Agile Development: Implementation, Best Practices and Lessons Learned

Christopher S. Ritchie
critchie@vt.edu
Virginia Tech

Abstract
As software languages have evolved from procedural-based applications to object-oriented, so too have the software development processes. This paper will give a brief explanation of traditional software development methods such as the waterfall and spiral models and will delve deeper into current trends surrounding agile development. A brief overview of scrum and extreme programming will be presented, but the focus will largely be on exploring in depth the more common, broader agile development processes. Several case studies on agile development will be presented that will highlight the growing pains and lessons learned associated with agile development. The case studies presented will help to illustrate how software development is an ever-evolving process that private and public sector industries employ to create products for businesses and consumers.

Overview

The development of software has been driven by the introduction of computers to the corporate world. Since their original deployment as large and cumbersome mainframes, a need for software to drive corporate calculating machines has been an ongoing and ever-evolving process. Computing languages such as FORTRAN and COBOL were originally developed for the corporate world as a means to manipulate and derive value from stored information to support business decisions. As computers evolved, so did the languages that support them, and in turn, the processes behind software development.

The waterfall model, which was introduced as early as 1956, gained wide acceptance as a standard model in the 1970s after the publishing of an article by Winston Royce. (Royce, 1970) The waterfall model, described by some as a traditional model, involves defined steps or phases such as requirements gathering, analysis, design, implementation, verification, and maintenance. The waterfall model is a rigid and linear approach to software development with each stage starting, ending, and transitioning to the next phase, as the image below depicts. The waterfall model is still frequently used today, especially with large-scale projects where the requirements are well-understood.

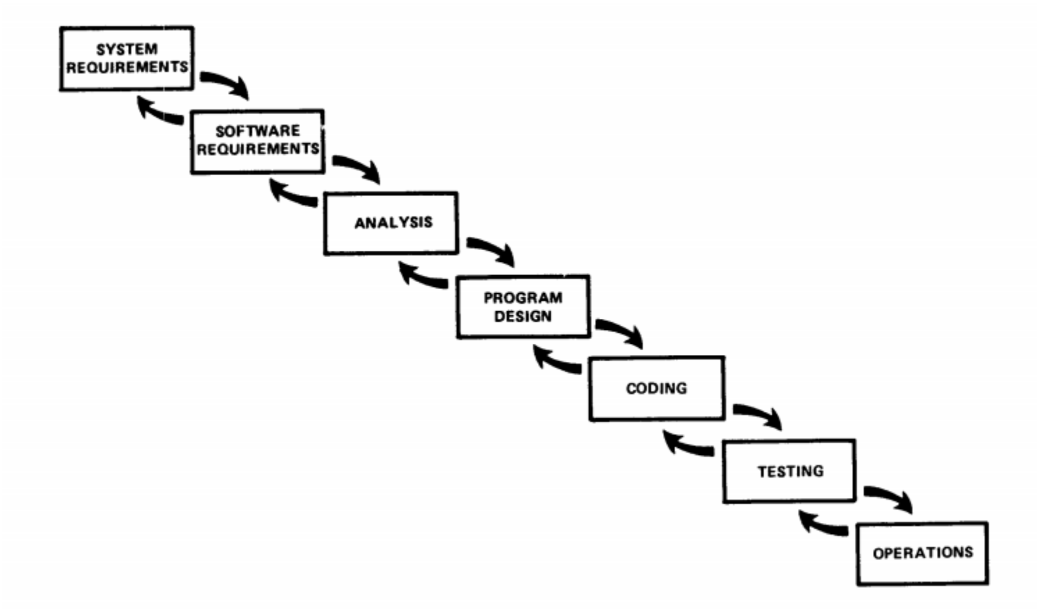

Figure 1: "The Waterfall Model," image credit to Winston Royce (Royce, 1970)

The spiral software development model was later created and documented in 1986 by Barry Boehm. The spiral model, in contrast to waterfall, outlines an iterative development process. The spiral model has four key phases, beginning with determining objectives, identifying and resolving risks, development and testing, and planning for the next iteration. The spiral model incorporates the idea of iterative programming and prototyping on top of the more controlled and defined aspects of the waterfall model. One of its key strengths is its ability to adjust to and incorporate changes to the requirements as they become better-understood. Because of its greater degree of adaptability, the spiral model is well-suited to projects in which requirements gathering is ongoing or nascent. Nevertheless, the spiral model's flexibility does not preclude it from large-scale software development efforts normally well-suited to the waterfall model: like its older counterpart, the spiral model is frequently used in large and complex development projects. A middle-of-the-road approach, the spiral model builds on the waterfall model by incorporating some aspects of agile development, particularly with regards to its iterative nature.

Both the waterfall and spiral model were pervasive up until the turn of the century where an even more adaptable, nimble development model was sought to manage and handle the often changing requirements of software. Agile development gained traction in 2001 with the publishing of the *Agile Manifesto*. The agile model is a lightweight development model geared towards rapid development. The agile manifesto states:

> *"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*
>
> > *Individuals and interactions over processes and tools*
> > *Working software over comprehensive documentation*
> > *Customer collaboration over contract negotiation*
> > *Responding to change over following a plan*
>
> *That is, while there is value in the items on the right, we value the items on the left more."*
> (Beck, 2001)

Two common implementations of agile development are scrum and extreme programming, each of which has their own specific goals and objectives.

Agile Development

The above-mentioned agile manifesto outlined a culture which rapidly took hold in the software development community and has since gained wide acceptance as a preferred means for software development, when it is appropriate.

