Consciously Evolving an Agile Team

Aaron Ruhnow
iLevel, by Weyerhaeuser
AaronRuhnow@hotmail.com

Abstract

By now it is common knowledge that it's just as easy for a team to fail using agile practices as it is with waterfall or any other method for building software. So what is the secret to making agile work? How does a team become "self-directed" and "cross-functional?"
The key to our team's success was to have the core development team change its attitude toward software development. The team could then work to remove the barriers between individual developers and then between the developers and the larger project team.

At first the team struggled putting into practices the ideas from books and papers on agile development. We eventually made headway by focusing on making changes in a very purposeful and incremental fashion, which I call "Conscious Evolution."

1. Introduction

This paper describes the year-and-a-half "conscious evolution" of a first-time agile team. Shedding the waterfall-ish and command-and-control practices of the past, this team of developers found the path to make agile work for them, and in time, work for the larger project team. They used the scrum framework and adopted many of the core agile practices such as short iterations, daily standups and task boards, but this paper is focused on the methods that helped get the team committed to this framework, and transform the way they developed software.

To start, the team removed the silos and personas each created around themselves. Eventually, they worked on removing the walls that were erected via the established processes. "Constructive conflict" was used as a force to drive change and improvement.

These incremental and continuous improvements in process eventually allowed the team to find the best ways to fully integrate the project members outside of the immediate development team.

The team also had to make that mental switch from having their processes “preplanned” to creating and improving on them as a team. The team’s ability to make the change to an “agile mindset” was just as significant as any of the practices the team started with. Without this attitude change, the team would have experienced the same disappointments as it had in the past.

2. Background

iLevel, a subsidiary of Weyerhaeuser, is a manufacturer of engineered wood products, primarily for residential framing construction. For several years, iLevel has provided free-of-charge software to support engineers, architects and lumber dealers as they choose iLevel products to include in their building projects.

iLevel relies on software and technical support to create a competitive advantage over competitors who view these engineered wood products as a commodity.

2.1. The SFS Group

The company’s Structural Framework Software group (“SFS”) creates and supports the applications used by iLevel’s customers to interactively “size” iLevel products to be used in these building projects. The user interfaces have a CAD-like interface and include a sophisticated structural engineering design engine.

2.2. The Developers – Cowboys, Pessimists Privates and Meeting Masters

The development team has traditionally been a mix of professional programmers, certified engineers with an interest in programming, and the occasional “promoted-from-the-ranks” employee. Each member had their established personal approach to software development, but there were a few unique personas that could characterize most of the group:

- The Cowboy: A very talented programmer usually with an engineering background can
quickly add features to the application, even when not requested by the customer. Because of his/her domain knowledge, he/she is often forced to be the sole contact for supporting legacy applications. Their code is normally modular and often they avoid code quality practices.

- **The Pessimist**: Often a developer moved to the QA role, The Pessimist knows where to find weaknesses in the applications. The other developers fear The Pessimist looking at their work.

- **The Meeting Master**: Attempts to attain business and technical knowledge by holding long meetings often at the expense of other team member’s time.

- **The Private**: An experienced and capable developer, The Private is most comfortable taking orders from the command structure (management). The Private is not used to elevating their thinking beyond what the project manager has requested of them.

All these personas include many traits of good developers, but silos were often unconsciously erected between the varying personas. The most talented of the developers were not often in a position to share their knowledge with the rest of the team, instead opting to take the lead and heroically churn out the code single-handedly.

2.3. The Processes – the Wild West, RUP and Waterfall

For many years, SFS developed software with very little formal process or structure. The development team was isolated from the business owners, and very few people within SFS had direct contact with the end-users. Because of this, new product development was often difficult, and there was an adversarial relationship between the business and the development teams. Architecture and planning were rarely considered. The days of The Cowboy ruled the early “Wild West” of SFS.

More recently, attempts were made to improve the development practices. Standardized documents such as use cases and processes like RUP and Change Control Boards were introduced. A lead architect was hired and more formal structures were put in place. Things like “coding qualities” and “iterative development” were even whispered.

Some of these changes made a positive impact, but in the end, it was “The Cowboy Way” that still delivered the applications. The Wild West was still the primary lifestyle.

