Scrum + Engineering Practices: Experiences of Three Microsoft Teams

Laurie Williams
North Carolina State University
Raleigh, NC 27695, USA
williams@csc.ncsu.edu

Gabe Brown, Adam Meltzer, Nachiappan Nagappan
Microsoft Corporation
Redmond, WA 98052, USA
{gabeb, ameltzer, nachin}@microsoft.com

***Abstract* - The Scrum methodology is an agile software development process that works as a project management wrapper around existing engineering practices to iteratively and incrementally develop software. With Scrum, for a developer to receive credit for his or her work, he or she must demonstrate the new functionality provided by a feature at the end of each short iteration during an iteration review session. Such a short-term focus without the checks and balances of sound engineering practices may lead a team to neglect quality. In this paper we present the experiences of three teams at Microsoft using Scrum with an additional nine sound engineering practices. Our results indicate that these teams were able to improve quality, productivity, and estimation accuracy through the combination of Scrum and nine engineering practices.**

Keywords - Agile software development, Scrum

I. INTRODUCTION

Scrum [16, 29] is the most often used [6, 30, 31] agile [10] software development methodology among teams that utilize an agile methodology. A large-scale survey [31] deployed in the software engineering industry from June/July 2008 received 3061 respondents from 80 different countries. For the question "Which Agile methodology do to you closely follow" 49% of the respondents mentioned Scrum and an additional 29% mentioned Scrum with Extreme Programming. Additionally, a survey of 10% of all engineers at Microsoft, indicated that more than 60% of the engineers use Scrum (Figure 2) [6]. Scrum provides a project management structure to a team. However, the Scrum methodology does not prescribe the engineering practices a team should use, purportedly to give organizations as much flexibility as possible in choosing their engineering practices.

With Scrum, gone are the days of a software developer reporting to the project manager that a new feature is 80% complete. Instead, in a Scrum environment, credit is "all or nothing" whereby a feature that is 99% done is considered "not done." For the developer to receive credit for his or her work, he or she must demonstrate the new functionality provided by a feature at the end of each short iteration during an iteration review session. Developers, testers, the project managers, the product owner/manager, and others attend this iteration review session. The expectations for demonstrating all planned features at the end of each iteration are high as the team works to meet its iteration goal.

This short-term focus of iterations coupled with a lack of prescribed engineering practices may lead to trouble. "Flaccid Scrum[1]" is a term coined by Martin Fowler to refer to teams that utilize only Scrum's project management practices. Progress eventually slows for Flaccid Scrum teams, according to Fowler, because the team has not paid enough attention to the quality of the code produced during each iteration. In some cases, only the easiest scenario of a feature (often referred to as the "happy path") is demonstrated at the end of the iteration. This "happy path" may be formally specified as the acceptance criteria for the feature. The feature can then be considered to be "done", with the development team getting credit for the feature. Focus then turns to a new set of commitments to deliver features for the next iteration, even if only the "happy path" of prior features has been done.

The Scrum methodology, however, may provide a solid project management framework for a team that also utilizes sound engineering practices. In this paper, we share the experiences and quantitative productivity and quality results of three Microsoft teams who utilized a Scrum-based software development methodology augmented with nine engineering practices recommended by the Microsoft Engineering Excellence group that takes care of companywide process initiatives.

Software engineering research can aid practitioners in their technology and/or process choices. Practitioners who read this paper will gain an understanding of the need to add engineering practices to a Scrum process to prevent Flaccid Scrum.

The rest of this paper is structured as follows. We explain Scrum and provide background in Sections 2 and 3. We provide the motivation for our paper in Section 4. In Section 5, we describe the practices adopted by the team. We then provide the results of the teams in Section 6 and limitations of our study in Section 7. We summarize the study in Section 8.