Characteristics

Granville Miller outlines nine characteristics of agile software processes in their paper, *The Characteristics of Agile Software Processes*. The first of such characteristics is the need for modularity which allows processes to be broken down into sub components called activities. These activities can then be tackled by programmers. (Miller)

A second characteristic, already touched upon, is the iterative nature of development. As Miller points out, the iterative process acknowledges the fact that "we often get things wrong before we get them right." (Miller) The focus of each iteration, a short cycle often one to three weeks in duration, is a product to be turned over, though it is almost certainly not 100% complete or correct. The next iteration then begins by tackling the shortcomings of the prior iteration. (Miller)

Another characteristic touching on iteration time management and scope is the time-bound characteristic. Time-bound explains the understanding that only certain activities will be conquered in a specific iteration. The essence of this characteristic is that programmers must focus on a component or small set of activities for a specific iteration and hopefully achieve a level of completion regarding those activities in a given iteration. A goal is set out for each iteration with the hope of completion by iteration's end. It should be understood that functionality and quality may be sacrificed in order to achieve a workable solution. What may be lacking in this iteration, such as functionality or quality, will be tackled in a subsequent iteration. (Miller)

Parsimony describes the need to divide components into the minimal number of activities required to reach a goal while still mitigating associated risks. This aids the development team by developing realistic deadlines which allow developers to lead a normal life, preventing long work schedules and burn outs. Parsimony attempts to prevent placing the onus on software developers to code under impossible deadlines or circumstances. (Miller)

The ability to adapt to unknown or unforeseen changes is another important characteristic of agile development. Its adaptability stems directly from its iterative nature, meaning that unforeseen challenges may be systematically tackled as they unfold in subsequent iterations. Some activities may be discarded as the product becomes better understood. (Miller)

This incremental characteristic allows agile developers to break down larger systems or projects into manageable tasks, which can then be independently tested and later integrated back into the software project as a whole. This allows for more robust and adaptive programming, while also improving efficiency, a la Ford's conveyor belt model. By allowing specific individuals or teams to focus on a singular task rather than an overwhelming project, they can more rapidly acquire subject matter expertise as they become adept in their tasks. Furthermore, agile development managers can mitigate risk in the event that one component fails testing by isolating each component from the overall project. (Miller)

Convergence describes agile's focus on tackling risks at each iteration, thereby removing the necessity that programmers conduct all risk management activities at the end. Risks are proactively addressed and development in turn is as rapid as possible. Miller states that this helps to ensure success at the fastest pace possible. (Miller)

Another characteristic describes the people-oriented approach agile development takes. Since people are favored over processes, programmers feel more empowered, which in turn raises their productivity, performance, and quality. Programmers evolve and adapt through an organic nature. (Miller) As case studies will point out, this is a very important and often misunderstood variable when adapting agile to an organization or project. Corporate culture is also one of the hardest hurdles to overcome. (Miller)

The final characteristic is agile's focus on collaboration. Communication is a vital part of the agile development process and is addressed through standing meetings and a cooperative culture which aids in the creation of a competent, cohesive, development team. The team discusses any problems encountered so that they can be better understood and addressed. Communication also helps developers understand the overall projects and its pieces. (Miller)

Miller's eighth and ninth characteristics embody one of the core ideals of agile development: people and the need for communication among programmers. Much of the success and failure of agile development within companies depends on support for the agile effort by developers and senior management. Agile development cannot simply be implemented; it is often a cultural shift within an organization. (Miller)

Growing Pains

In an IEEE article by Kieran Conboy and Sharon Coyle entitled *People over Process: Key Challenges in Agile Development*, the authors describe the challenges 17 companies reported regarding the management of people as their companies transitioned to agile development. The authors start by introducing a table which outlines the differences in key methods between agile and traditional development, depicted below. (Conboy, Coyle, Wang, & Pikkarainen, 2011)

