THE PERFORMANCE OF AGILE METHODS
COMPARISON TO TRADITIONAL DEVELOPMENT METHODS

ABSTRACT

The purpose of this research study is to perform a comparison and contrast analysis on three published longitudinal case studies to determine if there are significant and measurable differences in performance between software development projects using agile methods and projects using traditional methods. The research question is explored using four criteria: differences in development costs, schedule performance, quality, and stakeholder satisfaction. Findings from the studies indicate that for the three case studies analyzed, agile methods performed better than traditional methods in all four criteria. Despite these findings, there is insufficient evidence to conclude that agile methods are universally superior to traditional methods; additional research is needed to determine the edges of agile utility.

Keywords: adaptive, agile, agile manifesto, ASD, cost, Crystal, DSDM, extreme programming, FDD, incremental, iterative, lightweight programming, people-oriented, quality, safety, satisfaction, schedule, Scrum, SDLC, software development, XP

ACKNOWLEDGEMENTS

The inspiration for this project is due, in part, to a conversation I had concerning the ongoing disaster at Fukushima Dai-ichi in Japan, an event that history will likely record as the worst industrial accident of all time. It was observed that there is a lamentable tendency in mankind to focus on short-term rewards, to the exclusion of the “big picture.” Nowhere is this truer than in traditional business, where the focus is often primarily, and sometimes solely, on profit. In this small conversation lay the seeds of an idea, namely, that business methods that incorporate spiritual principles into a holistic view of the problem and its solutions would likely do a better job than traditional business methods focused solely on the triple constraints of budget, schedule, and scope.

In light of these principles, I would like to thank the following people: First, I thank Bill Wilson and Dr. Bob Smith, for writing down your observations even though they flew in the face of accepted wisdom. Not only do I owe you my life, but also your teachings provided a way for me to become happily and usefully whole. Next, I thank Dan Millman. At the time I was first exposed to your ideas, I really had no idea what you were talking about; all I knew was that it was important. Understanding came later. Next, I thank Neale Donald Walsch. You showed me where to look and how to put all of the pieces together. You returned to me the birthright I had unknowingly given away. Next, I thank the authors of the Agile Manifesto; so much wisdom encoded in 12 simple principles! Finally, and most importantly, I thank my wife, Laurie; your encouragement and your insights into “business with a soul” gave me the courage to attempt this endeavor.
# TABLE OF CONTENTS

Abstract ................................................................................................................................................. 1  
Acknowledgements ............................................................................................................................... 1 
Chapter 1: Introduction ......................................................................................................................... 3 
Chapter 2: Review of Related Literature ............................................................................................... 8 
Chapter 3: Research Findings ............................................................................................................ 21 
Chapter 4: Conclusions ....................................................................................................................... 33 
Appendixes ......................................................................................................................................... 38  
Appendix A ......................................................................................................................................... 38  
Appendix B ......................................................................................................................................... 39  
Appendix C ......................................................................................................................................... 40  
Appendix D ......................................................................................................................................... 44  
Appendix E ......................................................................................................................................... 45  
Appendix F ......................................................................................................................................... 46  
About the Author ................................................................................................................................. 47
CHAPTER 1: INTRODUCTION

CONTEXT OF THE PROBLEM
Computers and digitally controlled systems are a pervasive part of life in the early 21st century. From our alarm clocks to our power plants, every aspect of modern life is touched by computer systems. Generally, computer systems are complex constructs of hardware, software, and networks; hardware transforms data into useful information, software controls and directs the hardware and provides interfaces between systems, and networks provide input of data and output of information for the system. The primary role of software is to translate needs or requests into instructions that, when executed on the hardware, produce a result that satisfies the original need or request. (Burd, 2010, pp. 44 – 45)

Software is created to solve specific problems through the use of various software development methods. Many different approaches to developing software can be used, based on different Software Development Life Cycles [SDLCs]. Some approaches have been used since the advent of computer programming, while others are new and unique approaches to building systems that have emerged in response to development needs not satisfied by earlier methods. Generally, SDL Cs are classified on a continuum according to whether they are more predictive or adaptive; predictive approaches assume that a development plan can be created in advance of the actual development, and that the new software can be developed by following this plan. Adaptive approaches assume that a development plan created in advance of the actual development is of limited usefulness, because the exact development requirements are not yet known. The predictive approaches are more traditional and were developed from the 1970s to the 1990s. Many of the newer, adaptive approaches were developed during the 1990s and into the 21st century. (Satzinger, Jackson, & Burd, 2010, p. 39)

In February 2001, seventeen people dissatisfied with traditional software development approaches met at Snowbird ski resort in Utah; their goal, in addition to spending some time on the slopes, was to find a more effective way for developing software. The unanticipated outcome of this meeting was the “agile manifesto,” a new ideal for software development as opposed to traditional “heavyweight” document-driven processes. (Beck, Beedle, Bennekum, Cockburn, Cunningham, Fowler, . . . Thomas, 2001.)

The goal of agile development is to satisfy the customer by “early and continuous delivery of valuable software.” (Beck, et al., 2001) Additionally, a goal of agile development is to introduce complaisance into the development process; a flexible process enables customers to change or add requirements late in the development cycle. (Pfleeger & Atlee, 2010, p. 59)

The four tenets and 12 underlying principles forming the foundation of agile development are not new. (Abbas, Gravell, & Wills, 2008) However, integrating these principles into the various agile software development methods has shown that practical agile development, with its “agile thinking” focus on adaptability, iterative and incremental development, and orientation to people, is an effective tool for managing an essentially unpredictable activity, i.e., software development. (Abbas, et al., 2008, pp. 2 – 3)

STATEMENT OF THE PROBLEM
It is the contention of the author that for software development, traditional methods and agile methods are not simply two different ways to achieve the same goal. Because agile methods are based on principles that encourage cooperative and holistic problem analysis and solution synthesis, they tend to induce agile thinking into the development team. Agile thinking, with its adaptive, iterative, incremental, and people-oriented focus (Abbas, et al., 2008, p. 2), changes the way we look at problems. The ability to view problems from an agile point-of-view provides an effective basis for developing software. Effectiveness is beneficial to the software development business, because it creates a competitive advantage in an extremely demanding field. (Edwards, 2005) However, if agile
THE PERFORMANCE OF AGILE METHODS:
COMPARISON TO TRADITIONAL DEVELOPMENT METHODS

thinking does in fact create such a competitive advantage, it must be measurable and verifiable; the
problem of this study is whether there are significantly measurable differences in the performance of
traditional and agile methods when both are used to develop real-world software projects of
comparable size and complexity. Therefore, this paper will compare software development methods in
order to determine if measurable differences in performance actually exist, in the hope of proving or
disproving the contention that agile methods are inherently more effective methods for developing
software.

MAIN RESEARCH QUESTION AND SUB-QUESTIONS
The purpose of this research project is to compare agile software development methods to traditional
methods to determine if there is a measurable difference in performance between the two. Thus, the
major research question that this project will explore is: Are there measurable differences in
performance between agile development methods and traditional methods?

The first sub-question is: Are there significant and measurable differences in **cost performance**
between software development projects using traditional development methods, compared to software
development projects using agile development methods? This question is best explored using
quantitative methods. The specific method used to explore this sub-question is a comparison and
contrast analysis of longitudinal case studies between software development projects of similar size
and complexity; some use traditional methods and others use agile methods. Additional data from
other sources is incorporated with the case study data where relevant to the specific sub-question.
Comparison and contrast of costs between the projects provide insights into how the method used
affects development costs.

The second sub-question is: Are there significant and measurable differences in **schedule performance**
between projects using traditional development methods, compared to projects using agile development methods? This question is best explored using quantitative methods. The specific method used to explore this sub-question is a comparison and contrast analysis of the same case study material used for the first sub-question, along with data drawn from additional sources as it relates to the specific sub-question. Comparison and contrast of schedule performance between the projects will provide insights into how the method used affects the development schedule and completion of the projects.

The third sub-question is: Are there significant and measurable differences in **quality** between
projects using traditional development methods, compared to projects using agile development
methods? Because quality is difficult to define precisely, this question is explored using both
quantitative and qualitative methods. The specific method used to explore this sub-question is a
comparison and contrast analysis of the same case study material used for the first sub-question,
along with data drawn from additional sources as it relates to the specific sub-question. Comparison
and contrast of quality factors between the projects will provide insights into how the method used
affects project quality. Of special interest is a comparison of the projects with regard to how well they
minimize faults and mitigate risks. Useful software is inherently complex; the consensus within the
software development community is that it is practically impossible to remove all potential faults from
software, especially considering the fact that the software's operating environment is constantly
changing. Of necessity, the focus within the community has been on minimizing faults and mitigating
risk from faults.

