next iteration, this lagging developer could start work on features scheduled for that next iteration. The only loss is that the feature originally scheduled to be completed in iteration N is now completed in iteration N+1, but without the quality risk of a hastily completed piece of functionality.

A very important aspect of the development flow was for the Scrum Master to monitor new feature progress during the first half of a release. If any particular team was falling behind schedule, that team's remaining unimplemented features for the current release would be renegotiated with the Product Owner. The Scrum Master can make this determination based on a feature of Scrum toolsets called a *burndown chart*. This is a graph showing the time estimate of the remaining work for the current iteration (the *iteration backlog*) versus the time elapsed. Ideally, this graph should show a convergence towards zero remaining work at the end of the iteration (or release). See Figure 2 for an example.

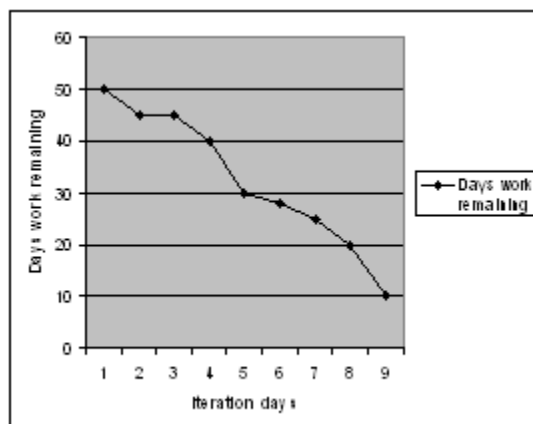


Fig 2. Iteration burndown chart

If one of these jeopardized features could be pushed into the next release, it would be. The idea here was that any customer need for such a feature could be addressed in the short term by an end-of-iteration qualified build early in the following release. (A customer would see only a few weeks' delay in the

delivery of a feature that had slipped, rather than a full quarterly release cycle.) As part of this re-planning process, it was important for the Product Owner to provide a properly prioritized feature list at the start of each release. For risk reduction, release critical features would not be planned for completion by the development team in the second half of a release cycle if at all possible.

An alternative approach to moving features out to a later release was to add an additional iteration(s) to the current release, effectively extending the release date. While the team responsible for the late features could complete their tasks, this had a side effect of delaying other developers. If they were done with their new features and critical bugs for the current release, there was an inherent risk in allowing these developers to continue to add new features or bug fixes into the current release in the final development iteration(s). These other developers could work on planning and implementation for the next release, but they had to work on a branch and not commit any new code to the repository main line until the current release was completed.

Testing

Nightly builds and regression tests were run to ensure that all code checked into the repository main line would build and pass all tests. Any large disruptions in test results were addressed immediately, with one or more developers setting aside current work to trace and implement the needed fix. Small changes to test results (for example, a few new failing tests out of our few thousand regression tests run each night) would be inspected to determine the root cause. A bug report would be filed to be fixed as appropriate within the current iteration (if small) or the current release (if more significant work was involved). In the worst case, the code changes causing the failure could be backed out, and the new feature fixed on a branch prior to being further tested and checked back into the repository when ready.

In addition to regular developer and QA testing, whole team “bug hunts” were included in each release, typically one part way through release, and one near the end. A bug hunt entailed a half-day testing effort where all team members read up on one or more new features (preferably not in their area) and attempted to expose problems. Typical strategies often included corner case tests, invalid or null parameters to APIs/UI dialogs/language constructs, simulating “new user mistakes”, capacity testing, and so on.

The QA team manager held a weekly bug scrub meeting with the team leads to triage recent new bug reports. This meeting served to identify critical bugs which had to be addressed immediately, redirect inappropriately filed reports, detect duplicate bug reports and generally acted as an early warning system for overall release quality. Towards the end of a release cycle, this meeting would be held more frequently and focused only on release critical bugs. All other non-critical bugs were reassigned to be fixed in a future release.

Benefits of this Approach

This model provided the following benefits over our prior, less formalized software development approach:

Thorough testing of new features (including significant bug fix tasks) on a branch by both developers and QA team members significantly reduced the introduction of bugs to the main code repository. This was a significant contributor to the quality level of each iteration's qualified release build. Tight integration of the development and QA teams was required to achieve this, however; development team unit testing alone was not sufficient.

Qualified release builds were able to be produced roughly every two weeks, in addition to a full release every three months. The qualified release builds could be delivered to a customer as an interim release

for critical bug fixes or early access to new features. These frequent small releases provided a sense of consistently accruing new features in the product; our former longer release cycles tended to engender feelings of “there’s still so much to get done before the next release...”.

As we incrementally improved our development process, there was markedly less “death march” feeling to the end of a release (e.g., fixing a sea of critical issues, uncontrollable feature creep, etc.). This provided far more predictable release quality and release delivery date with far less stress and higher team morale.

Our last few releases were delivered with acceptable quality and all or nearly all initially planned features within one week of the originally planned quarterly release date. Hitting multiple three month release cycles within one week of their planned dates provided significant credibility of our approach to our executive management team. It also gave our sales team and field force confidence in our ability to meet promised deliveries.

Key Findings

Refine, refine, refine. It took several releases to refine our use of the iteration model to the point where team members were comfortable and productive. This included being better able to estimate not only a feature's development time but allowing for the testing and associated initial bug fixing phase prior to merging new features into the code repository main line of development. As the development and delivery process improves, it builds trust with your field and sales force as well.