II. SCRUM

The Scrum methodology is an agile software development process that works as a wrapper with existing engineering practices to iteratively and incrementally

[1] http://www.martinfowler.com/bliki/FlaccidScrum.html

[2] http://www.sei.cmu.edu/cmmi/

[3] http://www.nunit.org/index.php

[4] http://junit.org/

develop software. Scrum is composed of the following project management practices:

- The Product Owner creates the requirements, prioritizes them, and documents them in the Product Backlog during Release Planning. In Scrum, requirements are called features.
- Scrum teams work in short iterations. When Scrum was first defined [16, 29], iterations were 30-days long. More recently Scrum teams often use even shorter iterations, such as two-week iterations. In Scrum, the current iteration is called the Sprint.
- A Sprint Planning Meeting is held with the development team, testers, management, the project manager, and the Product Owner. In the Sprint Planning Meeting, this group chooses which features (which are most often user-visible, user valued, and able to be implemented within one iteration) from the product backlog are to be included in the next iteration, driven by highest business value and risk and the capacity of the team.
- Once the Sprint begins, features cannot be added to the Sprint.
- Short, 10-15 minute Daily Scrum meetings are held. While others (such as managers) may attend these meetings, only the developers and testers and the Scrum Master (the name given to the project manager in Scrum) can speak. Each team member answers the following questions:
  - What have you done since the last Daily Scrum?
  - What will you do between now and the next Daily Scrum?
  - What is getting in your way of doing work?
- At the end of a Sprint, a Sprint Review takes place to review progress and to demonstrate completed features to the Product Owner, management, users, and the team members.
- After the Sprint Review, the team conducts a Retrospective Meeting. In the Retrospective Meeting, the team discusses what went well in the last Sprint and how they might improve their processes for the next Sprint.

III. BACKGROUND AND RELATED WORK

In this section, we provide related work on the Scrum agile software development methodology. We also discuss case study research.

A. Scrum Research

A myriad of qualitative experience reports about the Scrum software development methodology have been published. However, few studies have been conducted on the Scrum that report quantitative results, as ours does. In this section, we summarize information available in the literature about the use of Scrum by industrial software development teams whereby the papers provided details beyond qualitative experience reports. A pattern among the published studies is that the successful Scrum teams also utilized proven engineering practices.

Tain, a Swedish gaming company, adopted Scrum and Extreme Programming (XP) engineering practices to develop an online poker game [21]. The team delivered a stable, scalable product on schedule. During this time, the team also was reduced in size by half and those that remained worked less overtime to produce more business value than previously. The engineering practices the team enumerated as critical to their success are the following: continuous integration, refactoring, and test-driven development.

A development team for an Internet portal utilized the Scrum methodology [12]. Initially, the short term focus of Scrum caused this team to ignore the use of some best engineering practices, leading to "cumulative and often irreversible" problems. Early in the development cycle, the team established source control, coding standards, processes for code reviews and check-ins, and informal rules for design discussions and team meetings. However, the team did not initially establish an automated build system, a unit test framework, or a practice of creating automated quality assurance tests. The eventual implementation of these practices aided the team in successfully implementing a higher quality product by a team with improved morale.

Two teams at Systematic utilized a Scrum-based process [18]. Systematic is an independent software and systems company focused on complex and critical information technology solutions. Systematic often produces products that are mission critical with high demands for reliability, safety, accuracy, and usability. In 2005, Systematic was rated a Capability Maturity Model – Integrated (CMM-I)[2] Level 5 company, an indication of its use of engineering best practices. Through the use of Scrum, Systematic estimates that it doubled its productivity and cut defects by 40%.

B. Case Study Research

The experiences shared in this paper can be classified as case study research. Case studies can be viewed as "research in the typical" [13, 19]. As opposed to formal experiments, which often have a narrow focus and an emphasis on controlling context variables, case studies in software engineering test theories and collect data through observation of a project in an unmodified setting [34]. However, because the corporate, team, and project characteristics are unique to each case study, comparisons and generalizations of case study results are difficult and are subject to questions of internal validity [20]. Nonetheless, case studies are valuable because they involve factors that

[2] http://www.sei.cmu.edu/cmmi/

staged experiments generally do not exhibit, such as scale, complexity, unpredictability, and dynamism [27]. Case studies are particularly important for industrial evaluation of software engineering methods and tools [19]. Researchers become more confident in a theory when similar findings emerge in different contexts [19]. By recording the context variables of multiple case studies and/or experiments, researchers can build evidence through a family of experiments. Replication of case studies addresses threats to experimental validity [2].

IV. MICROSOFT TEAM AND PROCESS

In this section we provide information on the three Microsoft teams included in our study that utilized a Scrum-based software development methodology plus engineering practices. We then discuss the software development process used by the teams.