The fourth sub-question is: Are there significant differences in **stakeholder satisfaction** between
projects using traditional development methods, compared to projects using agile development
methods? The term “stakeholder” refers to the people affected by the project, in this case, by the
software development project being examined. (Schwalbe, 2011, p. 10) For the purposes of this
paper, stakeholders will be divided into three groups for analysis: *Developers*, those involved with
actually creating the software developed by the specific development method; *management*, those
THE PERFORMANCE OF AGILE METHODS:
COMPARISON TO TRADITIONAL DEVELOPMENT METHODS

involved with managing and overseeing the development of the software created by the specific method; and customers, those who use the software developed by the specific method. Even more so than quality, stakeholder satisfaction is a subjective measure, so this question is best explored using qualitative methods. The specific method used to explore this sub-question is a comparison and contrast analysis of the same case study material used for the first sub-question, along with data drawn from additional sources as it relates to the specific sub-question. Comparison and contrast of stakeholder satisfaction between the projects as expressed within the respective studies will provide insights into how the method used affects the stakeholder experience.

SIGNIFICANCE OF THE STUDY

Traditionally, businesses are in business to advance the objectives of the business, and usually the traditional objective of business is to produce a profit. (Edwards, 2005) Experience with software development suggests that this focus on traditional business objectives may not be adequate in a world increasingly dependent on software products. Consider:

- The Therac-25 was a computer-controlled radiation therapy machine released on the market in 1983. Between 1985 and 1987, the Therac-25 massively overdosed six people, resulting in deaths and serious injury. (Leveson, 1995, p. 1) After months of investigation, the root causes of the accidents were traced to two different software faults that were introduced into the software when the software was designed for earlier versions of the Therac machine. These faults were not detected and corrected in either the earlier versions or the Therac-25 version prior to the market release; however, earlier versions did not suffer catastrophic failure, as these machines were more reliant on hardware safety interlocks that were not included in the Therac-25 design due to their expense. (Leveson, 1995, pp. 44 – 49)
- On December 20, 1995, American Airlines Flight 965 experienced a “controlled flight into terrain” while attempting to land at the Cali, Columbia, airport. The crash killed 159 of the 163 people aboard. Investigators concluded that a poorly designed user interface for the computerized flight management system was a contributing factor in the crash; the captain of the flight apparently thought he had correctly entered the code for the intended destination, Cali. Due to peculiarities of the flight management system, he had instead entered the code for Bogota, 132 miles in the opposite direction. As designed, the input error into the flight management system was practically impossible to detect, requiring the pilot to drill down through multiple screens in addition to flying the plane. The code for Bogota directed the plane toward a mountain, and there was insufficient time in the emergent situation for the flight crew to detect and correct the problem. (Ladkin, 1996, pp. 57 – 60)
- In August 2003, significant areas of North America suffered a blackout of electrical power; an estimated 50 million people were affected. Ultimately, the blackout was caused by a cascade of failures; one of the critical initial events was a race condition in the monitoring and reporting software used to manage power distribution. A race condition can occur when two or more processes attempt to access the same shared data. In this particular instance, the race condition caused the monitoring and reporting software to “hang,” i.e., it stopped responding to ongoing events. (Race condition, 2012) As a result of this failure, critical events failed to trigger associated alarms and operators were unaware that anything was amiss until the entire management system crashed. (U.S. – Canada Power System Outage Task Force, 2004, pp. 60 – 63) At least 11 deaths were attributed either directly or indirectly to the blackout. Combined cost estimates for recovery and opportunity lost due to the blackout range between $4 billion and $13 billion.

Clearly, software systems can have a great impact on human life, and on the quality of human life. In this light, it is ethically appropriate that an emphasis be placed on the holistic view whenever software solutions to problems are developed, especially when the software product is critical to safety. Generally, software can be divided into three criticality categories:
1. **Non-critical systems** are software systems whose failure does not affect lives, health, or business; an example is an office productivity tool, such as a word processor. In non-critical systems, failure is more properly an annoyance than a hazard.

2. **Business-critical systems** are systems whose failure would significantly affect business operations. The impact of a failure is described in terms of a loss of revenue or employee productivity.

3. **Safety-critical systems** are systems whose failure could significantly affect lives and health. With safety-critical systems, ethics demand that all potential safety aspects of the system be considered and incorporated into the final product, despite potential negative effects on business priorities; anything less is ethically unjustifiable. (Pfleeger & Atlee, 2010, pp. 413 – 415)

This study is significant, because it is important to determine the most effective way to develop software from both a business viewpoint and an ethical viewpoint. If a particular type of software development method shows evidence of being more effective at developing software products, then it is incumbent on us, as leaders in the field, to use it. Likewise, if it can be shown that a particular type of development method enables developers to more effectively evaluate the big picture and successfully mitigate potential failures, it is therefore ethically irresponsible to use anything else. Finally, if it can be shown that the use of a particular development method results in a more positive stakeholder experience, then the use of that method is preferred over other, less satisfactory methods.

Some advantages that may be realized as a result of this research include a better understanding of various development methods, understanding of how agile methods differ from traditional methods, and an understanding of if and why one development method is superior to another. If we have a clear understanding of why a particular method is more effective than another, we can use this understanding to help business succeed in its implementation. If we discover new methods to improve the development of successful and useful software products, we can integrate these methods into our operations, thereby reaping all of the rewards associated with increased efficiency. Finally, if we can discover ways to demonstrate the effectiveness of a particular development method, it is highly likely that all affected stakeholders can be brought into accord with the method, which greatly increases the likelihood of successful software development projects. Project management experience has shown that an effective development methodology is a critical factor in the success of a project, regardless of the specific methodology used. (Schwalbe, 2011, p. 81) If the potential discoveries described in this research are explored and a more effective method for developing software is identified, it represents a win-win situation; businesses will be able to operate using processes which if properly implemented, can produce high-quality software, happy developers, and methods that are better able to predict and prevent failures. If the potential discoveries described in this research are not explored, it is certain that software will continue to be developed using less effective methods, with continuing potential for inefficient processes, wasted resources, employee burnout, environmental damage, and loss of life.

**RESEARCH DESIGN AND METHODOLOGY**

The data is presented in the qualitative and quantitative formats listed in the respective research questions section, using secondary data gathered from publically available sources, including such data as project summaries and published expenditure statements, industrial averages, professional journals, preexisting interviews pertaining to software development methods, and Internet sources pertaining to the use of traditional and agile software development processes.

The first section of the research study compares agile software development methods to traditional development methods, primarily using longitudinal case studies between software development projects of similar size and complexity, with additional supporting data from other sources incorporated as relevant to the topic at hand. Comparison and contrast of costs between the projects provide insights into how the methods used affect development costs. The null-hypothesis of this
section is that there are no significant differences between development costs of traditionally developed software projects and projects developed using agile methods.

The second section of the research study compares agile software development methods to traditional development methods, using the same case study material as the first section along with additional data relevant to the problem at hand. Comparison and contrast of schedule performance between the projects provide insights into how the methods used affect the project schedule. The null-hypothesis of this section is that there are no significant differences between the schedule performance of traditionally developed software projects and projects developed using agile methods.

The third section of the research study compares agile software development methods to traditional development methods, using the same case study material as the first section along with additional data relevant to the problem at hand. Comparison and contrast of product quality between the projects will provide insights into how the methods used affect product quality. The null-hypothesis of this section is that there are no significant differences between product quality of traditionally developed software projects and projects developed using agile methods.

The fourth section of the research study compares agile software development methods to traditional development methods, using the same case study material as the first section along with additional data relevant to the problem at hand. Comparison and contrast of stakeholder satisfaction between the projects will provide insights into how the methods used affect stakeholder satisfaction. The null-hypothesis of this section is that there are no significant differences between stakeholder satisfaction of traditionally developed software projects and projects developed using agile methods.

Each section of the research study targets a specific research sub-problem. The data in each section, after exploration, should provide evidence either for or against the associated null-hypothesis. Taken together, the data should be able to illuminate the central research question; specifically, are there significant measurable differences in performance between agile development methods and traditional development methods?

ORGANIZATION OF THE STUDY
Chapter one, Introduction, includes the context of the problem, the statement of the problem, research questions, the significance of the study, research design and methodology, and the organization of the study. Chapter one includes all of the elements required for a Directed Research Project proposal, as delineated and required by Strayer University. (Strayer University, 2009)

Chapter two, Review of the Literature, provides an overview and summary of the literature used in the research project, including government statistics, company presentations and data, and articles from peer-reviewed managerial, professional, and business journals. It explores both agile and traditional development methods, and introduces the case studies used in the research chapters to explore the research questions.