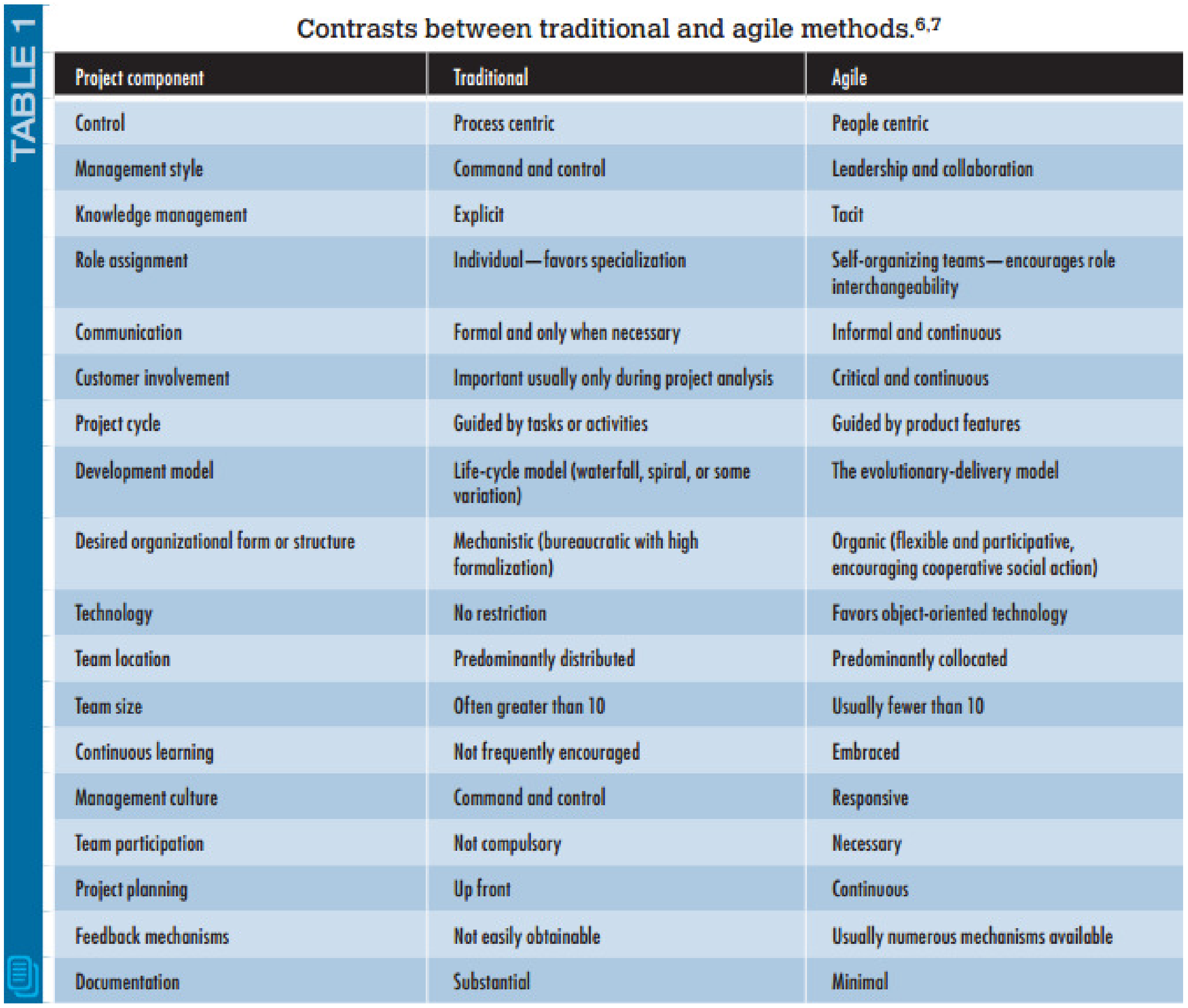

TABLE 1

Contrasts between traditional and agile methods.[6,7]

Project component	Traditional	Agile
Control	Process centric	People centric
Management style	Command and control	Leadership and collaboration
Knowledge management	Explicit	Tacit
Role assignment	Individual—favors specialization	Self-organizing teams—encourages role interchangeability
Communication	Formal and only when necessary	Informal and continuous
Customer involvement	Important usually only during project analysis	Critical and continuous
Project cycle	Guided by tasks or activities	Guided by product features
Development model	Life-cycle model (waterfall, spiral, or some variation)	The evolutionary-delivery model
Desired organizational form or structure	Mechanistic (bureaucratic with high formalization)	Organic (flexible and participative, encouraging cooperative social action)
Technology	No restriction	Favors object-oriented technology
Team location	Predominantly distributed	Predominantly collocated
Team size	Often greater than 10	Usually fewer than 10
Continuous learning	Not frequently encouraged	Embraced
Management culture	Command and control	Responsive
Team participation	Not compulsory	Necessary
Project planning	Up front	Continuous
Feedback mechanisms	Not easily obtainable	Usually numerous mechanisms available
Documentation	Substantial	Minimal

Figure 2: "Characteristics of Agile vs. Traditional Development," image credit to Conboy and Coyle (Conboy, Coyle, Wang, & Pikkarainen, 2011)

The authors surveyed 17 companies and found nine challenges related to people, and offered suggestions on how to best combat these issues.

Developer Fear of Skill-Deficient Exposure

In the 17 companies surveyed by the authors, all companies to one degree or another suffered from developers' fears of being exposed as technically deficient. Developers' concerns went beyond typical psychological resistance to change; largely because they simply weren't sure whether they'd "make the cut," so to speak. In order for agile development to work, story boards are often used to describe customer interactions, standup meetings are needed to bring to light current challenges, and onsite customers for constant feedback are a norm. These very strengths of agile development pose problems for many introverted developers. Standup meetings, for example, often made developers nervous. The amount of automated testing also prevents developers from hiding poorly developed code. Again, many of the strengths of agile development prove difficult for developers to cope with, at least initially.

Some means that were employed by companies to combat these deficiencies included developers filling out short forms documenting their fears and concerns on a biweekly basis.

Another company made it voluntary for junior developers to discuss their issues in the standup meetings. Several companies also provided separate, lengthier meetings for their junior staff where mentors were available for coaching purposes.

Broader Skill Sets for Developers

In order for agile development to work, developers need to become a "jack of all trades." Developers are asked to be users, testers, architects, and coders. Numerous responses to the author's surveys explained that this was a difficult task for many. (Conboy, Coyle, Wang, & Pikkarainen, 2011) Traditionally, developers worked in niche areas and received training based on their area of responsibility. Agile development asks for developers to do more and adapt more rapidly. A byproduct of this adaptive ability is the need for developers to become skillful in all development areas. As mentioned, training becomes more difficult when you are trying to cultivate a cross discipline team who not only needs numerous technical skills, but also communication skills.

Increased Social Interaction

The authors pointed out that the vast majority of respondents expressed issues surrounding individuals who were extremely technically proficient but had inherently weak communication skills. Other problems that came to light included the agile requirement for constant customer interactions. Several interviewed companies expressed issues with developers and client interaction, specifically developers revealing politically sensitive and sometimes corporate confidential information. (Conboy, Coyle, Wang, & Pikkarainen, 2011) In short, developers often did not know when to keep their mouths shut. An interesting caveat was that strong communication skills also proved to be problematic. Companies often were left wondering if an interviewed prospective employee truly had the technical skills they claimed.

These communication difficulties were combated by supplying further communications training and also recording past presentations and meetings and playing them back so that employees could track their progress. Although agile development keeps system documentation to a minimum, this sort of recording was a means used to better communicate certain information.

Lack of Business Knowledge

Due to the constant flow of communication necessary between developers and customers, in many cases it became rapidly apparent that developers lacked a basic understanding of the business for which they were designing an application. This lack of basic business domain knowledge often translated to a loss of confidence by the customer. One way to combat this deficiency was to hire recent graduates with both IT and business backgrounds. Basic business training was also used to help train staff; while this helped, client specific and institutional knowledge were still missing. In two cases, the clients provided specific business training. Hiring domain experts was another means of combating business knowledge issues.