2.4. New Project, new Attitude

In 2005, a new team was formed to take one of iLevel’s popular legacy applications and re-write it from the ground up (some code in the application had been originally written in FORTRAN). Enough progress had been made understanding iterative development and coding qualities that project leadership determined this was the project to put more agile practices in place. Agile was still a new buzzword for the team, so the goal wasn’t necessarily to go “full agile.” However, as the new attitude and practices took hold, the team quickly evolved into a very agile form.

The project had a medium risk profile, which allowed the team to try some new things and “fly under the radar” (making the “Start Small” agile mantra possible).

3. Keys to “Conscious Evolution”

To become an agile team, it first took a change in attitude. Cowboys and Pessimists were encouraged to work with and learn from each other. The team was encouraged to continuously adapt their processes to the unavoidable changes that would come their way.

The team was told to expect the changes to be incremental as they moved forward. This was in contrast to a major shift in processes, as was often the case.
case with process changes of the past (such as the implementation of RUP). The “inspect and adapt” mantra helped us keep focus on these continuous and small improvements. The retrospective was the mechanism to do this.

At first the team was intimidated by the number of unfamiliar practices agile teams used. The team had to do their due diligence and find out which practices were the ones that made sense to do first. The practices outlined in this paper focus on the practices the team created that were most significant in their evolution.

3.1. Getting Incremental – Smaller Bites and Learning

The first challenge was to understand how to change to the incremental way of thinking, which started with the transition to shorter iterations. To accomplish this, iterations were gradually shortened from the 8-week iterations performed most recently, down to two weeks. By slowly evolving the team to shorter iterations, the team and leadership was able to adjust its practices to make two week iterations possible. Going directly from eight weeks to two would have been disastrous for the team, and unmanageable for leadership.

Another example of incremental changes was the requirements gathering techniques. The team was comfortable with creating use cases, and as a start, they generated use cases for the project. However, the team quickly learned that use cases were not easily transformed into distinct pieces of work. After a few iterations and some trial and error, the team was able to transform the use cases into what Mike Cohn calls “Epics and “Themes” [1]. The epics and themes could then be prioritized and broken down into individual stories for work in iterations. The concept of stories was initially difficult to grasp. By evolving from use cases to stories, the team could manage this change.

3.2. Eliminating Silos and Walls

Next, the team looked to break the silos between developers. The team generated a list of coding qualities that they believed were the most important and encouraged each other as they improved their ability to adhere to them. These qualities included an emphasis on keeping the code maintainable and adaptable, transparent to the entire team, and testable. This list was improved as the team learned what was truly effective, instead of just what was included in books and white papers.

Developers were also encouraged to be more collaborative in the coding process. One of the first practices the team added was pair programming, which helped improve the quality of work produced. The quality of all levels of tests and the requirements gathered were also improved by collaborative efforts.

The most important way the team broke down the walls between developers was for the entire team to be involved in all phases of development. This included discovering the requirements, writing and implementing the tests as well as writing and delivering the code. In a world where developers were not used to being testers, this was a revolutionary idea.

The team used the retrospective to step back and continue to find ways to eliminate silos and improve and reinforce the list of coding qualities and practices to remove the silos.

3.3. “Constructive Conflict”

Part of the team’s attitude change meant encouraging “constructive conflict.” The team would allow for the debate and questioning of all aspects of the application and processes used. From testing techniques to understanding and implementing requirements to finding better code qualities, team members were encouraged to question and challenge preconceived notions.

The key was to avoid being combative or allow The Meeting Master to dominate the discussions. This also required The Private to question their commanders.

With these discussions came a better understanding of the entire project and software development by the entire team. As the team improved its ability to have this constructive conflict, less and less time was spent in discussion and agreements came faster.

3.4. Shared Workspace

The development team gradually moved to a shared work area to eliminate the silos and encourage collaboration and constructive conflict. Developers were not initially comfortable with tearing down cube walls, so this change started small by having the team work together in conference rooms for a day or two at a time.

The team spent more and more time working a group until very little time was spent in their own cubicles. After the development team gained a comfort level with this shared space, other project members were encouraged to spend time with in this shared workspace.
3.5. Amplified learning

The new emphasis on collaboration techniques helped amplify the learning for the entire group. As one developer learned a new tool or technique, it was immediately taken to the team. As The Cowboys and Privates learned a useful new technology, everyone would learn from them. When The Pessimist found a new way to break the code, they would teach the rest of the group how to do it themselves. Pair programming allowed this kind of learning to happen daily.

