Applying Agile Practices to Improve Software Quality

White Paper

Abstract
Increased frustration with failed software projects coupled with the need to keep up with rapidly changing business needs is driving software development organizations to revisit the way they go about building software. Agile development has emerged as one possible solution to the woes of the software industry. Agile enthusiasts claim significant increases in the quality of their software while detractors cite instances where rapid development and loose structure lead to decreases in quality. This happens because not all 'agile' is created equally. Some agile practices are more likely, when implemented correctly, to impact quality than others. This article discusses specific agile practices which have been proven to have a positive impact on quality and offers practical advice about how best to implement them to maximize their impact on quality.

PRICE Systems

PRICE Systems provides more than 40 years of cost estimating software expertise across a variety of government and commercial market applications and business needs. PRICE cost models benchmarked from 16,000 major projects, ongoing PRICE research into refining predictable cost estimates, and new PRICE frameworks for integrating client-specific cost history, put PRICE Systems at the forefront of cost estimating and cost management solutions. For more information, visit PRICESystems.com

Applying Agile Practices to Improve Software Quality

Contents

Increased frustration with failed software projects coupled with fast-paced software development efforts driven by rapidly changing business needs is driving software development organizations to revisit the way they go about building software. Agile development has emerged as one possible solution to the woes of the software industry. Agile enthusiasts claim significant increases in the quality of their software while detractors cite instances where software development quality decreases as a result of the introduction of agile practices. How do business leaders decide whether agile is a higher quality solution then their current practices?

As with most software topics, there is no easy answer. There are many different methods that that are loosely grouped together under the agile umbrella (Xtreme Programming (XP), SCRUM, Lean Software Development, etc) . Most organizations then tailor a particular method by choosing a subset of the agile practices it encompasses. Therefore not every organization that is dong agile is doing the same thing. Add to this the fact that agile practices are better suited to certain types of projects and certain organizational philosophies than others. Organizations with projects or cultures that don't adapt well to agile are less likely to show quality successes. It is important for business leaders to understand not only which agile practices address software quality but also the potential roadblocks to agile success for their organization.

This paper studies agile development with an eye toward its impacts on software quality. The first section contains an overview of agile development methods and practices. In the second section software quality is discussed. The next section reviews agile practices which have shown to effect software quality along with an analysis of the circumstances that might alter this effect.

Agile Development

As documented in the agile manifesto[1]…

"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."

Agile development was introduced in the mid '90's as organizations were struggling with fast-paced software development projects brought on by changing business cultures. Its premise is that productivity and quality are derived from the techniques and disciplines used and our interpersonal relationships.[2] Agile practices focus on simplicity, customer focus, shared responsibility, close collaboration, and frequent and direct communication. More traditional methods focus on well documented and reviewed work products at each phase of the software development lifecycle. They require that the entire release be understood and designed before coding start. The agile philosophy assumes that things will change before the release is complete, thus some of the upfront design and planning effort is wasted. Agile requires the development team to break a software project into small pieces of functionality, developing enough code to deliver only this small piece of functionality, and then frequently building incremental releases to be reviewed by customers. Agile practices allow for continuous integration and continuous customer reviews and feedback.

Agile development is based on several values (different forms of agile name them differently): communication, simplicity, feedback, courage and respect. While different organizations implement different agile practices – these core values are a constant theme. Communication can be accomplished through various mechanisms including daily stand-up meetings, iteration planning and retrospective meetings, and customer walkthroughs. Co-location of the development team is also a catalyst for improved communication. While this does tend to create a culture of interruption, it leads to an environment where no one is afraid to ask for help, advice, suggestions or ideas.

Basic agile concepts include:

- Test Driven Development – unit tests are written prior to any code, developer writes just enough code to pass the test
- Simple Design – do the simplest thing that can possibly work without speculating about future features
- Pair Programming – all production code is written in pairs as an on-going collaboration
- Refactoring – improving existing code without changing functionality based on lessons learned through implementations
- Continuous Integration – integrations occur hourly or daily and automated tests are applied
- Collective ownership – all developers are responsible for the integrity of the code
- User Stories – stories describe a piece of system capability to be implemented, a placeholder for an on-going conversation about requirements
- Short iterations – Frequent releases (not necessarily external) that deliver business value

Software Quality

Wikipedia defines software quality as a measure of how well software is designed (quality of design) and how well the software conforms to the design (quality of conformance) [3]. This definition implies two aspects of software quality:

- Building the right thing
- Building it right

Building it right can be measured by a count of defects per size unit (Source Lines of Code, Function Points, Use Cases, etc.), number of passed tests, defect rate, etc. Determining whether the right thing is being built is somewhat problematic as it really is a measure of how delighted the customer is with the software that is delivered.

The literature contains many examples of studies and experiments (both academic and in industry) that indicate agile improves both software quality and customer satisfaction. The Data & Analysis Center for Software (DACS) published a report in 2007 [4] cites many instances where agile practices have improved quality both with reduced defects and improved customer satisfaction.