Chapter three, Research Findings, introduces the comparison framework, presents the research findings, and compares the performance of agile projects and traditional projects using development costs, schedule performance, product quality, and stakeholder satisfaction as the comparison factors.

Chapter four, Conclusion, interprets the data covered in the research project, and summarizes findings and conclusions as they relate to the research problems and sub-problems. (Leedy & Ormond, 2010, p. 297) This chapter will determine if the research answers the research questions and will explore the answers provided. It also presents suggested directions for future research, in order to increase the body of knowledge concerning agile and traditional development performance.
CHAPTER 2: REVIEW OF RELATED LITERATURE

INTRODUCTION TO THE REVIEW
To provide a framework for understanding the research that follows, the chapter begins by examining the origins of software development methods, and how they have evolved as software development experience has winnowed viable methods from methods that present too many drawbacks to be practically useful. It then explores the origins and history of agile development, explores the differences between agile and traditional software development methods, describes the different types of agile methods that have emerged in response to the agile movement, and describes the reaction of the software development community to the advent of agile methods. Finally, the chapter presents the case studies used to explore the research questions.

HISTORY OF SOFTWARE DEVELOPMENT METHODS
Concurrent with the advent of computers and digitally controlled systems was the birth of software development; after all, computers are not useful without software to control their functioning. (Burd, 2010, p. 44) The first software was developed using ad hoc software development methods, also known as “code and fix.” (Fowler, 2005, p. 2) In other words, the developer would develop the solution to the problem as the code was being written, without any overarching plan. This method is effective for developing solutions to simple problems, but quickly breaks down once a certain level of complexity is reached. The result of attempting to develop software solutions to complex problems using this method is often a “big ball of mud;” code components are highly coupled, lack cohesion, and are difficult or impossible to trace, troubleshoot, and modify. (Foote & Yoder, 1999, p. 2)

In response to the inadequacy of ad hoc development, engineering, or plan-driven, software methodologies were developed; these methods attempted to impose the same type of discipline as that required for engineering projects, such as building a bridge, upon the software development process. (Fowler, 2005, p. 3) For the purposes of this paper, plan-driven methodologies will be referred to herein as “traditional” methods of software development.

The first of the traditional development methods was the “Waterfall” method. With the Waterfall method, the development process is divided into a series of sequential steps: Scope, requirements analysis, design, development, test, implementation, and operation & maintenance. The foundation of all the traditional software development methods is the Waterfall. (Dillman, 2003, p. 3)

![Waterfall Method Diagram](image)

**Figure 1. The Waterfall Method (Dillman, 2003, p. 3)**

An assumption of the Waterfall method is that the entire software development project can be planned at the beginning of development, and that each phase begins where the previous phase ends; a result of this assumption is that the Waterfall method does not support changes to the software requirements. (Dillman, 2003, p. 11) As change is inevitable in any practical project, the Waterfall method is not considered to be practically useful in its pure form.
An improvement to the Waterfall method is the “V” method. The assumption of the V method is that testing activities performed after coding can be used to validate the correctness of the planning activities performed prior to coding. Any problems found during testing can be resolved by repeating the linked preceding activity; thus problems found during unit and integration testing can be resolved by returning to the program design phase, and so forth. The V method acknowledges that change will occur during software development, and provides feedback paths to make change possible. (Pfleeger & Atlee, 2010, pp. 52 – 53)

Figure 2. The V Method (Pfleeger & Atlee, 2010, p. 52)

The “Incremental” method is a variation of the Waterfall method, designed to minimize some of its drawbacks. With the Incremental method, development is divided into a series of increments, which can then be developed in parallel. The focus of each increment is in developing only a subset of the desired requirements; the process is then repeated until all of the desired functionality is in place. While this method breaks development up into more manageable “chunks,” it still relies upon the ability to analyze and design the entire system at the beginning of the project and, therefore, has the same problems with incorporating change as the Waterfall method. (Dillman, 2003, p. 3)
The "Evolutionary/Iterative" method is another attempt to modify the Waterfall method to make it more practical. Like the incremental approach, the Evolutionary/Iterative method develops only a subset of the solution to the problem. Unlike the Incremental approach, an iteration of the solution begins with the requirement analysis for that particular subset, and continues with the subset of the requirements throughout the iteration. The set of phases from requirements analysis through implementation is repeated until the complete solution is in place. (Dillman, 2003, p. 4)
While all of these methods can be used to develop complex software, they all share a fundamental assumption that limits their usefulness in developing complex software to solve practical problems: namely, that software can be developed using methods that were initially derived from engineering methods. Fowler (2005, pp. 4 – 7) argues that software development is fundamentally a different activity from engineering activity, and therefore requires a different type of methodology to be effective.

ORIGINS AND HISTORY OF AGILE DEVELOPMENT

In response to a common dissatisfaction with traditional software development methods, the authors of the agile manifesto recorded and presented to the world a new approach to software development. At the core, just as traditional software development methods are an attempt to impose discipline onto the development process, agile methods are an attempt to impose ethical values onto the development process. Agile methods recognize that the purpose of software is to solve a problem, and that the desired solution is often dynamic rather than static. Agile methods try to provide an optimal level of process for the intended solution, balancing between not enough process and the tendency of the software to devolve into a big ball of mud, and too much process and the tendency of the process to become rigid and resistant to change. (Fowler, 2005, p. 3)

No one was more surprised at the success of the meeting than the authors of the manifesto. According to Alistair Cockburn, many of the participants did not expect anything substantially useful to come out of the meeting; “I personally didn’t expect that this particular group of agilites to ever agree on anything substantive.” (Beck, et al., 2001.) His initial skepticism was replaced by a belief in the importance of the manifesto, a belief that was apparently shared among the authors: “Speaking for myself, I am delighted by the final phrasing [of the manifesto]. I was surprised that the others appeared equally delighted by the final phrasing. So we did agree on something substantive.” (Beck, et al., 2001.)

The principles underlying agile methods are not new. There is evidence that some of the ways in which agility is imposed on software development were being used as early as 1957. (Abbas, et al., 2008, p. 2) Many of the methods that are now considered agile have been suggested throughout the history of software development as improvements over the traditional methods available at the time. (Abbas, et al., 2008, pp. 4 – 6) What is new is that the principles and tenets are now amassed into a paradigm on which specific software development methods are based.

There is a deeper theme underlying agile methods; at the close of the meeting that resulted in the manifesto, one of the authors joked that he was “about to make a mushy statement.” (Beck, et al., 2001) Another of the authors made the following observation:
At the core, I believe Agile Methodologists are really about “mushy” stuff about delivering good products to customers by operating in an environment that does more than talk about “people as our most important asset” but actually “acts” as if people were the most important, and lose the word “asset”. So in the final analysis, the meteoric rise of interest in and sometimes tremendous criticism of Agile Methodologies is about the mushy stuff of values and culture. (Beck, et al., 2001)

Generically, the agile development process replaces the scope, requirements analysis, and design phases and their associated documentation with user stories, scenarios, and use cases. Development and testing are integrated, and placed into incremental “time boxes,” which are time periods of a specific predetermined length; once the length of a time box is set, it is locked and cannot be changed. The requirements are prioritized, and functionality is implemented in each incremental time box in priority order. Multiple time boxes are used to support parallel development, and iterative development occurs in response to the functionality developed in each time box; the next iteration begins with the next priority requirement. Finally, throughout the process, previously made design decisions are reviewed and evaluated in order to simplify the associated code and/or adapt to evolving needs; this is known as “refactoring.” (Pfleeger & Atlee, 2010, p. 295)

Figure 6. A Generic Agile Development Method (Dillman, 2003, p. 6)

Differences Between Traditional and Agile Methods

A definitive difference between the two approaches is that traditional methods assume that software development is a defined [i.e., linear] process; in other words, a process is delineated and, if allowed to run to completion, produces the same results each time it is run. On the other hand, the assumption behind agile methods is that software development is an empirical or non-linear process, where the actual process changes each time it is run in response to the conditions encountered during the run. (Williams & Cockburn, 2003, p. 40) The differences between traditional methods and agile methods are well illustrated through the four tenets of the manifesto:

1. **Agile development values individuals and interactions over processes and tools.** With agile methods, people and communication are emphasized, whereas with traditional methods the development process and development tools are emphasized.

2. **Agile development focuses on producing working software rather than in producing comprehensive documentation.** With agile development, the emphasis is on solving the problem, whereas, with traditional methods, the emphasis is on documenting the problem and the intended solution.
3. **Agile development focuses on customer collaboration rather than contract negotiation.** In agile development, the customer is an active participant in the development process, whereas, with traditional methods, customers are usually only involved with the initial requirements-gathering phase; their next involvement is usually at delivery, where they may [or may not] be presented with a solution to the actual problem.