4. Evolution: Moving Quality from the End to the Front

Undoubtedly the biggest challenge in changing the mindset of the team (and management) was to get a true emphasis on quality. When the team started out, quality was the responsibility of a separate QA team, who would test as best they could near the end of projects.

For this project, the entire team was to be responsible for quality. This meant the team had to transform dedicated QA members into day-to-day developers. The former Pessimists now took on the role of “quality champions” who would encourage the creation of quality unit, integration and user acceptance tests by everyone. These champions would also take the lead when the team tried new testing techniques and tools such as FitNesse. They ensured the team made progress each iteration improving the quality infrastructure.

The other challenge was to encourage management to get quality on equal footing with schedule. Fortunately, the project manager and product owner were willing to help the team evolve their practices to meet a team goal to “build quality in.” As the “test-first” mentality became a larger part of their day-to-day tasks, quality was moved from the end to the front of the development process.

Work on bugs and refactoring needs added to the backlog and given equal footing with work on stories. With higher confidence in the quality of the application, re-work and bugs took up less of the team’s time.

5. Evolution: Adversaries to Allies – Enlarging the Conscience

Only after the development team was able to operate in a more cross-functional manner could efforts be focused toward improving their relationships with those outside the immediate developers. This included the product owner and what SFS calls “shared resources” - the architect, QA lead and install developer.

The team found that the more these other members were “embedded” within the development team, the greater their contributions and understanding became. This also offered the team an amplified learning of these people’s considerable skills. These embedded specialists greatly increased the collective team conscience.

The team used each iteration kickoff to allow the shared resources to have a say in what was worked on. The lead architect could request on an important refactoring. The QA lead could request the team improve tests in a certain area. This caused some “constructive conflict” between product management and the shared resources, but everyone made their arguments in terms of how it affected the quality of the final application.

5.1. Project Management

The team still had a traditional project manager. As the team worked to become more agile, the PJM’s role in the day-to-day activities decreased. Fortunately, the PJM was willing to drop the traditional command-and-control style and work with the team to develop and refine the team’s processes.

The team still leveraged his considerable skill at project planning that the developers (and even the ScrumMaster) did not have. The PJM worked with the team to establish the best way to effectively manage and track progress of the project.

Using standard techniques such as tracking velocity and metrics such as percentages of automated test coverage and cyclomatic complexity, the entire team was able to gain a higher level of confidence in the progress and quality of the application.

5.2. Product Owner

The product owner, also a newcomer to agile, was willing to allow the team to try the new practices. Traditionally, product owners were on the outside of the team looking in. However, this team was able to evolve their practices to allow the product owner to be a part of the team.

The processes that were most effective in strengthening the bond with the product owner were:

1. Working with him to evolve the initial list of use cases into stories. The product owner then “owned” the story list – he could add, remove and change the list and reprioritize it. This gave
the product manager more control over the team’s workload than they ever had in the past.

2. Writing the User Acceptance tests as a group with the product owner. The team found that writing these tests up front was a much more effective form of requirements discovery than any requirements document.

3. Allowing the product owner to be involved with the retrospectives and give the final call on the completeness of stories. The development team’s success now directly relied on meeting the product owner’s expectations. By involving him early in the iterations, the team was able to understand and manage those expectations.

4. Providing daily updates on the progress of the stories. The team reviewed completed portions of the story with the product owner as soon as they were done. The more the team did this, the more comfortable the product owner felt with their progress.

5.3. Architect

SFS’s lead architect was one of the early champions of agile practices, so getting him to adjust his day-to-day practices for the team was not difficult. Traditionally, the architect was brought in to help create the high-level structure of the application and review subsequent redesigns of the system in a “Big Design up Front” (BDUF) fashion.

As the team began to develop more iteratively, the BDUF process was used less and less. Changing to the practice of a more emergent paradigm meant the architect needed to be “in the trenches” with the team. The architect would sit in on pair programming sessions to get the look and feel of the application and offer suggestions to help evolve the architecture.

The team still needed his experience and insight, but it had to come out as part of the day-to-day development, instead of large modeling sessions before coding, or reviews after coding.

