

Daily Iterations: Approaching Code Freeze And Half The Team Is Not Agile

Curtis R Cooley
curtis@radsoft.com

Abstract

As an XP consultant, I'm asked to bring XP to diverse teams. But what happens when a team is split geographically and half the team prefers non-agile methods? Approaching "Code Frost" we decided to do daily iterations to kill bugs and keep the non-agile half of the team happy.

Introduction

Our team was split geographically. One team was working on an embedded application. The success of XP on an earlier, similar project, led management to decide to do XP with some members from the previous team. The embedded developers were convinced that XP would not work for hardware or firmware development. The GUI team used XP from the start. The firmware team chose to use a form of iterative waterfall.

After we passed what the embedded team called Feature Freeze, the development process changed. We found ourselves in a non-agile situation, where the embedded team was dictating the process. We adapted by choosing daily iterations and abandoning Release Planning.

The Methodologies

The GUI team used XP while the embedded team used a process that was based on iterative waterfall. The project manager was the manager of the embedded team, but success on an earlier, similar project convinced him to let us use XP to build the GUI. He kept his milestones, however, and held the XP team to these, thinking that would lower the risk on the project. The big milestones were: Feature Freeze, Code Frost, and Code Freeze. These were intended to reduce risk in a non-agile development methodology, but they were all completely driven by fear.

To get the two teams to work together, the embedded team adopted The Planning Game and Small Releases. They met with us daily for standup meetings and every other week for planning meetings. They built the server and released it to us whenever the project lead deemed it stable, so we had the latest version of the server to work with. They definitely were not doing continuous integration, since the project lead was the only one who could checkout the server, build it, and release it.

The embedded team felt that they could not use XP on

embedded code. I feel the resistance was based more on fear of change than any real technical limitations. There is no reason an embedded team cannot:

- pair program
- write code test first
- write code as a whole team
- refactor
- keep the design simple
- operate at a sustainable pace

The embedded team did have a coding standard and a system metaphor. Perhaps there are technical roadblocks to continuous integration and collective code ownership, but these can be overcome.

On a true embedded project, continuous integration is tough. You would have to burn new chips for each integration, but there you could use integration software. This project was unusual since the code that ran on the chip very rarely changed. It was a framework designed for this kind of application that allowed the embedded developers to write code at a level above the chip. There was an emulator that they could integrate against, and even when the team lead decided that the server was stable enough for a release, all he did was merge and compile code package it. No chips were burned in the process.

Each embedded team member relied on his narrow expertise for job security. This created a huge, but artificial, barrier to collective code ownership. They wouldn't pair program. They wouldn't leave their cubicles for standup meetings—they teleconferenced in, so the sharing of expertise that produces real code ownership was severely limited.

One GUI programmer worked at the site with the firmware team. We tried to get him to write tests and pair with us using VNC. It worked for a while, but as milestones approached he paired and tested less and less. We soon realized we had to isolate his work to one section of code and hope we didn't break his stuff when we checked in. This was a case study in the relative velocity of an XP team vs. a cowboy coder. At first, it looked like the cowboy was getting more done, but as time went on, he spent more and more time fixing his code and we spent more time introducing new features. It became very clear to the customer which approach produced features faster. Whenever we did any refactoring that touched his code, his code broke.

Before Feature Freeze

At first, the mix of agile and non-agile methodologies worked fine. We met daily for standup meetings and every other week for planning.

Having half the team estimate in real time and the other half in function points looked a little strange in XPlanner and the numbers made it look like the firmware team did more work. However, the customer could easily tell which team implemented features faster and wrote less risky code.

Members of the firmware team were assigned tasks instead of signing up, but that seemed to work for them. That was what they were used to, and the embedded team lead knew which new feature matched the expertise of each embedded team member.

Iteration Planning meetings started with the whole team in a teleconference using netmeeting and XPlanner. The customer would talk about the stories we felt we could fit into the iteration. We would revisit our velocity and each story estimate and build an iteration of stories. Each team would then split off and task out each story. The XP team signed up for tasks based on velocity, while the embedded team was assigned tasks. We would then get back together and confirm the plan for the iteration.

As we approached Feature Freeze, the project manager reminded us that once we reached this major milestone, we could no longer implement new features. We asked why, knowing the answer (I just wanted to hear him say it). He replied that implementing new features meant a risk of creating more bugs. We reminded him that we had 1500 tests that we ran all the time to reduce that risk, but he remained firm. He did, however, admit that in the eight or so months leading up to Feature Freeze, we had implemented features faster and with fewer bugs than any team he had seen at the company.

Feature Freeze was a phase in the project where implementing new features stopped. It was an arbitrary date on a calendar that was designed to provide the development team time to get the bug severity numbers in line with the magic requirements for release.

We worked around this by quickly implementing half of each feature the customer wanted, knowing that we could finish them once the missing pieces were reported as bugs.