A. Research Methodology

The second and third authors can be considered action researchers. They have participated as software engineers on the three teams. The first author obtained information about the teams' experiences by interviewing the second author using pre-prepared questions, which were intertwined with opportunistic follow-on questions based upon his answers. One interview was conducted on the phone and the second in person. Both interviews were approximately one hour in duration. The fourth author participated in the interviews. He also had numerous informal conversations with the two software engineers on the teams. The second and third authors provided quantitative data, which was interpreted and analyzed by the first and fourth authors.

B. Team Demographics and Context

Table I provides an overview of the context factors of the three teams. The context factors were motivated based upon those specified in the Extreme Programming Evaluation Framework (XP-EF) [33]. We do not provide information about the exact Microsoft products the teams implemented to enable us to share more information about the team's results. The domain of the each of the products is provided in the table and ranges from infrastructure to test infrastructure to mobile web applications. In all cases, the teams were working on the first release of their products in either C# or C++. The teams produced between 9 and 31 thousand lines of implementation code during a period of between 11 and 18 months long.

Teams A and B were small teams of between three and five members. Team C was larger with 19 members. Teams B was co-located teams while Teams A and C were distributed between the US and China, challenging the usual face-to-face communication advocated in the Agile Manifesto [5]. Other context factors presented in Table I will be discussed in Section 5.

C. Scrum-based Process Used by Teams

The three teams utilized a Scrum-based software development process and added nine additional engineering practices. These nine practices are Microsoft Engineering Excellence Best Practices. Microsoft Engineering Excellence is an organization responsible for supporting Microsoft's engineering community by providing the engineers with learning and development opportunities, and with discovering and propagating engineering best practices across the company and into the information technology (IT) ecosystem.

This sub-section provides information on the software development methodology used by the teams.

1) Basic Scrum

The teams utilized the basic practices of Scrum laid out in Section II. All teams began with a four-week iteration. Team A then transitioned to a two-week iterations. Team A team found estimating and planning for a two-week period easier than planning for a four week period. They also found the move to a two-week iteration allowed them to respond faster to changing business requirements reduced risk because the team was able to address issues more rapidly through more frequent iterations.

The teams performed "just-in-time" design of features before or during the iteration in which the feature was to be developed. The form of the design was often class diagrams and annotations on interactions with other major components. Team A team members in Shanghai sometimes created a prototype (called a "spike" among agile software developers) of larger features or those with significant unknowns before the feature was accepted into a Sprint. The purpose of the prototype was to gain knowledge about the feature prior to accepting the feature into a Sprint. Only when enough knowledge was available for the feature would it be considered ready for a Sprint. Such delaying of stories until adequate information is available was also done by the Systematic team discussed in Section III [18].

The teams only held the Daily Scrum three times per week in Redmond, Washington, USA. Subsequently, a Redmond team member would follow up with a call to their Chinese colleagues. The teams conducted retrospective meetings at the end of every Sprint.

2) Planning Poker

The teams used Planning Poker to estimate the person-hours required to complete functionality within an iteration. In recent years, some agile software development teams have estimated the effort needed to complete the requirements chosen to be implemented in an Sprint and/or in a release via a Wideband Delphi [8] practice commonly called Planning Poker [11, 14]. Several reports on the use of Planning Poker are found in the literature. One Norwegian industrial team "immediately took a liking to the new estimation process" [15] of Planning Poker, and the technique was "very well received." Another Norwegian

TABLE I: MICROSOFT TEAM CONTEXT FACTORS

	Team A	Team B	Team C
Project Management Type	Scrum	Scrum	Scrum
Team Size	4	3	19
Team Location	Redmond + Shanghai	Redmond	Redmond + Beijing
Experience > 10 years	1	1	1
Experience 6-10 years	2	0	10
Experience < 5 years	1	2	8
Domain Expertise	Medium	Medium	Medium/High
Language Expertise	High	Medium	High
Program Manager Expertise	Low	Low	Medium/High
Programming Lang.	C#	C#	C++
Team Location	Distributed	Local	Distributed
Domain	Infrastructure	Test Infrastructure	Mobile Web
Version/Legacy	1st Release	1st Release	1st Release
Source LoC	24,952	8,826	31,399
Test LoC	20,912	4,031	26,283
Test LoC / Source LoC	0.84	0.46	0.84
% of Code Coverage (unit-tests)	82%	53%	N/A
Development Time	14 months	11 months	18 months
Legacy Code	no	no	no
Test Run Frequency	Every check-in/daily	Every check-in/daily	Every check-in/daily
Actual Defects (Sev 1,2,3,4)	14 P1, 56 P2	76 P1, 111 P2	8 P1, 141 P2
Physical Layout	Offices	Offices	Office/shared space
Customer Communication	Onsite, email, chat, phone	Onsite, email, chat, phone	partners, in person, email
Customer Cardinality and Location	On-Site, Remote	On-Site	N/A