Understanding Agile Values and Principles

Many companies moved to agile development after completing some sort of formal agile education. This often provided a good foundation for beginning the move towards agile development. This initial training, however, was often the sole training requirement, and in many cases it proved not enough. Several companies had issues achieving some of agile's goals due to a lack of understanding and proper alignment. It was recommended that developers receive continuous agile training in an effort to keep agile ideals and principals at the forefront. The constant reinforcement and education helped change corporate culture as well.

Lack of Motivation

A few companies experienced setbacks regarding agile's adoption in the organization. Companies seeking to implement agile development in a top-down manner, with upper management spearheading the effort, typically had the least success. In fact, some of the most successful companies at executing agile development procedures began with a grassroots movement towards a more agile paradigm. This "management first" phenomenon was combated by offering success stories, which helped gain developer buy-in and support.

Developed Decision-Making

Agile's free-flowing nature often contributed to problems in which developers picked tasks for which they were not skilled. Another cited problem revolved around managers coping with the loss of power in the traditional sense. A voting technique was used to help combat this issue which allowed all the developers to weigh in on key issues and decision making.

Implementing Agile-Compliant Performance Evaluation

All companies responded to the interview with issues around performance measurement. They cite the relative ease of measuring the performance of a specific individual, but express greater difficulty in measuring the overall performance of a team of coders. (Conboy, Coyle, Wang, & Pikkarainen, 2011) Many companies reviewed junior developers by looking at their technical skills; social skills were skipped entirely. Other companies simply used their traditional performance measurements which didn't articulate the work done by agile teams. To combat their performance measurement deficiencies, some companies introduced 360-degree reviews where peers could review one another. Another team-oriented performance incentive is team-awarded bonuses, something every employee – introverted or not -- worked hard to attain.

Recruiting Challenges

The recent creation and adoption of agile principles makes it hard for companies to recruit individuals with the skills required for agile development. Its sheer newness has meant that it receives limited attention in traditional 4-year college curricula. Colleges and universities too often skim over agile methods and simply delve deep into business or technical content. Companies battled these issues by developing their own agile recruiting policies which had applicants refactor code, create user stories, and participate in standup meetings. These practices quickly brought a developer's social or technical skill deficiencies to light.

As Conboy and Cole point out, companies face a number of challenges adopting agile development methods, many specifically around managing the individual. (Conboy, Coyle, Wang, & Pikkarainen, 2011) The authors point out that they were only able to interview managers, and not developers. They speculated that developers might have a different spin on many of the issues addressed in the article. Some of the overarching problems around managing the individual and implementing agile involve a cultural shift. Agile principles are fairly new and only recently have begun to be taught in university programs. The authors suggest solid recruitment policies as a means to begin a company's cultural shift to agile. Proper training and success stories are a means to motivate current developers.

Presently, agile development is commonly touted as best-suited to small teams of developers; current research focuses on mapping the manifesto's principles to large-scale software systems and mission critical systems. Agile development is currently not a preferred design model for mission essential systems and those involving life safety, largely due to their criticality and the absolute requirement that the software may never fail. Though agile development may not be right for every software system or software development team, lessons can be taken away no matter what the purpose of the software system. Agile development has gone as far as to describe a technique for not only software development but also flexible product development, and project management. Specific agile principles can often be selected and molded for certain situations or a corporate culture trying to adopt them.

Scrum

Scrum is one such framework that employs distinctively agile principles. Scrum is similar to its counterpart extreme programming (XP), but is more formalized and normally works on slightly larger sprints (iteration cycles). (Hayata & Han, 2011) Scrum's framework consists of a product owner, who creates a product backlog which describes activities or a wish list for future development. Sprint planning outlines the process a scrum team will take to tackle one or more of these activities. The team will then focus on completing the activities outlined in the current sprint over the next few weeks. Daily meetings are scheduled where the day's issues and obstacles can be addressed and planned. A team member is normally nominated or assigned the role of scrum master. It is this person's job to keep the team focused on the tasks at hand and to handle any day-to-day management of the team. The end of each sprint provides a potentially (though not often) complete product with the possibility of being final. The product of the sprint is often shown to key stakeholders, who provide feedback for the next iteration. The process starts again by either refining the product of the last sprint or by pulling new items from the product backlog. In most cases a sprint review takes place summarizing the sprint's activities and any issues uncovered.

Extreme Programming (XP)

Extreme programming is another common type of agile development. Extreme programming's goals are to quickly develop quality software with the ability to rapidly respond to changing customer requirements. The concept of extreme programming attempts to take industry best practices to an extreme level – hence the name. It was originally proposed by Kent Beck (also the author of the agile manifesto, mentioned above) when developing for Chrysler. (Copeland, 2001) Extreme programming is an agile implementation often best suited for highly disciplined developers. Extreme programming has four key activities: coding, testing, listening, and designing.

With regards to coding, the ideas surrounding XP are that code is the final product and the fruit of developer labor. Code is often used to explain complex programming initiatives and shown to fellow developers to explain their unique approach to the problem.

Testing is done in two phases. Unit testing is conducted after a specific function has been developed. As much of the testing as possible is automated; the goal is to run through a battery of test scripts to test all aspects of the code to unearth defects. Acceptance testing is later conducted to ensure the code addresses the requirements correctly. Acceptance testing also verifies to the customers and developers that the problem is correctly understood.

Listening to customers is the job of application developers. It is necessary for developers to listen and demonstrate to the customer that their problem is well understood. Listening is also required due to the iterative nature and extreme programming and its ability to rapidly adapt to changing business needs. Collocating customers with developers is a common XP practice.

The designing activity encompasses the tasks related to the overall system design. In most cases a good design of even a complex system will yield little coupling between components. This insures that modification of one function or component minimizes impact to other components between iterations. Low coupling and high cohesion are a common design goal.