4. **Agile development focuses on responding to change rather than on creating and following a plan.** Agile methods are adaptive to changes to the problem and its environment, whereas traditional methods are predictive; they assume that the problem and its environment can be analyzed sufficiently during initial planning efforts. (Beck, et al., 2001)

An analogy that illustrates the differences between traditional methods and agile methods is a road trip. Traditional methods assume that it is possible to sit down and plan the entire route to the destination by studying roads and distances on a map. Traditional methods have a hard time adapting, however, to the discovery that a road that was planned to be used in the middle of the route is closed. Agile methods, on the other hand, acknowledge that they don’t really know precisely how they are going to get to the destination; instead, they start moving in the general direction of the destination immediately. Agile methods depend on frequent stops, where feedback is gathered and digested as to the current location, and the direction needed to continue toward the destination. As such, agile methods tend to be more thoughtful processes than traditional methods, since they require constant feedback and evaluation. This constant examination is the mechanism which brings about agile thinking. (Williams & Cockburn, 2003, p. 2)

In addition to the four tenets described above, the agile manifesto provides 12 principles to be used as a foundation for software development. [See Appendix A]

**DIFFERENTIATION OF AGILE DEVELOPMENT PROCESSES**

There are several different implementations of agile development principles into specific processes. While they are all based upon the principles set forth in the agile manifesto, they differ in the ways in which these principles are placed into practice. This paper explores the following specific agile processes:

- **Extreme Programming**, also known as XP. This process was founded by Kent Beck in 1996, during the Chrysler payroll development project. XP has been proven to be a successful agile development process for many software projects of different sizes and in different industries. XP stresses customer satisfaction; instead of delivering all the functionality of the system at some future date, the process delivers the software needed as it’s needed. XP empowers developers to respond to changing customer requirements, even late in the development cycle. XP emphasizes teamwork; managers, customers, and developers are all equal partners in a collaborative team. XP creates a simple yet effective environment, enabling teams to become highly productive. Teams self-organize around the problem to solve it as efficiently as possible. XP is based on the following values:
  - **Simplicity**: Do what is needed and requested, but no more. This maximizes the value added for the investment made. Take small simple steps toward the goal and mitigate failures as they happen. Create something to be proud of and maintain it over the course of its useful life for reasonable costs.
  - **Feedback**: Take every iteration commitment seriously by delivering working software. Demonstrate the software early and often; then listen carefully and make any changes needed. Talk about the project and adapt the process to it, not the other way around.
  - **Respect**: Everyone gives and feels the respect they deserve as a valued team member. Everyone contributes value even if it is simply enthusiasm. Developers respect the expertise of the customers and vice versa. Management respects the right to accept responsibility and receive authority over one’s own work.
• **Courage**: Tell the truth about progress and estimates. Do not document excuses for failure but plan to succeed. Do not fear anything because no one ever works alone. Adapt to changes whenever they occur.

• **Communication**: Everyone is part of the team; communicate face to face daily. Work together on everything from requirements to code. Create the best solution to the problem possible, together. (Wells, 2009)

![Extreme Programming Project](https://example.com/figure7)

**Figure 7. The Extreme Programming Process (Wells, 2009)**

• **Scrum**. The “scrum” framework is an agile software development process that functions as a wrapper for existing engineering-based development processes to iteratively and incrementally develop software. The name comes from the game of rugby, where a scrum is used to return an out-of-play ball to the playing field through teamwork. It is currently the most popular of the agile development processes. (Williams, Brown, Meltzer, & Nagappan, 2011, p. 1) The Scrum process is as follows: The product owner creates the requirements, prioritizes them, and documents them into a list; this list is now referred to as the product backlog. In Scrum, requirements are called features. Scrum teams work in short iterations, often between 2 weeks and 30 days long. In Scrum, the current iteration is called a “sprint.” A sprint planning meeting is held with the Scrum development team: testers, management, the project manager, and the product owner. In the sprint planning meeting, this group chooses which features from the product backlog are to be included in the next iteration, driven by highest business value, risk and the capacity of the team. Once a sprint begins, it is locked; at that point, features cannot be added to the current sprint. Daily 10-to-15 minute Scrum meetings are held. While these meetings are open, only members of the Scrum team are allowed to actively participate; all others may only observe. 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 the way of you doing your work?” (Williams, Brown, et al., 2011, p. 2) 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 for the next one. (Williams, et al., 2011, pp. 1 – 2)
The Crystal Family of Methodologies. Crystal was founded by Alistair Cockburn, one of the authors of the agile manifesto. He developed the Crystal family of software development processes as a group of approaches tailored to fit different size teams. Crystal is seen as a family because of the belief that different approaches are needed, depending on the size of the development team and the criticality of the application.

Each color in the Crystal family denotes its "heaviness;" the darker the color, the heavier or more plan-driven the method. The character symbols in the illustration represent the potential loss due to a system failure: Comfort [C], Discretionary money [D], Essential money [E], and Life [L]. Currently, only
Crystal Clear and Crystal Orange are realized. All Crystal approaches share common features; they all have three priorities: safety [in project outcome], efficiency, and habitability [developers can live with the process]. They also share common properties, of which the most important are: frequent delivery, reflective improvement, and close communication. The focus on habitability is an important part of the Crystal mindset. The overarching goal of Crystal is to put into place the smallest effective amount of process possible. As a result, Crystal requires less discipline than XP, trading off efficiency for greater habitability and reduced chance of failure. (Fowler, 2005, pp. 23 – 24)

Dynamic Systems Development Method. DSDM is a development process based on agile principles, including iterative and incremental development and continuous user/customer involvement. DSDM consists of five phases: [1] feasibility study, [2] business study, [3] functional model iteration, [4] design and build iteration, and [5] implementation. DSDM incorporates iterations through time boxes; a time box lasts for a predetermined period of time, and the iteration ends with the expiration of the time box. DSDM fixes cost, quality, and time at the outset and uses prioritization of scope into musts, shoulds, coulds, and won’t haves to adjust the project deliverable to meet the fixed business constraints. There are nine underlying principles of DSDM:

1. Active user involvement is imperative.
2. Teams are empowered to make decisions.
3. The focus is on frequent delivery of products.
4. Fitness for business purposes is the essential criterion for acceptance of deliverables.
5. Iterative and incremental development is necessary in order to converge on an accurate business solution.
6. All changes during development are reversible.
7. Requirements are baseline at a high level.
8. Testing is integrated throughout the life-cycle.
9. A collaborative and cooperative approach shared by all stakeholders is essential. (Abrahamsson, et al., 2002, pp. 61 – 64)

![Figure 11. The DSDM Process (Abrahamsson, et al., 2002, p. 62)](image)

- **Feature-Driven Development.** FDD is an agile, highly adaptive software development process that is based on multiple short iterations. FDD emphasizes quality at all steps; delivers frequent, tangible working results; and provides accurate and meaningful progress and status information with the minimum of overhead and disruption for the developers. FDD is composed of five phases: [1] Develop an overall model, [2] build a feature list, [3] plan by feature, [4] design by feature, and [5] build by feature.

  FDD decomposes the entire problem domain into tiny problems, called features, which can be solved in a small period of time, usually two weeks. A feature is a small, client-valued function that can be implemented in the time allotted. In a business system, a feature maps to a step in an activity within a business process. Any function deemed too complex to be implemented within two weeks is further decomposed into smaller functions until each sub-problem is small enough to be called a feature. Features are then developed independently. Unlike some other agile methods, FDD claims to be suitable for the development of critical systems. (Abrahamsson, et al., 2002, pp. 47 – 49)
Adaptive Software Development. ASD is an agile software development process that grew out of rapid application development work done by James A. Highsmith III and Sam Bayer. The central principle of ASD is that continuous adaptation of the process to the work at hand is the normal state of affairs. ASD replaces the traditional waterfall cycle with a repeating series of phases: speculate, collaborate, and learn. The ASD process is mission focused, feature based, iterative, time boxed, risk driven, and change tolerant. The phases are named as they are to emphasize the role of change throughout the process. The word "speculate" is used rather than plan, because of the common assumption that a deviation from the plan is a failure. Likewise, "collaboration" refers to the teamwork necessary to effectively develop software under changing circumstances. The word "learning" acknowledges that mistakes are inevitable when developing software in a rapidly changing environment; the key is to learn from correcting the mistakes, which leads to greater experience and eventually mastery over the problem domain. (Abrahamsson, et al., 2002, pp. 68 – 71)
This is by no means a comprehensive list of all of the existing agile development processes. Some of the additional processes that are considered part of the agile development family include the Rational Unified Process [RUP], Agile Modeling, Pragmatic Programming, and Context Driven Planning, to name a few. Some of these, such as RUP, are more properly considered to be development frameworks that can be imposed upon any software development method regardless of whether it is agile or traditional. Others, such as Agile Modeling, are not stand-alone processes, but can be used within a complete process. These processes and other agile processes that have not been mentioned are outside of the scope of this paper. [See Appendix B for a table showing comparative features of the agile software development processes described.]

