From Traditional, to Lean, to Agile Development: 
Finding the Optimal Software Engineering Cycle

Jay Trimble  
NASA Ames Research Center  
jay.p.trimble@nasa.gov

Christopher Webster  
Intel Corporation  
christopher.webster@intel.com

Abstract

In 2008, our team at NASA Ames Research Center launched a five-year project to deliver a user-centric software platform for mission control. We began with a six-month delivery cycle. Within two years we were delivering functional software every three weeks.

Our evolution from traditional, to lean, then agile, did not happen because of a focused goal to become lean or agile. Rather, we responded iteratively to problems; we were disconnected from our customer, the long delivery cycles created issues with testing and verification, and we were unable to effectively measure our progress.

We changed our delivery cycle first to six weeks, then three, with the team focused on the highest-priority features and bug fixes. Our one measure of progress became working code. We delivered a nightly build to our customer. Our QA team tested features as they rolled out.

In our journey from traditional to agile, we tailored our processes to our team culture and our context of work. We found that agile methods increased customer and team satisfaction, and enabled us to use limited team resources where they were most effective – the design and development of the software.

1. Introduction

We detail our experience and lessons learned in going from a traditional to a lean to an agile software engineering practice for delivery of a mission control user application architecture. We explore the impact on the development team, customer relations, product quality, and cost. A noteworthy aspect of this journey is that it was not the result of an intentional switch from traditional to agile. Rather, by iteratively solving problems with the traditional development methods with which we began, we became lean, then agile [1,2,3,4].

In late 2007, the user centered technology team at NASA Ames Research Center began a collaborative project with NASA’s Johnson Space Center to develop a user-composable software technology that would allow NASA flight controllers to compose their own displays using certified software objects representing real-world domain objects on board the International Space Station.

Fresh off a successful prototype, in which the team demonstrated the core concepts of user object composition and acceptable engineering performance in the mission control test facility, the team set out to build the real product. Unlike the prototype, this product would need to be certified for flight operations, and would need to follow a software engineering process that provided accountability. We were given two years by our customer to deliver a first version of a working product, which we called Module 1. The total project lifecycle was to be five years, with a delivery every six months.

We quickly ran into problems. Over the course of six months, we could not effectively verify the state of the software. Customer interaction was limited. The long cycle required extensive testing. There was a significant lag time between designing the software functional specifications and rolling those functions out. The period of months between specification of features with the customer and delivery of those features to the customer created a mismatch in expectations. Worse, the early deliveries were platform services, often with no user-facing components.

2. Traditional Methods: A Six-Month Cycle

At project kick off our team (Table 1) was composed primarily of developers (the number of developers fluctuated, nine was the peak number), two user experience designers, a project manager (PM), and the group lead/principal investigator (PI). The term PI is a holdover from the original NASA proposal process. The user experience designers used participatory design methods to work with the customer to design and specify the product features and functions. The PM was responsible for maintaining
the schedule and associated internal processes and for implementation issues. The PI, who had duties on multiple projects, was responsible for project definition and strategic objectives and maintaining relations with the customer. The PI was the final decision authority if conflicts arose.

Our initial delivery cycle was six months. Figure 1 shows the six-month cycle as it would have looked over a two-year period. Each delivery focused on completion of a subsystem rather than verifiable and meaningful end-user functionality. In part, this was because we were building a platform and core underlying capabilities needed to be developed to make the user-facing components function. However, in this plan lay the roots of some classic software delivery issues, including a lack of customer connection to the product, complicated testing and verification issues, and an inability to respond to situational changes in a timely manner.

Our software functionality was detailed in a user experience (UE) specification. We defined the key UE elements at the start of Module 1. The UE functions were divided into subsystem deliveries. However, the focus of attention during customer discussions was Module 1, and most issues that arose were discussed in the context of the completed Module 1 delivery, which was two years away. This allowed important issues to be deferred because the perception was that if a feature or function was not ready, it would somehow be fixed within the Module 1 timeframe. The lack of user-facing components in the initial six-month subsystem deliverables made it impossible to effectively verify what was working and what was not, and whether what we were building would satisfy the customer requirements. The combination of long time frames and the inability of the customer to see our progress by trying out the software created a growing mismatch in expectations between the customer and the development team. In the customer’s mind the product became a series of specs and promises, not grounded in the reality of working software.

Further complicating this situation was the reality that mission control software must be tested in mission control for users to understand functionality in context. It quickly became clear that four six-month cycles were insufficient to deploy, test, and verify functionality.