5.4. QA

Another big advocate in the early days of the team was the SFS QA lead. This was due to the team’s desire to make all team members responsible for quality. Like the architect, the QA lead was brought in as much as possible when tests were written. Because quality was traditionally a minor part of the daily practices of the SFS developer, the QA Lead was relied on to help the team understand and implement good testing practices and learn testing tools.

5.5. Install Developer

Another important key to the team’s success was to achieve Continuous Integration (CI). SFS traditionally has a designated developer to make the install packages for all SFS applications. Fortunately, the current install developer was familiar with several of the popular tools to help the team create automated builds and reporting along with the install packages.

The automated builds were run daily, and eventually hourly. Between the CI and automated tests, very few missteps by the developers would linger for more than a few hours. By building the install package from the earliest iterations, the team was able to avoid the thrashing that normally occurred just before beta testing when the first installs were created for the application.

6. Evolution: Feedback – From 6 Months to two weeks

The team was used to waiting until the Alpha test cycle at the end of projects before anyone outside of SFS would see the applications. In this project, the early feedback the team received from those outside of SFS proved to be valuable in delivering what the customers wanted. Eventually the team learned to actively seek out all feedback loops that could be shortened to improve the quality and value of what the team delivered.

6.1. Shorten the Iterations

The first step was to shorten the iteration length. After a few iterations of learning how to create working software every two weeks, the team was able to solicit feedback from iLevel employees familiar with the application.

6.2. Getting Test-Driven

As the team strived to improve its ability to “test first,” the team learned that the code had to be inherently testable, even in early stages. There were portions of early code the team thought was really slick, but proved to be very un-testable. As the team wrote more of their code with tests up front, the more testable (and more maintainable) the code became. Writing tests first also helped uncover emerging (or “hidden”) requirements before one line of code had been written.

Developers also had to be more comfortable with understanding that coding itself was evolutionary, and the sooner the code could be put in front of the product
owner, the sooner it could be reviewed and analyzed for even more hidden requirements and opportunities for improvement. Most of the developers had to become secure with changing and improving their code based on feedback from the product owner and other developers.

6.3. Beyond Unit Tests - Blur the Requirements-Test-Code Cycle

As the team matured, they found many ways to keep the entire “loop” of development as tight as possible. The structure to do this was not in place when the project began, but was consciously evolved, using the best of the agile processes as a guide. Eventually, the team was able to efficiently write tests, find requirements, immediately code them in, run the tests to verify the expected results, and review with the product owner. Since these cycles were now so small, the team was able to quickly make any changes the product owner and/or development team thought necessary.

6.4. Tests as the Requirements

Once tests and requirements occurred almost simultaneously, the team was eventually able to get rid of something dreaded by many developers: writing requirements documents.

Mike Cohn recommended the use of FitNesse as the repository of tests and requirements [2]. This was indeed a bold and foreign concept to the team at first. However, after reviewing the team’s current requirements documents, the team found that they were using these documents less and less as the product owner became an integrated member of the team.

It took several months to get it in a good form, but eventually the use cases and detailed requirements documents were completely dropped and all new requirements were kept in FitNesse as tests.

This may have been the most radical and advantageous break the team made from the traditional SFS practices. It was, however, a change that could only be done after the team reached an appropriate level of maturity to allow this. There were no shortcuts in getting to this point.

Now, when a new story introduces a change in requirements, the altered tests document the changed requirement. And with that, the bane of the existence of many developers was destroyed!

7. Conclusions

The team found the key to moving to agile was one of very measured and conscious changes. This was enabled by adopting an agile attitude toward development.

In the early phases, it’s also important to look for the “easy wins” and “biggest bang for the buck” changes to encourage the team to get comfortable with the incremental, but constant change.

Unlike many of the other processes implemented by SFS in the past, this team is not looking to make it to a “final” or “ultimate” state. Some practices that worked very effectively for the team early on were no longer meeting their needs and were changed. Things the team does today will likely be changed in the near future as the team continues to evolve and mature.

As the team began to accept the fact there is no “Agile Nirvana,” they became more comfortable constantly questioning and changing their practices as needed. The “it’s not a destination, but a journey” cliché works well in this case. It sounds simple, but several years of going through efforts to put in place a prescribed process for software development, this highly collaborative style, along with a few simple attitude changes, made a real difference to this development team.

8. References