REACTION OF THE SOFTWARE DEVELOPMENT COMMUNITY
The reaction of the software development community to the introduction of agile methods has been mixed. Proponents of agile methods have strenuously and vociferously promoted agile methods as a panacea for all software development problem domains. Opponents have just as strenuously argued that a structured and planned methodology is necessary to successfully develop software, especially for safety-critical systems. There is quite a bit of anecdotal evidence that agile methods are effective in certain situations; as experience with agile methods has grown, a body of evidence has slowly emerged as to its practical effectiveness. By 2001, there was enough practical experience with agile development to conclude that agile development works, given a relatively narrow set of circumstantial parameters; specifically, agile processes can be said to work where the project is non safety-critical, has volatile requirements, and is developed by small, skilled teams located near enough to one another that rapid and consistent communication is possible. Although the agile value set can be adapted to work under different circumstances, experience to date seems to be defining the boundary conditions for truly agile behavior to this parameter set. (Williams & Cockburn, 2011, p. 40)

INTRODUCTION TO THE CASE STUDIES
There are three related case studies supporting this research; the first is a longitudinal case study performed by Layman, Williams, & Cunningham (2004a.) This case study evaluates the effects of adopting the Extreme Programming [XP] methodology in an industrial environment [i.e., Sabre Airline Solutions]. The case study compares two releases of the same product. The first release was developed using traditional development methods, immediately prior to the team’s adoption of XP methodology. The second release was completed after approximately two years of XP use. The second case study is a similar study, done by the same research team in the same industrial environment. This study compares the release of a larger project developed using XP to published industry averages. (Layman, Williams, & Cunningham 2004b) The third case study compares the second and third releases of a Servlet/XML application developed by a seven-person team at IBM for a toolkit that is used to create products for external customers. (Williams, Krebs, et al., 2004, p. 2.) The team used a subset of XP practices deemed “safe” and appropriate for their team culture and project characteristics. In the second release, the team began their initial adoption of XP practices. The team then expanded their XP adoption in the third release. Supporting these three studies is an evaluation framework developed by Williams, Krebs, & Layman (17 June, 2004) which provides a benchmark measurement method to assess how thoroughly and how effectively an organization has adopted XP practices. Additional research data are drawn from Hanssen & Faegri (2006), which presents data concerning customer engagement for both traditional development methods and agile methods, as experienced by a small software product company. Additional empirical data are provided by Lindvall, Basili, Boehm, Costa, Dangle, Shull, ... Zelkowitz (2002.) An empirical case study concerning stakeholder satisfaction within agile development projects is presented by Ferreira & Cohen (2008.) Data concerning project failure within software development projects of all methods are presented in a case study by Linberg (1999.) These data are useful in determining the effects of software development methods on product quality. Additional data are drawn from a case study presented by Williams, Brown, et al. (2011), which describes the experiences of three development teams implementing Scrum at Microsoft. A method for determining the optimal development method
CHAPTER 3: RESEARCH FINDINGS

INTRODUCTION TO THE RESEARCH
To provide a framework for drawing useful conclusions from the case study data that follows, the chapter begins with an explanation of the XP evaluation framework. Next the chapter presents findings gleaned from the case studies used as the foundation for this paper. Finally, the selected data are compared and evaluated in terms of their impact on development costs, development schedule, quality, and stakeholder satisfaction.

THE XP EVALUATION FRAMEWORK
One of the reasons that this particular set of case studies was chosen from the relevant literature is the existence of an evaluation framework used throughout the studies to organize the data. This framework, the XP evaluation framework, version 1.4 (Williams, Krebs, & Layman, 2004), is useful because it helps to normalize the inevitable differences in particulars between the compared projects. Case studies are useful for research, because they explore the problem in real-world conditions with all of the complexity inherent therein, whereas experiments are usually structured so as to explore the problem in a strictly controlled environment. One problem that is caused by this complexity is that it then becomes difficult to draw conclusions from the results of the case studies, because the inevitable differences induced by real-world conditions introduce confounding variables into the results. This difficulty is exacerbated when exploring empirical processes. Because empirical processes are usually never executed in precisely the same way, making comparisons between results of case studies examining empirical processes leads to questions of internal validity; did the changes we observe occur because of deliberate changes to the process, or did they occur due to incidental changes to the process caused by real-world conditions, which were not the intended subject of the study? The goal of the evaluation framework is to minimize the impact of these confounding factors on the results of the studies, in order to make the conclusions drawn from them more practically useful.

The XP evaluation framework [XP-EF] is composed of three parts: XP Context Factors [XP-cf], XP Adherence Metrics [XP-am] and XP Outcome Measures [XP-om].

![Extreme Programming Evaluation Framework](image)

Figure 14. The XP Evaluation Framework (Williams, Krebs, & Layman, 2004, p. 2)

The goal of the XP-cf component is to quantify the ways in which software development projects can differ. XP context factors are organized into seven categories:

1. **Software classification**: Software that is developed to solve a particular type of problem often has different characteristics from software that is developed to solve another type of problem, e.g., office productivity software has different characteristics compared to software designed to supervise an industrial process. This category captures and assigns a weight to the type of development project being evaluated.
2. **Sociological**: Captures and assign a weight to personnel factors of the development team. Factors such as level of education, experience with the problem domain, team size, and attrition rate can affect the outcome of a project.

3. **Project-specific**: Captures particulars of the specific project being evaluated, e.g., project size, project budget, and project schedule.

4. **Ergonomic**: Captures and assign weights to the specific development environment. For example, agile methods stress collaboration, which may be impeded by a development environment consisting of private cubicles and offices.

5. **Technological**: Captures and assign weights to differing processes and tools used during development. Information regarding specific fault prevention methods, reusable code libraries, and project management factors are recorded here.

6. **Geographical**: Captures and assign weights to geographical factors affecting the development process. For example, agile methods usually stress face-to-face communication and pair programming, which is the practice of having two developers use one computer, where one produces code and the other checks the code for faults; these aspects of agile development may be hindered by distributed teams or language barriers.

7. **Developmental**: Captures the natural development method for the project under evaluation. Based on the work of Boehm and Turner (2003) this category plots five project factors: [1] team size, [2] criticality, [3] personnel understanding, [4] dynamism, and [5] culture onto a polar chart, as shown in figure 15. When the five data points are connected, the resulting shape indicates the project’s optimal method. Shapes that cluster near the center of the graph are suggestive of an agile development method, while shapes that cluster toward the edges of the graph are suggestive of a plan-driven method. Shapes where some portions cluster toward the edges while other portions cluster near the center suggest the usefulness of a hybrid method containing aspects of both traditional and agile methods. (Williams, Krebs, & Layman, 2004, pp. 2 – 15)

---

**Figure 15. Example Developmental Factors Polar Chart**
The goal of the XP-am component is to quantify the thoroughness in which agile practices are implemented within the specific software development project under evaluation. It is not unusual for software projects using agile methods to implement only a subset of agile practices. Another common situation is that a project implements all recognized agile practices, but does not implement them consistently. The XP-am component is divided into two components: objective measures such as test lines-of-code/source lines-of-code, iteration length, and pairing frequency, and subjective measures such as surveys' responses to questions concerning the perceived frequency of use of agile practices during the project under evaluation. (Williams, Krebs, & Layman, 2004, pp. 15 – 19) The Shodan Adherence Survey [see Appendix C] is an anonymous survey used within the XP-EF to gather XP adherence information from team members. (Boehm & Turner, 2003) Additionally, the XP-EF uses semi-structured interviews of the development team members to elicit supplemental qualitative data related to the project under evaluation.

The goal of the XP-om component is to quantify the outcome of the specific software development project under evaluation. The XP-om asks five questions about the outcome of the development project being evaluated:

1. Does the pre-release quality improve when a team uses XP practices?
2. Does the post-release quality improve when a team uses XP practices?
3. Does programmer productivity improve when a team uses XP practices?
4. Does customer satisfaction improve when a team uses XP practices?
5. Does team morale improve when a team uses XP practices?

Of these measures, both pre- and post-release quality and programmer productivity are quantitative, based on the quantification of the data throughout the evaluation framework. Customer satisfaction and team morale are qualitative measures, based on data gathered during interviews with the development team. (Williams, Krebs, & Layman, 2004, pp. 19 – 22)

FINDINGS FROM THE CASE STUDIES

The combined XP-om results for the studies are included in Appendix D. The XP-am results for the three central case studies are combined and presented in Appendix E.