team [25] "decided to implement it for all tasks in the project's future" because they felt it was an effective means of collaboratively producing unbiased estimates.

Planning Poker is "played" by the team as a part of the Sprint Planning meeting. A Planning Poker session begins by the customer or marketing representative explaining each requirement to the extended development team. We use the term extended development team (often called the "whole team" [4] by agile software developers) to refer to all those involved in the development of a product, including product managers, project managers, software developers, testers, usability engineers, security engineers and others. In turn, the team discusses the work involved in fully implementing and testing a requirement until they believe that they have enough information to estimate the effort. Each team member then privately and independently estimates the effort. The team members reveal their estimates simultaneously. Next, the team members with the lowest and highest estimate explain their estimates to the group. Discussion ensues until the group is ready to re-vote on their estimates. More estimation rounds take place until the team can come to a consensus on an effort estimate for the requirement. Most often, only one or two Planning Poker rounds are necessary on a particular requirement before consensus is reached.

Planning Poker provides a structured means for:

- obtaining a shared understanding;
- exposing hidden assumptions of the technical aspects of implementation and verification;
- discussing the implications throughout the system for implementing a requirement;
- surfacing and resolving ambiguities realized via divergent perspectives on the requirement; and
- exposing easy and hard alternatives for achieving desired goals.

The Microsoft teams felt the use of Planning Poker enabled their team to have relatively low estimation error from the beginning of the project. Figure 1 depicts the estimation error for Team A (the middle line) relative to the cone of uncertainty (the outer lines). The cone of uncertainty is a concept introduced by Boehm [8] and made prominent more recently by McConnell [24] based upon the idea that uncertainty decreases significantly as one obtains new knowledge as the project progresses [22]. Team A's estimation accuracy was relatively low starting from the first iteration. The team attributes their accuracy to the use of the Planning Poker practice.

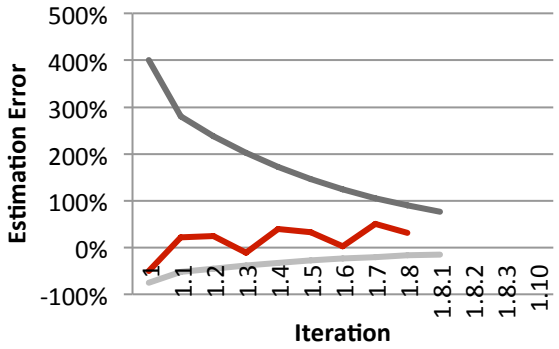

Figure 1: Cone of uncertainty

The teams indicate that the use of the Planning Poker practice required more upfront work prior to each Planning Poker session at the start of each iteration. This upfront work includes the product owner fully defining small user-visible, user-valued feature requirements that could be completed in one iteration or less, high level architectural analysis or prototyping, and possibly preliminary user-interface design. However, this upfront work enables the team to complete features in one iteration as uncertainty about the expectations for a feature has been reduced. The teams indicate that occasionally the Planning Poker voting results in a deadlock when no consensus can be reached. In most cases, the deadlock in estimation is desirable. The deadlock signals that the product owner has not fully described the work to be delivered or that a spike is necessary to investigate a significant unknown prior to the feature being accepted into a Sprint and that the feature needs to be put on the backlog until the investigation has been conducted.

3) Continuous Integration

The teams utilized the continuous integration practice. Continuous integration is a software development practice where members of a team integrate their work into the main build system frequently. Usually each developer integrates at least daily. Each integration is verified by an automated process that runs all automated tests that should detect integration errors as quickly as possible.