Less isn't always more. We chose a two week iteration length, primarily due to suggestions from our Scrum tool set provider at the time. Some team members, however, felt that a longer iteration time (3 to 4 weeks) might have provided more development time with less test/release overhead. In addition, with a shorter iteration time, new feature development was nearly always performed on a branch, to avoid having partially completed new feature code checked in to the repository main line at the end of the iteration. A longer iteration time would allow developers to use the code repository main line for much of an iteration, only resorting to branches for features started near the end of an iteration.

Be willing to bend the rules – a little. Having the patience and fortitude to declare an iteration's qualified build as incomplete due to a discovered critical issue was hard at first, as it delayed the completion of the current iteration's release – with the harsh side effect of closing the code repository main line to code changes. Extending the current iteration by a day or two to solve such issues, and then completing the qualified build, kept the confidence in our end of iteration qualified builds high and new feature reliability high as well. Having an adaptable process rather than a iron-clad one kept the team morale up.

Managing release quality takes time, and you must budget for it. We were “our own worst critics” - bug hunts would often turn up many small, corner case bugs or integration issues that customers likely would not encounter. Prioritizing these bugs (fix in current release vs. future release) was time consuming but necessary. We realized we simply couldn't both fix all of the issues we'd turned up and deliver all needed new features in the current release cycle. As our product was still early in its life cycle, and many of our customers new to our product's tool set, we favored providing new features over fixing every bug that our team had found internally. To be safe, our internally discovered, lower priority bugs were often left on the fix-in-current-release list until bug scrub meetings near the end of the release, when they were finally either fixed if time permitted or reassigned to be fixed in a future release. The choice of which bugs to fix and which to postpone was best made with the advice of one or more customer advocates, such as a field applications engineer.

Which comes first, the bug fix or the new feature? Managing these reassigned bugs was important in order to avoid a steadily increasing backlog of open bugs. Eventually, we employed team reviews of open bugs during release iteration 1 and included the needed bug fixes in the current release development task

list. Including only new feature development time in a release cycle led only to end-of-release bug fixing frenzies in order to meet our quality goals; allocating time to address the open bug list led to more reliable quality, but at the cost of pushing lower priority new features out into the future.

Verify quality early and often. Our first bug hunts were scheduled towards the end of each release cycle, and would produce a significant number of new bug reports. This tended to disrupt the end of release cycle, both in terms of quality metrics and development flow. We decided to split this one large bug hunt into two or more smaller ones, spread throughout the release cycle, focused on newly completed features. This spread the new feature bug fixing effort throughout the release cycle instead of loading up this effort at the back end.

When is “fixed” really fixed? During one of our first releases, we were far behind on bug verification as the release date approached. I volunteered to verify over 150 “fixed” bugs – and discovered that 1 in 15 was not, in fact, fixed. Some had been partially addressed, and some had regressed since the initial work was done. The longer you delay the verification of a bug fix, the more unexpected bug rework you may encounter at the end of a release. There’s a downside to early verification, however – code changes later in the release cycle may undo a prior fix. To prevent this, regression tests must be created for all non-trivial bug fixes and included in regularly run regression test suites to catch any such regressive behavior.

Stack the feature deck for success. At the start of a release cycle, we would make sure we’d scheduled all of the critical release features to be completed by the end of the third of the five development iterations. Any new features beyond that point – those scheduled in the fourth and fifth iterations – were considered candidates to be bumped out to the next release. This was all agreed to in advance between the team and the Product Owner. If the completion of work on high priority features was delayed, this prior ordering arrangement made it far easier to complete those higher priority features a little late and sacrifice a low priority release feature.

The team benefits as well as the product. The qualified build model really did allow us to have a new release every two to three weeks of reasonable quality for use by the field engineering team, or to act as an “alpha level” interim release to a customer. This boosted team morale as we weren’t constantly struggling with product quality concerns and could feel a constant growth in product features over time. This effect has been noted by other adopters of agile processes as well [4]. Team-wide bug hunts had a positive side-effect of allowing developers to learn about portions of the overall product that they didn’t directly work on. They also turned up incorrect/incomplete documentation issues, as testers worked from the new feature documentation available at the time.

A good workman doesn’t blame his tools – but does prefer good ones. The Rally Software tools [5] helped us formalize our development model as described above. We initially used another scrum-based tool, but with little success. The tool wasn’t really the issue in this case, however, it was our team adhering to and refining the iteration process model, which the Rally tool set supported very well. Having a good bug tracking tool was important, as well, especially towards the end of a release cycle when critical bug reviews were held daily. Here, Bugzilla fulfilled our needs quite well.

Conclusions

Our product was used in the field for demonstrations and delivered to many customers for evaluation, as well as a smaller number for actual use in their own product development. Overall, a very small number of bugs were ever reported by customers; our process was successful in catching nearly all potentially customer-visible issues in house.

Adopting the iteration development model described above took several releases, or the better part of a year, to evolve to the point where the team was both comfortable with it and adept at using it. The idea of pushing a feature out of a release and into the following release was difficult at first for management to accept; the ability to quickly provide that feature in an interim qualified build from an early iteration of that

next release at a delay of only a few weeks never turned out to present an obstacle for a customer, however.

Overall, this development model helped our software team to consistently deliver a tool set that was higher in quality and with far greater reliability of the delivery date and release features than was possible with our prior development process.

References

[1] K. Schwaber, M. Beedle, *Agile Software Development with Scrum*, Prentice-Hall, 2001.

[2] K. Beck, C. Andres, *Extreme Programming Explained*, Addison-Wesley, 2004.

[3] Wikipedia entry for the Scrum development process at [http://en.wikipedia.org/wiki/Scrum_\(development\)](http://en.wikipedia.org/wiki/Scrum_(development))

[4] S. Maguire, *Debugging the Development Process*, Microsoft Press, 1994.

[5] The Rally Software toolset at www.rallydev.com.