Case Study: Intel

Bill Greene from Intel Corporation details their transition to agile practices as a technical team lead in the Itanium processor development department. His experiences and lessons learned are documented in *Agile Methods Applied to Embedded Firmware Development.* (Greene, 2004) Greene starts by talking about the importance of processor firmware as a method of correcting hardware defects in processors that role off the assembly line. He also acknowledges that traditionally agile practices inherently adapt better to object-oriented languages, whereas his department deals almost entirely in Itanium assembly, a non-object-oriented language. Despite this, Greene still felt that agile had enough to offer that it was a worthwhile venture; he was specifically drawn to agile's high emphasis on people over processes.

Greene would strive to bring agile processes to a department in charge of coding firmware for what was then a high visibility project and revolutionary processor, the Intel Itanium. Greene's department specifically worked on the PAL or Processor Abstraction Layer firmware, which brought features such as power management, error handling, and higher-level feature sets. It should also be noted that most of the developers are actually electrical engineers who happen to have picked up coding as a secondary discipline. Greene hoped that agile would address five specific issues they were grappling with.

Greene's team had difficulty with quarterly planning; he speaks of instances where not a week after planning, the schedule had to be abandoned. This proved frustrating and a little demoralizing for the team. A second issue surrounds team member's knowledge being too domain-specific. There was no cross-training and he hoped to change this with agile. Third, testing was poorly implemented. Greene's team currently did not have a regression testing suite and often needed to elicit the help from other engineering teams when it came to debugging and troubleshooting. A fourth issue surrounded code and its maintainability. Greene cites the need for constant workarounds (often to circumvent hardware deficiencies) and developers over-engineering code causing it to become less manageable later on. Lastly, there was no clear coding style or techniques employed by his department. (Greene, 2004)

Greene then explains his search for training and more information on agile principles in practice. Once he found the right resources, he asked the members of his team to read through a book and be prepared to discuss and plan their shift to agile. The developers did so and responded back with a few concerns, one of which was performance evaluation. The idea of pair programming contradicted their current performance measurement system, which was merit-based. Developers were worried that their contribution may be marginalized when working in pairs. Pair programming was also a major source of contention; many developers felt it was a waste of time. The constant collaboration required of agile also caused some developers to fret as they feared losing individual time to work, think, and create. Lastly, the increased amount of testing and its place earlier in the development cycle was a point of contention.

The Itanium firmware team, despite some obvious obstacles, decided to implement agile principles. They employed a combination of scrum project management techniques with some of the extreme programming development ideas, a mix that was tailored to their team. Of the scrum techniques, they decided in particular to implement sprints and their associated planning meetings, daily scrums, and sprint reviews. The team decided that sprints would be thirty days in duration, which allowed the team to commit to well-understood tasks, but still allowed for a level of flexibility. Sprint planning meetings were used to encourage task completion. The black and white idea that a task is either complete or incomplete was something Greene's team direly needed. He spoke of instances where a feature in firmware was almost complete for months. The daily scrum raised everyone's awareness to what others on the team were doing. Greene does explain that complaints arose out of the length of time the daily scrum took when developers gave daily updates. This was not abandoned, but was trimmed for brevity. Finally, sprint reviews were used and were very valuable for Greene's team in the beginning, but as they completed more and more sprints, the sprint reviews grew shorter. (Greene, 2004)

The firmware team also employed several extreme programming techniques. Greene's team implemented the simple design principle by creating more generic code and halting the common practice of coding overly efficient, inflexible code. Greene explains that often developers would code very efficient routines but their efficiency often had a high cost in lack of flexibility.

The idea of unit testing was a great success for the firmware team. Greene explains the need for developing and writing tests prior to coding gave rise to a shift in the team mindset. Greene speaks about how developing tests often helped developers better understand the code they were about to write. In Greene's words, "It forced us to think about what the code should be doing before coding and made the coding process itself flow easily once the tests were in place." (Greene, 2004)

The introduction of agile principles to Greene's team led to reinvigorated code refactoring. Code refactoring is the process of modifying code without changing its function, with the aim of improving its performance or readability. Code refactoring had been in place but was often an afterthought. The introduction of agile resurfaced code refactoring and Greene's team looked for opportunities to refactor which led to higher overall code quality. Refactoring even took place on legacy code, where the team witnessed an increase in performance.

Pair programming was introduced but required some modification to suit assembly language programming. Greene states that pair programming worked well in the early coding stages, but as the code became more complex and domain specific, a single developer needed to take over. Greene also spoke about cross-training not being quite as successful as he had hoped. He believes that the nature of processor design is too complex for any one individual to fully understand. Cross-training did work to the extent that developers became more knowledgeable about overall processor mechanics, but not to the degree that agile principles suggest. Lastly, Greene spoke about the change to employee evaluation criteria to include teamwork. This new criteria alleviated the concerns mentioned by developers about ownership and merit-based evaluations. Greene also found it interesting that one of the biggest skeptics on his team during agile's introduction now does the most pair programming. (Greene, 2004)

While Intel's customers in the traditional sense are not onsite, the hardware design team acted in a very similar fashion. The hardware design team often looked to the PAL firmware team to make changes in firmware to support the latest processor design. Greene commented on the constant interaction from the hardware team with his development teams as a major plus. He thought that there would have been more formal demands if it weren't for the hardware team being collocated so close by. (Greene, 2004)

Agile's principle of a sustainable pace was difficult for the PAL firmware team to implement. Greene cites the team's critical location on the processors' production line as one reason that hard deadlines were simply a must. His team lacked the flexibility called for by agile's sustainable pace notion. They are still striving to implement it, however.

Over the course of the PAL firmware team's move to agile, Greene states that many external managers and team leads did not notice any negative effects. External members actually commented on the PAL team's increased communication and approachability. (Greene, 2004) Agile methods on testing were one of the most valuable and brought about one of the biggest changes, a higher quality of code.