As shown in Table I, the Microsoft teams checked in their new code at least once per day. Each check-in initiated a build. Each build entailed the running of automated unit tests and associated test coverage computation. The team automatically received an email confirmation of the completion of the build providing test results. All tests must pass for the build to be considered successful. The used Microsoft Visual Studio Team Build Server to manage their check-ins, build process, and automated test runs.

The Microsoft teams managed their build/continuous integration process themselves rather than getting help from a build support organization. They indicated that the benefits from continuous integration's ability to keep a constant focus on quality come with a cost. Builds and test runs are not always successful, causing engineers to need to deal with issues such as bad merges, build system problems, and source control integration problems.

4) Unit Test-Driven Development

Unit test-driven development [3] is a practice that has been used sporadically for decades [7, 14]. With this practice, a software engineer cycles on a minute-by-minute basis between writing failing automated unit tests and writing implementation code to pass those tests.

Case studies [7, 26, 28, 32] were conducted with four development teams at Microsoft (Windows, MSN, Visual Studio, and one unnamed application) developed in C++ and C# and one IBM device driver team that developed in Java. All had transitioned from an ad hoc unit testing practice to the team-wide use of automated unit testing using the NUnit[3] or JUnit[4] frameworks. The TDD teams realized a significant decrease in defects, from 20% to 91%.

The main difference between the Windows, MSN, Visual Studio, and IBM teams versus the Microsoft application team was that the first four developed automated unit tests incrementally on a minute-by-minute basis. Developers took from 15% to 35% longer to achieve this quality gain. However, the quality improvement due to reduced defects, leading to less debug and field support time makes up for this increase in development time.

[3] http://www.nunit.org/index.php
[4] http://junit.org/

Teams A, B, and C did not write unit tests on an minute-by-minute basis. Instead, they backtracked to write automated unit tests after completing a major piece of functionality or completing a class. They also estimate that writing test cases added approximately 20% to their development time. As shown in Table I, the ratio of test lines of code (LOC) to source LOC ranged from .46 to .84. The test coverage ranged from 53% to 82%. Team B had the lowest test LOC/source LOC ratio and the lowest coverage. As will be discussed in Section VI, Team B also had the highest defect density. As indicated above, automated unit test runs were done as a part of the continuous integration build process. All testing was done using Visual Studio's test tools, such as the use of the NUnit test framework with the MSTest adapter.

5) Quality Gates (a.k.a. "Done Criteria")

Rather than tracking what percentage of a new feature is complete for an engineer, most agile teams use a binary "all or nothing" means of feature completion tracking. The feature is considered "not complete" until it is not only implemented but can pass all the quality "done criteria" that has been pre-established by the team. Done criteria is essential for preventing Flaccid Scrum. Sound 'done' criteria can prevent the team from rushing through the implementation of features such that a simple demonstration of a feature can be done in the Iteration Review meeting without the feature being robust enough to handle alternative flows and/or error handling.

The Microsoft team calls their done criteria "quality gates." The quality gates established for these teams included the following:

- All unit tests must pass
- Unit test code coverage must be at least 80% (for all teams except Team B)
- All public methods must have documentation
- All non-unit test code must not have any static analysis errors or warnings (see Sub-Section 9 of this section)
- Build must compile with no errors or warnings on the highest level

The team feels these quality gates provide concise and measurable exit criteria for their feature development, putting the focus on quality of features rather than quantity of features. However, the use of quality gates do, however, impose overhead on their process due to the need for monitoring.

6) Source Control

Source control is management of changes to documents, programs, and other information stored as computer files through a source control system. The Microsoft teams used the Visual Studio Team Foundation Server Version Control tool. Any contributor could check code into the development branch. Only project administrators could integrate code into the main branch. Code was moved from the main branch to the release branch at the end of each Sprint.

The teams felt that source control was beneficial for change tracking, branching, and merging. Source control was also used to manage access control to code whereby some could read code and others could contribute to code. Source control also provided a central managed store for data. Additionally, complicated use cases were challenging to manage since such use cases would involve a larger quantity of files that might be owned by other engineers.