Feature Freeze: The Fear Begins

Once we passed the Feature Freeze date, we could not implement new features, so the customer and project leads decided it was too much overhead to try and plan two-week iterations because the bug find rate was so

high. In reality, since we couldn't implement new features, most of the defect reports were enhancement requests. We called these Story Bugs.

The project manager and customer stopped caring about velocity, so Release Planning and Iteration Planning stopped. At first the XP team suffered a motivation set-back. We were used to having a plan: knowing exactly how much work we had to do; knowing how fast we were going; knowing if we were going to make it. The lack of planning took all this away from us. I brought this up with the customer and he felt knowing how fast we were going wasn't worth the overhead of Iteration Planning. That is when I noticed something strange about this company. At first, when we tried to introduce XP, they were afraid and remained true to their current processes. They are a very process-oriented company. But when fear of an approaching deadline builds, they are suddenly willing to cheat on their own process, thinking they can go faster. I explained to the team that the customer and project manager were no longer interested in the benefits of The Planning Game and that we would have to deal with it as best we could.

We evolved a daily iteration approach. Each day at the standup meeting, the customer and integration specialist would present the top priority bugs for the day. Developers picked from the list. We had a four-foot by two-foot by three-inch box that we took to meetings. We could pin story and task cards to it so we could always see the current state of the iteration. Each morning at the standup, the customer could present new bugs and rearrange the bugs on the box. Team member took the bugs off the box based on priority so the most important ones were getting the most attention. This effectively created a one-day iteration.

This short iteration worked well. The only real hitch was when the project manager started asking when we could Code Frost. We couldn't answer him since we were not tracking velocity. I had even proposed that we give each bug a cost of one so we could track the number of bugs per day or week we were able to fix. They were not interested in that information until the project manager wanted to know if we were going to hit certain target dates. We could not tell him because we did not know how fast we were going.

The biggest disadvantage of daily iterations is that you don't know how fast you are going, but there are other problems. One day is not enough time to deliver a cohesive chunk of functionality. Even in bug killing mode, we were producing functionality that required more than one day to complete. We could have easily written these chunks as stories and planned two-week iterations around them. We also lost the celebration at the end of the two-week Iteration Phase. It was informal, but essential to morale. We never recovered morale after we started the daily iterations. Daily iterations transferred the focus from the people to the quality of the software. XP is a people-oriented

process, but we continued to be blamed for all the bugs in the system even though the existing process we had to follow was largely responsible. The process caused us to rush to implement buggy features before Feature Freeze to produce all the features the customer wanted. The customer was aware of this and even brought it up in meetings when the project manager questioned the quality of the GUI. Without Feature Freeze, we could have continued using XP to ensure that the bug count was low. As further evidence of this, bugs reported against the features developed before Feature Freeze numbered below those of the bugs reported against the rushed features.

A benefit of the Daily Iterations is that the customer exhibited much more willingness to steer. We had told him during the two-week iterations that he could add or remove anything during an iteration. We would negotiate what he wanted and take that much out. During daily iterations we did not have to negotiate. Each day brought new work, so he could shuffle the tasks as he saw fit.

Code Freeze Approaches: Fear Takes Over

There were two remaining milestones for the GUI team: Code Frost and Code Freeze. We heard about these milestones for months beforehand. Code Frost is achieved when there are only a certain number of critical and severe bugs and the percentage of mediums and lows compared to the total number of bugs reaches some magic number. During Code Frost, only severe and critical bugs can be fixed, since any change to the system could introduce more bugs. This is basically a risk and fear based paradigm. Code Freeze means there are no critical bugs and the ratio of severe bugs to the rest is below some number. After Code Freeze no code is written unless QA finds either a critical bug or

enough severe bugs to cause the ratio to rise above the acceptable threshold. The project manager was not amused when I pointed out that you can get the ratio below the threshold by introducing more low level bugs.

As the release date approached, the project manager and team lead for the firmware team began to stress more and more. The more they stressed, the more they wanted us to stray from XP. It became very obvious that the firmware manager and project manager had never experienced the power of unit tests. Three times in nine months the GUI team introduced code that broke the system. The firmware manager remembered these events very well, of course, so the GUI team was held to the same standard as the firmware team. We could only fix critical and severe bugs, despite a suite of over 1500 unit tests.

Conclusion

Fixing XP when it's broken worked well, even when it wasn't XP that was broken. By remaining agile and adapting the methodology to meet the needs of the client, we were able to achieve the goals the project manager had set. When it became necessary to give up Iteration Planning, we simply reverted to daily iterations and planned each iteration in the standup. It cost us velocity calculations, so we were not able to tell the project manager when we would be done. This seemed to fit his fear-driven approach. He didn't want to know the project would be late until it was late anyway, preferring to push for overtime and other Herculean efforts he was accustomed to demanding from the estimates he usually gets.

However, we did reach Code Frost and Code Freeze, and the company is currently selling units running the software. Since the company is generating revenue from the software, I would deem the project a success.