Greene finishes by stating that Intel's Itanium firmware team for the most part found the transition to agile a positive one. Greene also states that although it was positive, it was a slow process with a bit of learning and constant improvement along the way. Greene states "in particular, there is always room for improvement in the people aspect of Agile." (Greene, 2004)

Case Study: Landmark Graphics

Marjorie Farmer in an article entitled *DecisionSpace Infrastructure: Agile Development in a Large, Distributed Team* discusses Landmark Graphics' transition from a traditional waterfall model to a more agile, iterative one. Landmark Graphics provides commercial software solutions for the oil industry, with integration being one of their key selling points. While they originally developed a wildly successful software suite in the 1990s, Landmark Graphics kept up with ingenuity mostly through acquisition, acquiring smaller companies and integrating their solutions within Landmark Graphic's current software suite. (Farmer, 2004) While their key software suite was their major revenue earner, the corporate brass recognized that it would not last forever. As their current suite began to mature and sunset, the board looked for a new suite to take its place.

Farmer goes on to explain Landmark Graphics' attempt with the development of "Reno." This product may have failed in the marketplace, but several things were learned from the experience. The board then looked at a new project to which it could apply its lessons-learned, and one that would create a new revenue stream. They dubbed it DecisionSpace. This new project incorporated many of the Reno developers but further broke development into small sub-teams. One of these sub-teams was the infrastructure team, primarily responsible for the common infrastructure that would be the cornerstone for integration activities.

The DecisionSpace team held a meeting to discuss the overall scope of the project. One output of this meeting was the desire for a more agile development framework; extreme programming was specifically considered for use. (Farmer, 2004) Farmer explains that the overall scope continued to be very vague; however, as the project manager for the infrastructure team, she began planning future iterations for their components. Farmer continues and discusses the failures of DecisionSpace's initial development. (Farmer, 2004) She explains that the infrastructure components began to take shape, however the vague scope did not lend itself well to the application teams. Farmer thinks that issues arose out of the marketing team working on use cases and their inexperience working with such early-release software.

Mentioned earlier, after several months it became clear to management that DecisionSpace's wheels were stuck and something needed to be done. Farmer says that major takeaways from the first few months included working on code even though it might be wrong, and the value in capable people. Although the project would do a bit of reinventing, much of the code would be reused in the subsequent effort. Capable people also lent themselves as good decision makers in the absence of guidance. (Farmer, 2004)

It was determined that in order to move forward, the applications team should define a deadline. This deadline helped to solidify the scope, which in turn gave developers more direction. Management also replaced several of the top-level project managers and further assigned a sponsor who was both technical and had domain knowledge as to what the application should do. One of the first directives from the sponsor was to "ship." This triggered Farmer's team as well as others to do an internal delivery of their software thus far. (Farmer, 2004) This internal delivery helped the application and infrastructure team further solidify requirements and scope. The sponsor also assigned a product manager from R&D who further helped define scope.

At this point, things started to shape up for the development teams. The teams began to work in cycles, similar to XPs releases which were on a three-month basis. Iterations, similar to scrum's sprints, lasted three weeks in duration. Farmer explains that the teams used the first three iterations to develop and deliver new functionality, whereas the final three week iteration was used to refine and stabilize the product. As the DecisionSpace product matured, releases began to incorporate two stabilization iterations instead of just the one final iteration. Farmer describes these three month software releases as providing clarity for people outside of the infrastructure team. People were able to visualize what was actually being delivered.

Landmark Graphics learned two lessons from their endeavor: the importance of a sponsor to apply focus, and the need for a well-defined scope. Landmark Graphics was plagued by the lack of consistent scope which hampered code development. In their latest endeavor, the waterfall model was specifically cited as a major hurdle due to the large number of unknowns in the project and waterfall's inability to cope. It was this large number of unknowns which also hindered the development of a refined scope. The sponsor aided by applying focus when vision was lacking. Landmark Graphics' management was also keen to their customer's business needs and was able to provide insights in times of crisis.

Farmer continues by describing Landmark's implementation of unit testing, continuous builds, and their communication difficulties with teams in 4 different cities and multiple time zones. The team struggled at first to implement unit testing. Developers initially felt this was a waste of their time. Unit tests, however, became a central part of the published metrics for the project and Farmer writes that developers began writing unit tests once these metrics were graphically presented in meetings.

Landmark Graphics' implementation of continuous builds helped with the products' overall stability, something the prior project, Reno, had lacked. Landmark implemented continuous builds by triggering a new build anytime code was checked in. If this code caused an unsuccessful build, a series of emails were sent out to the developer who checked in the code and several other individuals. These emails continued every ten minutes, which put a bit of peer pressure on the developer whose code failed to compile. Farmer says this greatly enhanced build stability.

Communication was handled via two main avenues: traditional, face-to-face interaction or phone calls, and through information distribution taking advantage of the company's intranet. As far as traditional communication, nothing worked quite as well as face-to-face communication. Teleconferences offered a level of success and worked often, however employees preferred face-to-face communication. Information distribution took place via various intranet resources. The individual responsible for configuration management put together a well-organized site that pointed to builds, metrics, and instructions which proved to be a great resource.

Farmer believes their teams would have been more productive if they were collocated instead of geographically dispersed. Farmer also states that the move to become more agile came from the team instead of management. This push from the team made individuals less tolerant of errors which increased build quality. Peer pressure was cited as another success factor. (Farmer, 2004)

Overall, Landmark Graphics' move to agile was a positive one. Farmer notes that although they started by trying to implement extreme programming principles, they more closely implemented the broader principles of the agile manifesto. She notes that they especially valued working software over documentation and responding to change over a plan. Farmer points out several strengths including teams consisting of top people in the company, their ability to "find their own way," and management support. (Farmer, 2004) DecsionSpace would later become a success and provide Landmark Graphics' with a revenue stream for years to come. In addition to DecisionSpace, Landmark would begin several other development efforts, using and applying their agile development lessons learned.