7) Code Coverage

Engineers were required to manage their automated unit test coverage and monitored this coverage with each build. Two of the teams (A and C) followed the Microsoft Engineering Excellence recommendation of having 80% unit test coverage. The team felt that managing coverage was helpful for finding dead code and areas that needed better testing. Obtaining higher coverage was difficult due to the need to force tests to execute error conditions; coverage beyond the prescribed 80% might be considered to have diminishing returns. The teams did not consider high coverage as an indication that the code is of high quality, only a measure of unit test effectiveness. Unit tests may detect errors of commission, such as an incorrect computation or incorrect logic, but not errors of omission, such as missing functionality.

8) Peer Review

In each iteration, the teams conducted design reviews of architecture diagrams and of code when adding new features. The built-in Visual Studio code review tool was used. Senior developers conducted the reviews. When code was checked in, the reviewer(s) names were entered into the tool. The teams felt the reviews significantly improved the quality of the code by removing faults that may have escaped to the field.

9) Static Analysis Tools

The use of static analysis tools can identify common coding problems [17] or unusual code [1] early in the development process [9]. The teams utilized the FxCop static analysis tool built into Visual Studio. They also had static compiler warnings set to the highest sensitivity. Engineers had to explain to a senior engineer when they suppressed warnings from the compiler or FxCop and document their justification in the source code. They felt that the use of FxCop trained them to use better coding practices and was an effective learning tool. They also felt that peer reviews were more effective because fixing FxCop and compiler warnings caused them to find and fix the small errors. The teams felt that the use of static analysis tools can be difficult when not used from the beginning of a

project because the engineers can become overwhelmed with warnings.

10) XML Documentation

The team used .NET-style inline XML generated documentation on all public classes, properties, and methods. As a result the code was self-documenting.

V. RESULTS

The transition to the new Scrum process did temporarily reduce the productivity of the team. However, the teams had recovered and improved productivity by the end of the fourth iteration. Team A was able to achieve a 250% improvement, (to sustain aggressive code addition all while able to meet stringent quality gates) in the number of lines of code produced in each sprint by the fourth sprint (Figure 2). This improved productivity directly translated into capacity the teams leveraged to complete more user stories and Sprint tasks while meeting all of the quality gates.

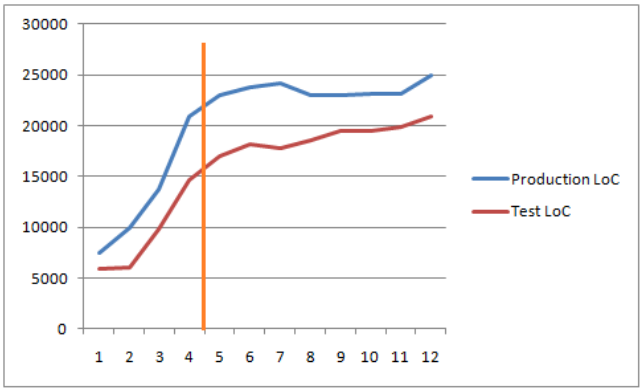

Figure 2: Cumulative Lines of Code Produced

Also in this paper we report on the various engineering practices that were followed during the course of software development using the Scrum development practice. To compare and contrast the quality of the software systems produced, we compare against prior published defect density rates [23] of a non-Scrum project at IBM. All three of the Microsoft projects were first releases so we could not use a prior Microsoft release as a baseline. The same project could not be repeated using a non-Scrum team as is common with case study research. hence and difference in comparison points. Further we also compare, in Figure 3, against data extracted from the Bangalore benchmarking group[5] where benchmarks were developed across 40 projects from nine companies (Honeywell, HP, IMR, Logica, Motorola, Novell, Philips, Verifone,WiproTech). Predominantly C, C++, Java are the languages used in these 40 projects. The average size of project was 43 KLOC (Min - 4 KLOC, Max - 300 KLOC). The average effort was 910 Staff/Person days (Min – 203 staff/person days, Max – 4664 Staff/person days).

The results in Figure 3 indicate that the Scrum projects had lower defect density (defects per line of code) that the non-Scrum projects except in the case of Team B. Team B's testing effort was relatively low (Source/test LOC ratio is 0.53) indicating the lowest effort amongst all Scrum projects. These results further back up our assertion on the importance of the engineering practices followed with Scrum (in this case more extensive testing) rather than the Scrum process itself.