<table>
<thead>
<tr>
<th>Traditional</th>
<th>Lean</th>
<th>Agile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developers (5-9)</td>
<td>Developers (7)</td>
<td>Developers (4)</td>
</tr>
<tr>
<td>User Experience Design (2)</td>
<td>User Experience Design (2)</td>
<td>User Experience Design (1)</td>
</tr>
<tr>
<td>Quality Assurance/Process Build Engineers (2)</td>
<td>Quality Assurance/Process Build Engineers (2)</td>
<td>Quality Assurance/Process Build Engineers (.5)</td>
</tr>
<tr>
<td>Project Manager (1)</td>
<td>Project Manager (1)</td>
<td>Project Manager role done by developer</td>
</tr>
<tr>
<td>Principal Investigator (Part Time)</td>
<td>Principal Investigator (Part Time)</td>
<td>Principal Investigator (Part Time)</td>
</tr>
<tr>
<td>Interns</td>
<td>Interns</td>
<td>Interns</td>
</tr>
</tbody>
</table>

Table 1. Team Composition

Figure 1. Our initial delivery cycle, linear, six months per delivery
3. Getting Lean: A Shorter Delivery Cycle

Our initial goal was simple – to shorten the delivery cycle. We thought the shorter cycle would facilitate more frequent customer interaction, a more manageable test load for each deliverable, and the capability to verify our deliveries and their functionality.

The team composition changed to include first one, then two quality assurance (QA) engineers. One of our QA engineers also took the lead in setting up build and check-in processes and exit criteria for delivery. We adopted a team policy that the only true measure of progress is working code.

During the six-month deliveries we frequently had to present our status to management and our customer. Short of demonstrable working code, it’s a challenge to measure, let alone present, the state of a software project. The short delivery cycle enabled us to show our progress with working deliveries, rather than through a series of status briefings.

Figure 2. Shorter delivery cycle

Shortening the delivery cycle gave our team and our customer increased insight into product status. However, the shorter cycles lacked focus. By only shortening the cycles, as shown in Figure 2, we simply divided our existing plan into smaller parts. It was a start, but we found that the cycles lacked clarity of purpose; they were merely incremental subsystem deliveries. We still lacked the ability to deliver meaningful and measurable functionality with which to measure our progress in a way that was meaningful to the customer.

The next step was to clarify the purpose for each cycle. To do this we created a strategic road map with milestones for each year, and within that we developed a focal point for each cycle rooted in end-user functionality visible to the customer. We labeled the cycles as *iterations*, each with a name, such as “plotting” or “tables.” The name reflected the primary functional goals of the cycle, which we would verify through user testing every six weeks. The iterations lead up to a release, which, as of that point in time, was still Module 1. We had been given two years from the start date to complete Module 1, and that was not going to change.

4. Integration of User-Centered Design and Prioritization

Figure 3 shows our updated six-week cycle. The primary goals of this updated cycle were to incorporate our user-centered design practice and to develop a common understanding between developers, the user experience team, and the customer as to what the highest-priority items were, and to assign work based on those priorities.

Prioritization and UE are inseparable. As the purpose of our software is to meet user needs, these needs must be understood to create a meaningful prioritization of work.

Figure 3. Updated and refined six-week process, with focus on constant prioritization of work
4.1. Prioritization

It sounds simple and obvious – always work on the most important issues first. But what is required to achieve this? The team must know the priority of the issues, the issues must be assigned, and changes must be tracked. The natural human tendency to do the easy things first must be resisted. We adopted a two-pronged approach, utilizing both tools and the social structure of the team.

To support prioritization, we added the Greenhopper plug-in to our issue tracking system, JIRA [5]. Greenhopper shows issues ordered by priority and assisted in our prioritization process called stack ranking. During stack ranking, we created an overall priority for each iteration, then an individual list for each developer showing the issues assigned to them in order of priority. The developers were responsible for time estimates for their issues, and for committing to getting them done in the time allotted. Because this was a stack ranking, and software estimation is an inexact science, we tended to assign more work than the developers could complete, with the idea that the ranked list functions as a rollover to-do list. The key goal was getting the whole team focused on working in priority order.

4.2. Ranking Lessons

We learned several lessons along the way. First, don’t over-rank. It is typically not necessary to rank every issue in exact order; rather, it is necessary that developers know their priorities. Next, we started out viewing the ranked list as a rolling to-do list, with the idea that it was okay if the list was much longer than what could be done. Long lists that couldn’t be completed demoralized the team, so we began cutting the lists down to what we thought was achievable in the time allocated to the iteration. We learned that a shorter “can-do” developer list often resulted in an “all-done” list with no rollovers.

The developers’ estimates for time to complete their features were often done at a high level because estimates take time that can be used for coding. On the other hand, a good estimate leads to success, and the developers learned to find the right investment level for time estimates.