Case Study: Saber Airline Solutions

The University of North Carolina conducted a study of Saber Airline Solutions' introduction of agile principles, specifically extreme programming. (Layman, Williams, & Cunningham, 2004) The university's study focused specifically on providing quantitative data regarding agile implementations. The study focused on two releases of the same software product by Saber Airline Solutions, one before the introduction of Agile and the second two years after its introduction. The university points out that Saber implemented extreme programming in a nearly pure form. It did not need much modification to fit their business. (Layman, Williams, & Cunningham, 2004) Saber's product consisted of a scriptable GUI environment for the development of custom business software. The university specifically looked at five null hypotheses and worked to provide evidence either supporting or refuting these hypotheses. The hypotheses roughly state that XP practices lead to no change in pre-release quality, post-release quality, programmer productivity, customer satisfaction, and team moral. (Layman, Williams, & Cunningham, 2004)

The university used the extreme programming evaluation framework (XP-EF) for benchmarking Saber's agile initiatives. They explain in their study that the XP-EF is made up of three parts: XP context measures, XP adherence metrics, and XP outcome measures. (Layman, Williams, & Cunningham, 2004) The study compared the third and ninth product releases and gauged each one using these three measurement categories. The older release, which was developed using the waterfall model, took roughly eighteen months to complete. The university research team notes that they were not present for the earlier release and only partially present for the later release. (Layman, Williams, & Cunningham, 2004) They were, however, granted access to source code, defect tracking systems, build results, and survey responses. The study describes findings in each of the three framework categories.

Context XP factors include sociological, geographical, project-specific, technological, ergonomic and developmental factors. The university did a comparison of the above dimensions between releases. Of sociological interest, the team size grew from six people to ten. (Layman, Williams, & Cunningham, 2004) The team's overall experience was greater, and turnover rates were lower. Geographically, the only major difference was Saber's customer base more than doubled between the third and ninth releases. Project-specific factors of interest include a decrease in person months from 108 for the older release to 14.7 for the latest release. (Layman, Williams,

& Cunningham, 2004) The overall elapsed time dropped to 3.5 months from 18. Technical factors included a move from waterfall to extreme programming, Microsoft Project to the planning game (a style of project management), and reusable materials to include unit test suites and automated build machines. They also moved to a more open floor plan for ergonomic reasons, getting rid of semi-private cubicles. Developmental factors that changed were a cultural shift from plan driven development to more agile methods. (Layman, Williams, & Cunningham, 2004)

Adherence metrics help to identify extreme programming's core value adoption for case study comparison. The article's authors note that this set of metrics was somewhat incomplete and anecdotal evidence was collected from the team leader who was present for both software releases. (Layman, Williams, & Cunningham, 2004) The adherence metric is made up of three main measurements, planning, code, and testing adherence. The authors also used the Shodan Adherence Survey which provided a subjective means to measure adherence. Respondents assign a percentage from 0 to 100 for how well they used an agile practice.

The planning adherence metrics showed a release length decrease from 18 to 3.5 months and an iteration length of ten days. Subjective results compiled from the Shodan survey showed that most developers found value in all measured metrics. These included in descending order of importance, standup meetings, short releases, on-site customers, and the planning game. The development team specifically said standup meetings provided great team communication and problem resolution. The team found planning only a few weeks at a time very beneficial and provided more accurate schedules when compared to waterfall. (Layman, Williams, & Cunningham, 2004)

Code adherence metrics measured pairing frequency, inspection frequency, and solo frequency. The older software release did not take advantage of pair programming, but did utilize pair review. Interviews revealed that some developers did not care for pair programming in certain instances such as when a large age difference exists between the programmers, levels of expertise are different, or the development task is trivial. (Layman, Williams, & Cunningham, 2004)

Code adherence metrics showed that pairing occurred 50% of the time with the new release, whereas it did not exist in the older release. The inspection frequency dropped from 60% in the older release to 20% in the newer release, replaced by the pair programming measured earlier. Also solo coding dropped to 50% from 100% in the older release. This is the inverse of the pair programming metric above. The above metrics were based on anecdotal information. High scoring subjective measurements conducted via the Shodan survey indicated a high use of coding standards, followed by continuous iterations, sustainable pace, and simple design. (Layman, Williams, & Cunningham, 2004)

Testing adherence metrics demonstrate how well an organization has implemented unit testing and automated scripts. Saber wrote an associated test class for almost every new class they wrote. The team indicated that test scripts were difficult to write for some GUI components, but believed this was just a limitation of current testing technology. Subjective measurements showed that developers believed they implemented automated unit tests 78% of the time (close to the quantitative measure of 80%), tested the first design 67% of the time, and performed customer acceptance tests 56% of the time. (Layman, Williams, & Cunningham, 2004)

Once the university had collected the above information they measured defects from Saber's defect tracking system. Saber tracked two different types of defects, those found internally and those by a customer. Each defect also had an associated severity rating indicating the impact the defect caused to the application. (Layman, Williams, & Cunningham, 2004)

In interviews with developers regarding their XP experiences, all team members pointed to increased communication as the most beneficial aspect of the XP implementation. Standup meeting were praised, as was the general increase in team communication. Layman et al stated that "often one developer would overhear a problem someone else was having and was able to suggest a solution or the two were able to collaborate on the problem." (Layman, Williams, & Cunningham, 2004) Programmers also commented on the usefulness of immediate feedback available from unit testing and pair programming.

Issues surrounding Saber's implementation of agile included developers' annoyance with mandatory pair programming, especially on trivial tasks. Saber also restricted the development labs for increased communication by breaking down some cubicle walls and rearranging furniture for a more open floor plan. Developers enjoyed the increased communication but also lost some ability to concentrate with the higher noise levels.