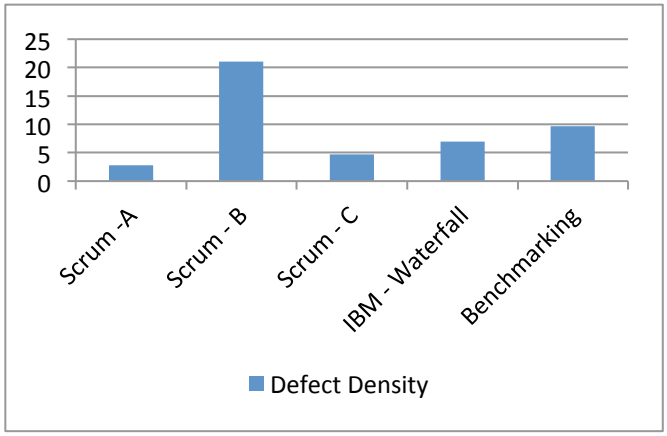

Figure 3: Comparing defect density for Scrum and non-Scrum teams

VI. LIMITATIONS

As with all case study research, our results are only valid in the context of these three teams and the results may not generalize beyond these three teams. The same teams could not repeat the projects in a non-Scrum environment. Therefore, our comparisons are not on the same projects. Additionally, all three Microsoft teams were relatively small teams so our research does not address scalability of Scrum to larger teams.

Also we compare the productivity of the teams relative to their productivity prior to Scrum. There could have been factors regarding expertise in the code base, which could have also contributed to these results. But considering the magnitude of improvement 250%, there would still have to be an improvement associated with Scrum even after taking into account any improvement due to experience acquisition.

The teams utilized all the basic Scrum practices as well as nine additional engineering practices. We cannot distinguish which of these practices were the biggest drivers in the productivity and quality results observed.

VII. LESSONS LEARNED

In this paper, we present the engineering practices that were part of the Scrum development process at Microsoft. The three teams at Microsoft used the following nine practices with their Scrum framework.

1. Planning poker

[5] http://www.bspin.org/archeives11/BSIG-SPINtalk-2000.ppt

2. Continuous integration
3. Unit Test-Driven development
4. Quality gates
5. Source control
6. Code coverage
7. Static analysis tools
8. Peer review
9. XML documentation

The productivity of the teams as they transitioned to agile temporarily dropped for three iterations. The team attributed this drop to their unfamiliarity of Scrum and required a "gelling" period to start delivering value based on the development process. From their fourth sprint on they experienced a significant improvement in productivity without an increase in defects. Teams transitioning to the use of an agile software development should plan for a similar temporary productivity decrease.

Teams that used Scrum and sound engineering practices showed better quality in terms of defect density compared with similar non-Scrum teams including data benchmarked across 40 projects from nine companies. These results indicates that Scrum combined with sound engineering practices have the potential to yield a higher quality product. Team B that followed Scrum but the engineering practices to a lesser degree than Teams A and C had the highest defect density.

Finally, our results indicate that estimation accuracy was enhanced by the use of the Planning Poker practice.

ACKNOWLEDGMENTS

Funding for first author Laurie Williams was provided by the ScrumAlliance.

REFERENCES

[1] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating Static Analysis Defect Warnings On Production Software," in 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, San Diego, CA, USA, 2007, pp. 1-8.

[2] V. Basili, F. Shull, and F. Lanubile, "Building Knowledge Through Families of Experiments," IEEE Transactions on Software Engineering, vol. 25, no. 4, pp. 456 - 473, 1999.

[3] K. Beck, Test Driven Development -- by Example. Boston: Addison Wesley, 2003.

[4] K. Beck, Extreme Programming Explained: Embrace Change, Second ed. Reading, MA: Addison-Wesley, 2005.

[5] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "The Agile Manifesto," http://www.agileAlliance.org, 2001.

[6] A. Begel and N. Nagappan, "Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study," in Empirical Software Engineering and Measurement Conference (ESEM), Madrid, Spain, 2007, pp. 255-264.

[7] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development: industrial case studies," in ACM/IEEE international symposium on International symposium on empirical software engineering, Rio de Janeiro, Brazil, 2006, pp. 356 - 363

[8] B. W. Boehm, Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[9] B. Chess and J. West, Secure Programming with Static Analysis, 1st ed. Upper Saddle River, NJ: Addison-Wesley, 2007.