The scheduled stack ranking session required significant preparation; we often scheduled two pre-ranking sessions before the final stack ranking. While we had the whole team present for the final ranking session, we did not find it necessary to have the whole team present for pre-rankings.

4.3. Scrum-like meetings

The traditional team meetings were gradually replaced with short tag-up meetings [6]. These meetings are held on Monday, Tuesday, and Thursday mornings and are kept to 15 minutes. Each engineer summarizes his or her tasks, that is, what was accomplished yesterday, what is planned for tomorrow, and whether any blockers are inhibiting progress. Our team lead may interrupt statuses that take too long. (Individual follow-up meetings often occur afterwards.)

5. Iteration Flow

Officially, we allocated 3.5 weeks per six-week iteration for coding. In practice, the developers were always writing code. During the “official” coding time, the QA team wrote their test plans, and the UE team developed the UE spec for the next iteration. The coding time could also be used for user testing of the previous iteration.

5.1 Technical Design

Before the start of coding we allocated four days for the engineers to write a technical spec. This is a simple one-to-two-page description of the feature implementation. It typically describes design trade-offs and the justification as to why a design is chosen. It may also contain use cases and test scenarios. The primary customer for the tech spec was our QA team, who used it to build test plans. The tech spec also served as a vehicle to drive communications within the engineering team as well as providing documentation for the software design required by formal software processes such as concept maturity model integration (CMMI) [7].

5.2 User-Centered Design

Our software is user centered, and we have a series of interactions with users ranging from design sessions to user testing. In Figure 3 you can see how we integrated agile and user-centered design over the six-week cycle. Before the start of each iteration, a UE specification was due. The spec was the developers’ map of what they needed to build. It was supplemented by direct and ongoing interaction with the UE team. During the coding part of our cycle, the UE team was able to conduct user testing and design sessions. The user testing showed users’ responses to the previous iteration(s). We could also user test
future designs using paper prototypes. Design sessions with users facilitated a common language and mental model, the output of which was variable-fidelity mockups used to specify and refine the design of upcoming features.

6. Reporting

Our customer wanted to know what we would deliver when, and how long each feature set would take to implement. Traditional forecasting methods for software would have had us estimating lines of code and creating detailed estimates for how long it would take to develop each key feature set, then using those estimates as a contract with our customer. Rather than expending limited team resources with software estimates, we chose to have the team focus on designing and building software. Instead of committing to specific time estimates for each feature set, we showed our customer exactly where we were every six weeks, with working code. We replaced detailed contractual delivery estimates with a close collaboration that allowed the customer to see our progress.

Our JIRA issue tracking system is automatically linked to our code repository, our tech specs, and test scenarios (Figure 4). Each customer requirement is represented as JIRA “requirement” issue, which is linked to a JIRA “task” issue. We have a trace from customer requirement to test cases and to code, allowing us to provide requirement traceability as well as support CMMI audits.

7. Automation

We use automated unit testing and continuous integration. With each check-in to our code base, the product is built and unit tests are executed. The resulting distribution – the nightly build – is ready for testing by our customer and our QA.

Our product has extensive user-facing features. New features need to be tested and existing features need to be regression tested, but GUI testing is always tricky to automate. Our first attempt at GUI test automation used fixed robot scripting of scenarios. We ultimately rejected this approach because it relied on fixed scenarios. Instead, we now use FestSwing in our automated unit tests. While this does not replace a system-level suite, it achieves some coverage of our GUI elements with high flexibility and simple dependencies.

8. Issues

An important part of lean or agile development is the ability to respond quickly to issues that arise during short delivery cycles.

8.1. Quality Assurance

Our quality assurance team developed test plans based on the UE and tech specs available at the start of a six-week iteration. If things changed after those wiki-based documents were “complete,” which they almost always did, then parts of QA’s test plans would be invalid. This raised the broader issue of how to perform testing in a lean environment. Our QA team was building a test plan based on specs, then testing after code freeze. Waiting until the code is frozen to start testing increases the probability of a shipment delay as the time between the bug introduction and detection increases, which in turn increases the time to fix the bug.

This is more a compressed waterfall than a lean or agile model. Our interim solution was to inform QA of any changes in the UE or the tech spec. We came to an internal agreement that if anything changed in either the UE or the tech spec, the person(s) responsible for the change would notify QA in writing by e-mail. Our ultimate solution was to replace our testing process with integrated “gorilla testing,” highly integrated with our development team. Our QA tester followed the features of each developer and tested upon each feature’s completion. This integrated testing model served us well as we moved into a truly agile process.