Laymen et al compared the business impact on agile by comparing measurements related to code quality, productivity, morale, and customer satisfaction. The study concludes that internal quality increased by 65% (Layman, Williams, & Cunningham, 2004), likely due to the increase in testing and its automation through unit tests. External defect counts dropped by 35% , an especially surprising statistics given that the number of customers for the newer product doubled. Evidence also suggested that productivity increased by 50% between the old and new release. (Layman, Williams, & Cunningham, 2004)

Customer satisfaction had to be answered anecdotally because customers could not be contacted in regards to their satisfaction with the product. One customer did explain to the team that they felt their latest release appeared to be one of the most polished applications they have used. (Layman, Williams, & Cunningham, 2004)

The Shodan survey also asked developers how often they could say they enjoy their job. It was the results acquired from this question that formed the basis for the morale measurement. Most developers stated their preference for the agile method over the prior waterfall model. Sustainable pace was also referenced as a major plus for agile.

The Saber case reiterates many of the common themes other cases have pointed out. The Saber case does go a step further and tries to quantify as best they can the case for agile. The extreme programming evaluation framework was introduced providing a consistent measure for cross case comparison. The quantitative metrics used to measure qualitative attributes was of interest.

Conclusions

Development models and methods are not a one-size-fits-all mantra. Development models and methods need to be tailored and fitted for an organization and project. The waterfall model is well suited for organizations with larger scale projects where requirements are well known and understood. The spiral model is suited for organizations with lesser known requirements. Agile methods, however, are best suited for younger organizations with the need to rapidly deliver a product to market or a specific customer. As the cases point out, many organization have begun to transition to more agile methods with the promise of increased productivity, a higher quality product, and higher workforce morale. In most cases, organizations seem pleased with their implementation of agile methods, almost all however met some level of resistance with their introduction, or specific agile methods failed completely.

The type of project or product plays an important role in the success of agile. The product needs to be one that is not mission critical and can grow with time. Agile methods take advantage of iterations which constantly improve the product and bring it closer to customer expectations. Iterations help agile adapt and meet certain business requirements as well. The iterative nature of agile allows a customer to often see what they are asking for very early on in the development process. As the DecisionSpace and Intel cases point out, the on-site customer (or someone acting in their stead) is of great importance to the development team, providing constant, constructive feedback.

Communication was a common theme in all of the cases. While organizations had qualms with certain agile aspects or increased communication implementations, none of the organizations regretted it. Increased communication was cited as one of the best problem solving techniques and team building exercises. While most organization did not find too many drawbacks, Greenee points out "a downside to the increased collaboration that agile methods foster and encourage is that is exposes people to each other to a greater extent than some are used to or comfortable with." (Greene, 2004)

Increased testing through the use of automated scripts and unit testing was another commonly discussed agile method which had an extraordinarily positive impact. As Greene states, "The emphasis of agile methodologies on testing is extremely valuable, and this brought about the biggest change to the bottom line: the quality of our code." (Greene, 2004) Developers enjoyed the immediate feedback given by unit tests and as the Saber case quantitatively points out, the increased amount of testing yielded a more bug free product for customers.

Iterations are one of the cornerstone methods of agile. The ability for a product to evolve and adapt to changing requirements or business needs is paramount in certain industries. The DecsionSpace case points this out when Farmer speaks to the infrastructure team's progress when compared to the waterfall model. "The Infrastructure team had been working in iterations since the beginning, so was able to make some progress, but the waterfall, plan-it-all first approach was deadly when there were so many unknowns." (Farmer, 2004)

Lastly, culture and people play a very important role in agile's introduction and adoption. As Conboy et al point out in their interviews of 17 companies, agile often was best implemented in organizations where it took hold in a grassroots form as opposed to a top-down approach through management. The same article also touches on a developer's age playing a role in how accepting they are of agile principles in general. Organizations should plan through their shift to agile with culture and people in mind.

The decision to implement agile principles into an organization often hinges on several key factors. Is the organization culturally ready for the shift? Is the shift mandated or wanted? Will iterations work for us? How do we measure employee performance? Do we have the right working atmosphere? These types of questions need to be asked and answered by organizations seeking to implement agile methods. It is equally important to remember that methods cannot simply be implemented or mandated; they need to massaged and tailored towards the organization. Agile's increased communication for example, was tailed for a globally diverse organization in the DecsionSpace case. Here teleconferencing and screen sharing technology was used to foster communication over geographically dispersed locations. Pair programming was limited and needed to be tailored for Intel's Itanium team due to language and domain knowledge limitations.

If an organization is contemplating a more agile approach, the organization needs to ask themselves the right questions. If they ask the right questions their journey will likely end in a happier, more productive workforce.

References

Beck, K. (2001). Manifesto for Agile Software Development. *Agile Alliance*.

Conboy, K., Coyle, S., Wang, X., & Pikkarainen, M. (2011). People over Process: Key Challenges in Agile Development. *IEEE Computer Society*.

Copeland, L. (2001, December 3). Extreme Programming.

Farmer, M. (2004). DecisionSpace Infrastructure: Agile Development in a Large, Distributed Team. *IEEE Computer Society*.

Greene, B. (2004). Agile Methods Applied to Embedded Firmware Development. *IEEE Computer Society*.

Hayata, T., & Han, J. (2011). A Hybrid Model for IT Project with Scrum. *IEEE*.

Layman, L., Williams, L., & Cunningham, L. (2004). Exploring Extreme Programming in Contect: An Industrial Case Study. *IEEE Computer Society*.

Miller, G. G. (n.d.). *The Characteristics of Agile Softwre Processes.*

Royce, W. (1970). Managing the Development of Large Software Systems. *IEEE.*