[10] A. Cockburn, Agile Software Development. Reading, Massachusetts: Addison Wesley Longman, 2002.

[11] M. Cohn, Agile Estimating and Planning. Upper Saddle River, NJ: Prentice Hall, 2006.

[12] K. Dinakar, "Agile Development: Overcoming a Short-Term Focus in Implementing Best Practices," in Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) 2009, Orlando, FL, pp. 579-588.

[13] N. E. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach: Brooks/Cole, 1998.

[14] J. Grenning, "Planning Poker or How to avoid analysis paralysis while release planning," 2002, https://segueuserfiles.middlebury.edu/xp/PlanningPoker-v1.pdf.

[15] N. C. Haugen, "An empirical study of using planning poker for user story estimation," in Agile 2006, Minneapolis, MN, 2006, p. 9 pages (electronic proceedings).

[16] J. Highsmith, Agile Software Development Ecosystems. Boston, MA: Addison-Wesley, 2002.

[17] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," in 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, British Columbia, Canada, 2004, pp. 132-136.

[18] C. R. Jakobsen and J. Sutherland, "Scrum and CMMI – Going from Good to Great," in Agile 2009, Chicago, IL, 2009, pp. 333 - 337

[19] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case Studies for Method and Tool Evaluation," IEEE Software, vol. 12, no. 4, pp. 52-62, July 1995.

[20] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary Guidelines for Empirical Research in Software Engineering," IEEE Transactions on Software Engineering, vol. 28, no. 8, pp. 721-733, August 2002 2002.

[21] H. Kniberg and R. Farhang, "Bootstrapping Scrum and XP under crisis," in Agile 2008, Toronto, Canada, 2008, pp. 436 - 444.

[22] T. Little, "Schedule Estimation and Uncertainty Surrounding the Cone of Uncertainty," IEEE Software, vol. 23, no. 3, pp. 48-54, 2006.

[23] E. M. Maximilien and L. Williams, "Assessing Test-driven Development at IBM," in International Conference of Software Engineering, Portland, OR, 2003, pp. 564-569.

[24] S. McConnell, Rapid Development: Taming Wild Software Schedules: Microsoft Press, 1996.

[25] K. Moløkken-Østvold and N. C. Haugen, "Combining Estimates with Planning Poker – An Empirical Study," in Australian Software Engineering Conference (ASWEC'07), Melbourne, Australia, 2007, pp. 349-358.

[26] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing Quality Improvement Through Test Driven Development: Results and Experiences of Four Industrial Teams," Empirical Software Engineering, vol. 13, no. 3, pp. 289-302, June 2008.

[27] C. Potts, "Software Engineering Research Revisited," IEEE Software, no. pp. 19-28, September 1993.

[28] J. Sanchez, L. Williams, and M. Maximilien, "A Longitudinal Study of the Test-driven Development Practice in Industry," in Agile 2007, Washington, DC, pp. 5-14.

[29] K. Schwaber and M. Beedle, Agile Software Development with SCRUM. Upper Saddle River, NJ: Prentice-Hall, 2002.

[30] Version One, "Second Annual Survey 2007 The State of Agile Development," 2007, http://www.versionone.com/pdf/StateOfAgileDevelopmet2_FullDataReport.pdf.

[31] Version One, "Third Annual Survey 2008 The State of Agile Development," 2008

http://www.versionone.com/pdf/3rdAnnualStateOfAgile_FullDataReport.pdf.

[32] L. Williams, G. Kudrjavets, and N. Nagappan, "On the Effectiveness of Unit Test Automation at Microsoft," in International Symposium on Software Reliability Engineering, Mysuru, India, 2009, pp. 81-89.

[33] L. Williams, L. Layman, and W. Krebs, "Extreme Programming Evaluation Framework for Object-Oriented Languages -- Version 1.4," North Carolina State University, Raleigh, NC Computer Science TR-2004-18 http://www.csc.ncsu.edu/research/tech/reports.php, 2004.

[34] M. V. Zelkowitz and D. R. Wallace, "Experimental Models for Validating Technology," IEEE Computer, vol. 31, no. 5, pp. 23-31, May 1998.