Agile Practices Most Likely to Impact Software Quality

While it's generally the combination of agile practices that lead to quality improvements, there are several practices, when implemented correctly that specifically target quality improvements. These are described in more detail in the following paragraphs.

Pair Programming

Pair programming is often cited as a reason for increased software quality. With pair programming two programmers are paired together to complete a single programming task. They have one computer between them and they collaborate to determine the best design and best implementation for that design. Pair programming applies the “two heads are better than one” paradigm while enforcing on-going continuous review of the code.

While the results are mixed on whether pair program increases or decreases productivity, its impact on quality has been documented in many studies. In a study reported on in [5], comparing programs developed by single programmers to those developed by pairs, the number of post development tests that passed increased by 15%. The report also presented experiential data that pair programming increased the quality of designs as well. [6] cites two examples supporting the quality benefits of pair programming – an experiment conducted at the University of Utah finding an average of approximately 14% increase in passed tests on programs using pair program and experiential findings on the Chrysler Company’s C3 project in 1997.

It is important that pair programming be implemented effectively. Pair programming is not just having one programmer looking over the shoulder of another as they type. At any given time, one programmer assumes the role of driver, sitting at the keyboard and writing code or tests. The job of the other team member is to navigate. This role requires that the programmer be thinking through the problem solution looking for better, cleaner ways to solve it while at the same time keeping an eye on the work product of the driver, scanning for mistakes in logic and coding. It is important that roles shift during the execution of the task. It is also important that within a development team, many different pairing combinations occur. If the same two programmers are always pairing with each other, the pairing may get stale and other benefits of pair programming (such as on the job training and team awareness of the code) will be lost. The more eyes on the code the more likely it is to be clean and error free.

There are situations where pair programming might not be effective. Distributed teams will find it difficult to be successful with pair programming although [2] suggests that techniques and tools to help distributed teams realize some of the benefits of pair programming. It could also be problematic in environments where telecommuting and flexible schedules are part of the culture. There are also types of project which may not be well suited to successful pair programming – where technologies and/or software content is very diverse within the organization requiring highly specialized programmers.

Test Driven Development (TDD)

Test driven development requires that no code is written for a feature until the tests for that feature have been written and shown to fail. Before a developer can begin to code for a particular feature they need to understand the requirements, intent and exceptions of the feature well enough to write tests for it. The process for each new feature begins with a study of the user story in order to assess requirements and write an automated test for the feature. This test is then executed to ensure that it fails (since the feature has yet to be written). Once the test has been proven to fail, the developer writes what they believe to be the minimal amount of code to make the test pass. Once enough code has been written to make the test pass, the developer then incrementally refactors the solution to make it simpler and cleaner. Because each increment includes rerunning of the test, the developer can refactor with confidence that the feature will not negatively impacted.

Tests conducted at Microsoft in two different environments showed defect rate (defects/KLOC) decreases at factors of 2.5x and 4.2x between projects determined to be similar in size and scope, one with TDD and one without.[7] Both tests indicated increased development time as well but not nearly as significant as the increase in quality. [8] contains two tables which summarize the results of 18 studies – 9 from academia and 9 from industry. These studies ranged from controlled experiments, to quasi-controlled experiments to case studies and the length of studies ranged from 1.75 hours to 1.5 years. Different measures of quality were employed by different studies (functional tests passed, defect rates) Of the studies, 10 found mild to significant improvement in quality, 7 were inconclusive or showed no difference while only one of the studies indicated a decrease in quality. Lisa Crispin in [9] correctly points out that there is more to quality than defect counts or passed functional tests. Her teams employ customer test driven development in addition to developer test driven development. Testers work with customers to develop high level functional tests for features. The developers use these tests to fully understand the user's requirements. The result is happy (often delighted) customers – contributing to the 'doing the right thing' aspect of quality.

Clearly there are factors that will make TDD more or less successful as a quality initiative, but it seems to deliver promise in many circumstances. A (non-scientific) perusal of the studies that have been accomplished to date seems to indicate that smaller projects and studies see less quality increases than larger projects This could be an indicator that the investment may not be warranted strictly as a quality initiative, although TDD reaps other benefits such as documentation of design details (through the test artifacts) and smoother maintenance over time as automated tests constantly monitor the impacts of changes to the code base. Other items important to the success of TDD include concrete understanding of the expectations of TDD, effective training and an organizational culture friendly toward TDD.

Continuous Integration

Continuous Integration is an agile practice that requires that changes to the code base be continuously integrated into an operational system. Generally this process is automated and integrated with an automated test suite in order to give real time feedback when a developer makes a change that causes bad behavior in unexpected places in the system. During development, individual developers will 'check out' a copy of the code, make changes to fix a bug or add a feature, tests this change and then 'check in' the changed code. During this time, other developers have made changes to other parts of the code base. A common problem arises when the 'check out' time is too long because while each individual developer is able to pass the suite of tests with their instance of the code, the combination of their code with the changes of other developers causes tests to fail.