The first of the three central studies took place between 2001 and 2004 as a cooperative research effort between Sabre Airline Solutions and North Carolina State University. The research team studied the third and ninth releases of a scriptable graphical user interface environment used by external customers to develop customized end-user business software. The third release, hereafter referred to as “Sabre-A Old,” was developed by a 10-person team over an 18-month period beginning in 2001. The ninth release, hereafter referred to as “Sabre-A New,” was developed by a 10-person team consisting of some of the original team members from Sabre-A Old, beginning in 2003. The Sabre-A Old release was developed using a Waterfall software development method, whereas the Sabre-A New release was developed using XP. In the 18-month period between the releases, the development team became first familiar with, and then proficient in, XP practices. The “A” suffix is used to identify these projects because both the project size and team size place the projects within the category of projects that are considered characteristically agile. (Layman, et al., 2004a., pp. 1 – 3)

The second of the three central studies took place between 2003 and 2004 as a cooperative research effort between Sabre Airline Solutions and North Carolina State University. The research team studied the 13th release of a large web application that was combined with a back-end batch component; the combined project consisted of over a million lines of executable code. Development on this release lasted for approximately five months. The team for this project consisted of 15 developers, one dedicated tester, and several specialists. This release will hereafter be referred to as “Sabre-P.” At the time of this project, the development team had approximately 20 months of experience in the XP
practices used. The results from this project are compared to two sets of published industry averages: industry averages documented by Capers Jones, and the Bangalore SPIN group. (Layman, et al., 2004b, p. 8) These two sources were chosen because of their availability, and also because the data contained therein is similar enough to the XP-EF format to allow meaningful comparison. The “P” suffix was used to identify this project because the project size and team size placed the project within the category of projects that are considered characteristically plan-driven. (Layman, et al., 2004b, pp. 1 – 3)

The last of the three central studies took place between 2001 and 2004 as a year-long cooperative research effort between IBM and North Carolina State University. The research team studied the second and third releases of a proprietary software component developed under contract to another IBM organization, where the component eventually became part of a software package marketed to external customers. The second release, hereafter referred to as “IBM Old,” was developed by an 11-person team. The third release, hereafter referred to as “IBM New,” was developed by a seven-person team consisting of some of the original team members from IBM Old. The IBM Old release was developed using a Waterfall software development method with some informal small-team practices that resemble XP. The IBM New release was developed using a “safe” subset of XP practices that was appropriate for the corporate culture in place at the time of the study. The results of the developmental evaluation of these projects place them within the category of projects that are best served by using a hybrid process, containing both plan-driven and agile processes. (Williams, Krebs, et al., 2004, pp. 1 – 5)

According to the original published studies, The Sabre A team experienced a 65% improvement in pre-release quality, a 35% improvement in post-release quality and a 50% improvement in productivity as a result of the implementation of XP practices. (Layman, et al., 2004a, p. 8) The Sabre P team reported an improvement in total defect density, similar pre-release quality, and similar productivity when compared to industry averages compiled by the Bangalore SPIN group. When compared to industry averages compiled by Jones, the Sabre P team reported an improvement in total defect density, similar pre-release quality, and higher productivity as a result of the implementation of XP practices, although defect removal efficiency fell. (Layman, et al., 2004b, p. 8) The IBM team reported that pre-release quality doubled and post-release quality improved by 39%. Productivity was measured three ways: as a function of lines of code, as a function of user stories, and through the calculation of the Putnam Productivity Parameter. (Putnam & Myers, 1992) Employing user stories, productivity improved by 34%. Using lines of code, productivity improved by 70%. Using the Putnam Productivity calculation, productivity improved by 91% as a result of the implementation of XP practices. (Williams, Krebs, et al., 2004, p. 8)

COST COMPARISONS

The performance metric most directly affecting development costs for a software development project is productivity. Simply put, productivity is a measure of efficiency; how much useful output is produced by a process in proportion to the inputs to the process. If a process can produce more output in relation to the same level of input, or if it can produce the same level of output with less input, then it is said the be more productive. An input into a process represents a business cost, whether it is raw materials or developer labor; anything that more effectively uses these assets reduces costs. (Farnham, 2010) Dillman (2003) points out that the focus of the business world is to do things better, faster, and cheaper, which corresponds to the traditional project management “triple constraint” of scope, schedule and cost. (Schwalbe, 2011, p. 8) It is important to recognize that these three facets are interrelated; for any project, it is possible to alter two of these facets, with an induced change in the third being the cost of altering the first two. All else being equal, “if you want it faster and cheaper, it will cost you quality. If you want it faster and better, it will cost you money. If you want it cheaper and better it will cost you time.” (Dillman, 2003, p. 6)
Productivity for the cases studied is measured as a function of thousands of lines of executable code [KLOEC] divided by person-months [PM]. The person-month is a measure of the effort needed to complete a task, e.g., if it takes two people three months to complete a task, the person-month metric for the task is equal to the product of 2 and 3, i.e., 6.

One concern to be aware of in reference to the KLOEC metric is that programming languages have different levels of instruction explosion. Currently, digital computers only “understand” machine languages based on the ones and zeroes of binary. Programming languages are used to translate between the natural languages of humans and machine languages of computers; the problem is that different programming languages translate between human languages and machine languages with differing levels of efficiency. For example, code for a database lookup may take 20 lines of executable code when written in the programming language “C” whereas the same problem can be solved by using a five line statement in the database query language SQL. (Burd, 2010, pp. 273 – 275)

Regardless, KLOEC is a convenient metric for comparing software development projects because it is readily available; it is often provided by automated testing tools.

Additional methods used for calculating productivity in the cases studied include user stories/person-month, function points/person-month, and the Putnam Productivity Parameter. Both user stories and function points attempt to uncouple the measurement from any peculiarities of the programming language used by focusing on the implemented functionality. User stories are an agile approach for capturing requirements using informal communication. Each discrete function needed by a system has an associated user story. (Ho, et al., 2006, p. 3) The function points method categorizes each discrete function and assigns a weight to it based upon past development performance. The cost of one function point is then calculated, and the results used to manage the development effort. (Boehm, Abts, Brown, Chulani, Clark, Horowitz, . . . Steece, 2000, pp. 15 – 17) The Putnam Performance Parameter is used to normalize differences in the software development environment, based on observation of a dozen large software development projects. [See Appendix D for more information on the Putnam Productivity Parameter.]

Some additional areas that can impact development costs are fault density and ergonomic factors. Fault density is measured in terms of total defects [TD] divided by thousands of lines of executable code [KLOEC]. Defects represent wasted effort; a process that decreases fault density reduces development costs. Ergonomic factors relate to the development environment. In regard to the cases studied, XP has an inherent advantage compared to traditional methods. The preference for XP labs is an open development lab, compared to the private or semi-private cubicles preferred by traditional development; the use of an open development lab can result in lower facilities costs due to the more efficient use of floor space. Likewise, the emphasis on pair programming can result in lower capital equipment costs, since each team member does not require a dedicated desktop computer.

The Sabre-A study experienced an increase in productivity and a decrease in fault density when XP methods replaced traditional methods. (Layman, et al., 2004a., p. 8) The Sabre-P study experienced a decrease in fault density when XP methods were used, compared to published industry averages. Productivity was measured using two different methods: KLOEC/person-month and function points/person month. Using KLOEC/person-month, productivity measured using XP methods was similar to published industry averages. Using function points/person-month, productivity measured using XP methods was higher than published industry averages. (Layman, et al., 2004b., p. 8) The IBM study experienced an increase in productivity using all three measurement methods, and a decrease in fault density when XP methods replaced traditional methods. (Williams, Krebs, et al., 2004, p. 8) In light of these results, the null-hypothesis that there are no significant differences between development costs of traditionally developed software projects and projects developed using agile methods is not supported by the evidence presented.
SCHEDULE COMPARISONS

Schedule performance refers to how well a project progresses compared to the original estimate of the amount of time needed to finish the project. It is important to remember that the original schedule is by necessity an estimate—an educated guess as to the amount of time that the development teams believe it needs to complete the planned scope of work. Project schedule performance is one of the factors that differentiate between a failed project and a successful project. (Linberg, 1999, p. 1)