8.2. Pre-Ship Review

The pre-ship review was intended to be a final review with the whole team, where we asked ourselves if we were really ready to ship. Because the pre-ship review came just before shipping, if we found a serious problem it was too late to do anything about it without delaying shipping, and that had its own repercussions. We needed to know if we were in trouble before the pre-ship review.
8.3. Shipping Delays

We occasionally delayed shipping to fix an issue or finish features. This irregularity caused more problems than it solved. For example, the team could not be fully utilized, as the next iteration planning activities had not yet occurred.

9. Getting Agile

As we moved toward agile, our team size shrank due to budget issues. The smaller team was composed of four developers, one user experience designer, one half-time QA engineer, and the PI part time. The project manager became a role within the team, but not a full-time position. A developer typically performed PM duties. With our automated build and QA processes in place, a half-time QA engineer was sufficient for internal testing.

The agile cycle is made up of four iterations with three-week durations, as shown in Figure 5. Four iterations is enough to add meaningful functionality for space operations, hence that constitutes a release. Note that the last iteration of a release focuses on bugs, usability, and testing, not feature additions. It is important to start the highest priority and most difficult features in the first two iterations, as it can take multiple iterations to complete features. Leaving the hardest features for last increases the possibility that they will not be completed by the end of the release. For mission control operations, the customer conducts a certification of the software for operational readiness after we deliver the release.

![Agile Release Into Operations](image)

**Figure 5. The agile release cycle**

Figure 6 shows a close-up of an iteration, and the relationship of iteration $n$ to $n+1$ and $n-1$. The choice of iteration length was driven by several factors. We adopted a strict shipment policy, with delays allowed only in extreme circumstances such as a product that does not work. Any features that are not complete simply don’t ship in that iteration [8]. Having a short iteration cycle supports this, because features that miss one shipment have another chance in only three weeks.

Note that the agile cycle has fewer discrete events than the lean model. The UE specs, rather than being due at the start of an iteration, have due dates set by the developers per feature, meaning the
developers tell UE when a spec for a particular feature needs to be ready so they can start coding. Stack rankings take place as needed, with continuous updates to priorities set in ongoing interactions between developers, UE, and the customer, with the UE and the development teams co-located and mostly remote customer participation.

We replaced testing after code freeze with a continuous testing model in which our internal QA tests each feature as it rolls out, the customer tests features as they roll out using the nightly build, and the customer verifies delivered features with each iteration.

Once and sometimes twice per iteration we hold a “hackathon,” where a half dozen or more people use the product together in our lab. Our product is tested against large sets of real customer data, and installed on hardware that matches – as closely as possible – our target deployment. This tests scalability, performance, and concurrency, and exposes problems that individual testing did not. In addition to the ad hoc scalability tests done during a hackathon, a long-duration stress test is conducted on a build of the software that passed the hackathon to ensure continuous operations under the expected load. Each delivery to mission operations must pass this test before it is shipped.

Our customer deliveries have three phases:

- A nightly build, used by our customer to test features as they roll out and provide immediate feedback. This process replaced the discrete event user testing from the lean model.

Figure 6. The agile iteration cycle
10. Conclusion

Our journey from traditional development to lean to agile was driven at each step by the need to address issues we were facing. By iteratively refining our process over two-years, we achieved our key objectives:

- A close relationship with our customer in which both they and we could had open continuous visibility into the state of the product.
- An development team making maximum use of limited resources, focused on the highest priority objectives, with minimal documentation
- The capability to measure and verify progress relative to a tactical plan and a strategic road map

It took multiple iterations of our process over two years to achieve relative stability. While we continue to make small adjustments, the fundamental process appears stable.

As a team, we found that agile increased the satisfaction of both the customers and the developers. With agile, we were able to delivery meaningful functionality to the customer and get immediate feedback; we were able to measure our progress daily. Over time, the customer-developer team became highly integrated.

In our experience the cornerstones of agile are

- The measure of progress is working code
- Rank issues, always focus on the highest priorities
- Demonstrations, not presentations
- Customer interaction, instead of extensive documentation of requirements
- Progress always visible – nightly builds, regular fully functional deliveries
- Always ship on time unless the product does not work. Features that are not ready go into a later delivery
- Process automation supports the team
- Verification of function begins with internal QA, final verification and issue closure comes with customer acceptance

Using agile methods, the team resources are focused on the highest priorities; designing and building the most important features of the product. Agile techniques such as designing and implementing only what will be used immediately help eliminate unnecessary work. Distractions to development, such as documentation and reporting, are minimized. We have a continuous feedback loop with our customer for testing and feature verification.

Software engineering processes, whether agile, lean, or traditional, must be tailored to fit the team culture, goals, and the context of work. There is no one right way to do it.

References