Continuous integration requires that an application is rebuilt and tested each time a change is made to the code base. Some shops automate this process through their version control system while others have automatic builds and tests that run every couple of hours with developers checking in code often. Solving any failures in these tests becomes a top priority for the team. When there are failures found during integration it is easy to isolate the problem code and correct the problem. The longer the period between integrations the more likely the team is thrown into integration hell – unable to determine which of many code changes is responsible for failed tests.

While there doesn't appear to be much quantitative evidence specifically relating continuous integration to increases in quality, it has been observed that projects using continuous integration tend to have dramatically less bugs in production and in process [10]. Steve McConnell in [11] indicates that one of the benefits of frequent builds is reduced risk of low quality. Clearly this is influenced significantly by the number and quality of the tests and the amount of automation applied to the testing for each build. Quality impacts also depend significantly on how seriously the team acts when tests do fail. If failed tests are not treated as a top priority the value of continuous integration is seriously depleted.

Short Iterations

Most agile shops 'release' software with added business value every couple of weeks. These releases may never see the light of day outside of the development group and customer stakeholders but the team works toward each release as though it were intended to be delivered to a customer. The team basically transcends a whole 'waterfall' inside the iterations with planning, requirements analysis, test development, coding and integration and test. Although the time and effort associated with creating production ready releases frequently introduces overhead that more traditional projects don't experience, there are several good reasons why creating software in short iterations makes sense. If a release is built every few weeks with incremental implementations of features, the customer has the opportunity to see how the development team is interpreting their requirements and has the chance to redirect efforts when they deviate from their expectations. If a software project gets into trouble and falls behind the customer has the opportunity to reprioritize requirements so the best set of functionality gets to market in a timely fashion. Frequently releases also give the development team and the customer lots of time to perform functional tests on features as they emerge.

As with continuous integration, there is little quantitative evidence that specifically speaks to short integrations increasing quality. Certainly early and frequent visibility of implemented features to the customer is highly likely to increase the probability that the software implemented 'does the right thing' contributing to delighted customers. Additionally, frequent small releases give the test team time to focus on features as they emerge, increasing the chance that defects are detected and addressed as they emerge.

Conclusion

Agile development in all of its many implementations has been proven in many, instances to decrease defects and increase customer satisfaction. There is also research to support the opposite conclusion. This disparity stems from the fact that different organizations adopt different forms of agile and often adopt different sets of agile practices within the form of agile they choose. Some agile practices, or combination of agile practices, if implemented correctly, are more likely to result in quality improvements than others. Practices such as pair programming, test driven development, continuous integration and short iterations have all be cited as components of a good quality improvement initiatives.

In addition to practices targeted to instilling quality in the software, agile brings another quality enhancer to the table. All forms of agile encourage frequent and meaningful communication both within the development team and with the customer. The best way to ensure that the software that is delivered meets the customer needs and does the right thing is to have on-going conversations with the stakeholders and between the developers.

References

1. Agile Alliance, “Agile Software Development Manifesto”, Feb 2001, available at www.agilemanifesto.org (retrieved January 2010)

2. Kile, James F., “An Investigation into the Effectiveness of Agile Software Development in a Highly Distributed Environment”, Dissertation, Pace University School of Computer Science and Information Systems”, November 2006

3. Available at http://en.wikipedia.org/wiki/Software_quality (retrieved January 2010)

4. DACS, “A Business Case for Software Process improvement, Measuring Return on Investment from Software Engineering and Management, A DACS State of the Art Report”, September 2007

5. Cockburn, A,, et.al., “The Costs and Benefits of Pair Programming”, available at http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF (retrieved Jan 2010)

6. Williams, L., et. al., “Strengthening the Case for Pair-Programming”, available at http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.PDF

7. Bhat,T, et.al, “Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies”, ISESE 2006, Sept 21-22, 2006, Rio de Janiero, Brazil, ACM 1-59593-218-6/06/2009

8. Jeffries,R., et. Al. “TDD: The Art of Fearless Programming”, IEEE Software, May/June 2007, pp24-30

9. Crispin, Lisa, “Driving Software Quality: How Test-Driven Development Impacts Software Quality, IEEE Software, November/December 2006, pp70-71

10. Fowler, Martin, “Continuous Integration”, http://www.martinfowler.com/articles/continuousIntegration.html#BenefitsOfContinuousIntegration

11. McConnell, Steve, “Daily Build and Smoke Test”, IEEE Software Vol 13, No 4, July 1996

About the Author

Arlene Minkiewicz is a software measurement expert dedicated to finding creative solutions focused on making software development professionals successful. She has over 29 years in the software industry, researching all aspects of the software development process and providing thought leadership to the software community. Ms. Minkiewicz is the Chief Scientist at PRICE Systems, LLC. In this role, she leads the cost research activity for the entire suite of cost estimating products that PRICE provides.