Both productivity and quality can impact schedule performance. When considering productivity and schedule performance, it is important not to fall into the trap of assuming that increased productivity automatically correlates with improved schedule performance. While it is attractive to assume that if a development method allows more output for a given amount of input, the method would also support the originally planned amount of output for a given amount of input in an accelerated period. This assumption does not jibe with practical experience; for one thing, it fails to take into account that one or more critical paths exist in every development project. A critical path refers to the longest path through the project schedule; it determines the earliest possible time that the project can be brought to completion. Another characteristic of a critical path is that a critical path has the least amount of float or slack time, which represents “wiggle room” in the schedule, i.e., the amount of time that an activity can be delayed without delaying another activity or the project’s scheduled completion date. (Schwalbe, 2011, p. 228) Understanding of the critical path allows for effective schedule management; if activities on the critical path can be accelerated, it enables the possibility of compressing the project schedule. Two methods used to accelerate projects are crashing and fast tracking. Crashing refers to adding resources to a task in order to complete it faster than scheduled, while fast tracking refers to performing tasks in parallel that were originally scheduled to be performed in series. While both methods can be used to improve schedule performance, they both have associated drawbacks. Crashing often results in increased project costs due to the addition of resources. Fast tracking can lead to wasted effort and rework when activities are completed out-of-sequence. The critical path does not illustrate the shortest path through the project, nor does it illuminate the critical tasks of the project. Multiple critical paths can exist through a project, and the critical path can change as the schedule changes in response to completed project activities and project schedule challenges. The critical path displays the best-case time estimate through a project, essentially saying: This the best possible time we can make from the beginning of the project to the end, using the schedule as it now exists. (Schwalbe, 2011, p. 233)

Another reason that productivity gains do not automatically translate to improved schedule performance is that project dependencies can have a significant impact on schedule performance. A dependency refers to the sequencing of project activities within the project. There are three types of project dependencies:

1. **Mandatory dependencies**, also known as hard logic, are innate features of the work to be performed. For example, the ability to perform the testing-the-code activity depends on the completion of the writing-the-code activity.

2. **Discretionary dependencies**, also known as soft logic, are dependencies that are defined at the discretion of the project team. For example, it is good practice to refrain from starting the final user interface design until the customer has an opportunity to evaluate the prototype and provide feedback to the development team.

3. **External dependencies** are caused by interaction between non-project and project activities. An example is that the installation of a developed software package may depend on the installation of a new operating system onto the existing hardware by an external contractor. Even though the installation of the operating system may be outside of the scope of the project, this creates an external dependency because late installation of the operating system will adversely affect the project schedule. (Schwalbe, 2011, p. 217)
While productivity can affect the project schedule, it is important to understand its limitations. Recall from the Costs section that productivity is usually calculated using the formula KLOEC/PM. Productivity changes affecting costs usually affect the numerator of the productivity formula, while productivity changes affecting schedule performance usually affect the denominator of the formula. The denominator of the formula is the person-month. Brooks (1995) points out that reliance on the person-month for measuring effort used on software development projects can be misleading. Implicit in the person-month formula is the idea that people and time are the same, which is simply not supported by software development experience. Software development is an inherently complex activity, relying heavily on communication. If people and time were interchangeable, it would be possible to achieve any level of schedule performance desired simply by adding more people. In the original example, assume that the project which originally required two months for completion was instead needed in one week. If people and time were equivalent, the project manager in this case could simply assign a total of 36 people to the project; the original person-month metric of six is preserved, with the decrease in time to completion offset by the additional development staff. One factor that this scenario fails to incorporate is that communication is critically important to creative processes such as software development, and that adding more people to a project increases the number of communication channels. (Schwalbe, 2011, p. 395) The number of communication channels is calculated by the formula \( n(n-1)/2 \) where \( n \) is equal to the number of people involved in the project. In the original example, there is one communication channel. In the accelerated project, there are 630 communication channels. Brooks notes that adding people to a software development project can actually decrease team productivity. (Brooks, 1995, pp. 6 – 10) Nonetheless, the person-month is a convenient metric for comparing software development projects, because it is readily available for any managed project.

Project quality has a clearer relationship to schedule performance than productivity does. Quality for the cases studied is measured as a function of defects divided by thousands of lines of executable code [KLOEC]. Further, quality is divided into two categories, internally visible quality and externally visible quality. Internally visible quality refers to defects that are found and corrected during development, while externally visible quality refers to defects that are found after the product is released to the customer. Generally, increased defects negatively affect the project schedule, while decreased defects positively affect the project schedule. Quite simply, correcting software defects takes time; the more defects existing in the software, the longer it takes to correct them. (Pfleeger & Atlee, 2010, p. 426)

An additional area that can impact schedule performance is the method by which the software is delivered to the customer. Traditional software development projects usually gather requirements from the customer, develop the software based on the requirements, and then deliver the software to the customer once it’s complete. Agile development projects, on the other hand, usually stress continual delivery of the product, even if the product isn’t complete and provides only partial functionality. Additionally, there is usually a great deal more interaction with the customer throughout the development project when agile methods are used, compared to traditional methods. This additional interaction can result in improved delivery of the desired functionality; increased customer interaction and continual product delivery allow the team to catch and correct mistakes made during the requirements gathering phase, thus ensuring that the product delivered solves the problem that it was intended to solve. Developers of traditional projects, on the other hand, may not be aware that they are solving the wrong problem until the product is delivered to an unhappy and unsatisfied customer. (Wells, 2009) Although this is a clear advantage of agile methods, it does tend to confound the comparison of agile and traditional schedule performance.

The Sabre-A study experienced an increase in productivity and a decrease in fault density when XP methods replaced traditional methods. (Layman, et al., 2004a., p. 8) The Sabre-P study experienced a decrease in fault density when XP methods were used, compared to published industry averages,
while productivity was similar to or higher than published industry averages. (Layman, et al., 2004b., p. 8) The IBM study experienced an increase in productivity using all three measurement methods, and a decrease in fault density when XP methods replaced traditional methods. (Williams, Krebs, et al., 2004, p. 8) Recall from the costs section that costs, schedule and quality are interrelated; for any project, an alteration of two of these facets results in an induced change to the third. (Dillman, 2003, p. 6) Since all three cases studied exhibited a positive change in both quality and productivity when XP methods were used, we must conclude that the relationship is bi-directional and that these improvements induce an improvement in schedule performance as well. In essence, XP methods allow us to “have it both better and cheaper; an additional benefit is that this improved efficiency allows us to have it faster as well.” This conclusion is supported by observation from the studies that the products developed using XP methods included more functionality than was originally specified; in other words, the development teams were able to deliver more than was requested, while remaining within the original schedule constraints. In light of these results, the null-hypothesis that there are no significant differences between schedule performance of traditionally developed software projects and projects developed using agile methods is not supported by the evidence presented.

QUALITY COMPARISONS

“Quality” is a nebulous term, because it means different things to different people. Developers may feel that quality refers to solving the problem at hand in an elegant manner. Managers may feel that quality refers to keeping the project within the specified business constraints of budget, schedule, and scope. End users may feel that quality refers to an easy-to-use interface. They are all correct; quality encompasses all of these areas. For the purposes of analyzing the case studies, quality will be viewed from two viewpoints: technical quality and business quality. Technical quality refers to how defects are managed within the project: how many there are, how severe they are, how difficult they are to correct, and whether correcting them will cause new problems. Business quality refers to how well the project solves the intended problem.

Problems with computer software are referred to as defects or more properly, faults. A fault occurs when a software developer makes a mistake in performing a software activity. For example, a designer may misunderstand a requirement and create a design that does not match the actual needed functionality. This design fault becomes encoded in the software, and can cascade into other faults, such as incorrect documentation. Thus, a single mistake can generate many faults, and faults can occur in any constructed software. (Pfleeger & Atlee, 2010, p. 6)

Faults are the result of errors made during the creation or maintenance of software. Failures, on the other hand, are departures from a system’s desired behavior. An example of a failure that will be familiar to any Microsoft Windows user is the infamous stop error, also known as the “Blue Screen of Death.”
A fundamental problem in software engineering is the problem of how to handle software faults in complex systems. Modern computing systems are extremely complex; a few lines of a computer program written in an Object Oriented programming language can have an instruction explosion of 100:1 or more when compiled into machine language; some versions of the compilation system used may introduce faults into the software while others may not. Further, in most cases software doesn’t operate as a stand-alone system; modern computer systems are amalgams of system and application software that must work together as a cohesive unit. [For example, on the computer used to write this document, the Windows folder alone contains nearly 100,000 files in 20,000 folders.] Subtle versioning differences in supporting software can cause unintended interaction, triggering seldom-executed faulty code and causing failures. A particularly insidious type of fault that can be inconspicuous until it causes a failure is the memory leak. A memory leak occurs when memory that was previously allocated by a programmer is not properly de-allocated. When executed, memory that is no longer in use by the software is still considered reserved, and that memory cannot be used by any process until it is marked as available. When it occurs, computer memory “leaks away” or is wasted, causing degraded performance or even out-of-memory failures in extreme cases. (Sahgal, 2012) Additionally, faults may cause failures only intermittently, making them extremely difficult to find. An example is a design flaw in the Therac-25 software system, in which a variable was programmed to increment rather than being set to a specific value; on every 256th cycle this variable would overflow, allowing operation of the machine in a potentially deadly configuration. (Leveson, 1995, pp. 35 – 38) Finally, in a perfect world malware such as viruses, worms, and Trojan horses wouldn’t exist. Ours is not a perfect world; malware can introduce faults or trigger otherwise faulty but unused code in order to produce exploitable failures. The consensus within the software development community is that it is practically impossible to remove all potential faults from software, especially considering the fact that the software’s operating environment is constantly changing; the focus within the community has been on minimizing faults and mitigating risk from faults.
Additionally, understanding and correcting software problems can be confounded by the nature of software itself. Hardware is tangible; a resistor that is out of tolerance can easily be measured and rejected. Software, on the other hand, is intangible. Software is, by its very nature, an abstraction of reality; it isn’t possible to measure whether software is within tolerance, as it would be with a resistor or other physical component. Generally, software is checked for problems by inspection, either visually or by other computer systems in order to verify that the software is correct grammatically and syntactically, or by testing, in order to verify that the software is correct logically, i.e., that it does what it was intended to do.

Generally, faults represent an inside view of the system, as seen by the eyes of the developers. On the other hand, failures represent an outside view of the system—a problem that is seen while exercising the software, either by developers during testing or by users during use. A complicating factor is that software faults may—or may not—correspond to a failure; for example, if faulty code is never executed or a particular state is never entered, then the fault will never cause the system to fail. (Pfleeger & Atlee, 2010, p. 6)

Quality for the cases studied is measured as a function of defects divided by thousands of lines of executable code [KLOEC.] Further, quality is divided into two categories, internally visible quality and externally visible quality. Internally visible quality refers to faults that are found and corrected during development, while externally visible quality refers to failures that are found after the product is released to the customer.

The Sabre-A study experienced a decrease in total fault density, a decrease in internally visible fault density, a decrease in externally viewable fault density, and fewer severe faults when XP methods replaced traditional methods. (Layman, et al., 2004a., p. 8) The Sabre-P study experienced a decrease in total fault density, similar levels of internally visible fault density, and a decrease in externally visible fault density when XP methods were used, compared to published industry averages. (Layman, et al., 2004b., p. 8) The IBM study experienced a decrease in total fault density, a decrease in internally visible fault density, a decrease in externally visible fault density, and similar levels of fault severity when XP methods replaced traditional methods. (Williams, Krebs, et al., 2004, p. 8) Additionally, all three teams reported the perception that XP methods did a better job of providing what the customer actually wanted, as opposed to what the customer originally requisitioned. In light of these results, the null-hypothesis that there are no significant differences in quality of traditionally developed software projects and projects developed using agile methods is not supported by the evidence presented.

SATISFACTION COMPARISONS
Stakeholder satisfaction for the cases studied is measured through the use of a survey, the Shodan Adherence Survey [see Appendix C]. The survey is used to gather subjective data from the affected stakeholders as to how well the specific project adhered to XP principles, and how well XP functioned within the project. The survey is divided into six sections:

1. **Foundations.** Covers automated user tests, customer acceptance tests, test-first design, pair programming, and refactoring. The goal is to ascertain the sturdiness of the XP base used for developing the specific project.

2. **Customer Planning.** Covers release planning/the planning game, customer access, short releases, and stand-up meetings. The goal is to determine how effective customer interaction and XP planning activities are for developing the specific project.

3. **Teaming.** Covers continuous integration, coding standards, and collective code ownership. The goal is to determine how effective XP personnel methods are for developing the specific project.
4. **Craftsmanship.** Covers sustainable pace, simple design, and metaphor/system of names. The goal is to determine how effectively XP principles support the development effort for the specific project.

5. **Introspection.** Covers lessons learned, growth, morale, and artifact reduction. The goal is to determine what the respondents learned about the XP process as a result of using XP for developing the specific project.

6. **Perspectives.** Asks the respondent to describe three XP practices that are a cause for concern, and three XP practices that hold the most promise.

The Sabre-A study expressed a high level of stakeholder satisfaction, and a morale rating of 68.1%. The calculated average of the Shodan survey scores supplied for the Sabre-A New release is 76.67%. Anecdotally, the customer base was highly satisfied with the product of the Sabre-A project; one customer remarked that [it was] “one of the most professionally developed products he had ever used.” (Layman, et al., 2004a., pp. 5 – 7) Customer satisfaction and morale ratings are not summarized for the Sabre-P study as they are for the other studies; it is worth noting, however, that the calculated average of the Shodan survey scores supplied is 70.27%, which is comparable to the results reported in the other studies. (Layman, et al., 2004b., pp. 5 – 7) The IBM study expressed a high level of stakeholder satisfaction and a morale improvement of 11% when XP methods replaced traditional methods. (Williams, Krebs, et al., 2004, pp. 5 – 7) The calculated averages of the Shodan survey scores supplied for the IBM study are 56.33% [Old] and 73.67% [New]. The customer for this project reported satisfaction with the performance of the development team. Unfortunately, the XP-EF tool was modified between the IBM study and the Sabre-A study; one result of the change is that morale scores are reported using a relative scale for the IBM study, while they are reported using an absolute scale for the Sabre-A study, confounding a meaningful comparison. If we assume that a 50% morale rating is the midpoint between high morale and low morale, then the Sabre-A team reports an 18.1% improvement when XP methods replaced traditional methods, a finding that is similar to the results of the IBM study. [See Appendix E for a summary of the survey results.]

The Sabre-A team felt that daily stand-up meetings, short releases, and the rapid feedback afforded by unit testing were particularly helpful XP practices. Mandatory pairing and collective code ownership were considered problematic XP practices for the team; additionally, some developers noted that automated tests were difficult to write in some instances, due to technological limitations. (Layman, et al., 2004a., pp. 5 – 7) The Sabre-P team felt that continuous integration and simple design were helpful XP practices. Unit testing, pair programming, and refactoring were considered problematic XP practices for the team. (Layman, et al., 2004b., pp. 5 – 7) The IBM team felt that stand-up meetings, customer acceptance tests, and collective ownership were helpful XP practices. Short releases, test-first design, and refactoring were considered problematic XP practices for the team. (Williams, Krebs, et al., 2004, pp. 5 – 7) In light of these results, the null-hypothesis that there are no significant differences in stakeholder satisfaction for traditionally developed software projects and projects developed using agile methods is not supported by the evidence presented.

Management satisfaction is not covered in the main case studies per se. Examination of management satisfaction in similar cases is illuminative. Decreased fault density, improved quality, improved developer morale, and increased customer satisfaction are all generally beneficial to management. Additionally, management reports that the agile focus on collaboration positively affects flexibility; multiple developers are familiar enough with the project to continue developing it, in case of loss of critical personnel. (Ferreira & Cohen, 2008, p. 3) These benefits come with a caveat, however: Agile methods allocate some traditional management authority to other entities, such as developers and customers. Care must be taken to ensure that managers understand and buy into agile methods, in order to avoid “turf wars” and potential resentments. (Williams & Cockburn, 2003, p. 3)
An additional concern is the management of customer collaboration. Hanssen and Fægri (2006) note that customers remain engaged in the development process only as long as there are direct business benefits in doing so. Keeping the customer involved in the development effort is a critically important management task, as the development team depends on timely detailed feedback from the customer, to ensure that the product continues to develop in a useful direction. (Hanssen & Fægri, 2006, p. 6)

ADDITIONAL OBSERVATIONS
The Sabre-P study was the largest development project of the three studies. It is worth noting that XP practices showed signs of stress when applied to a project of this size. Stakeholders observed that the daily stand-up meeting became problematic with the larger team size, as some points brought up were irrelevant to others on the team. Additionally, stakeholders felt that customer access was problematic, as the customer representative was not available as often as needed; these team members felt that project planning suffered because of delayed customer input. Furthermore, stakeholders felt that the planning game was adversely affected by outside factors such as scope creep, which is the introduction of requirements that have not been approved by the affected stakeholders, and externally mandated deadlines. Occasionally, the team was forced to include features into a release before they were intended to be included, due to business demands; these features were forced into the release without the feature tradeoff that is mandated by XP. This project included a large amount of legacy code, which made unit testing problematic; developers often abandoned unit testing when faced with schedule pressures, despite entreaties not to do so. Similarly, pair programming was often abandoned in the face of schedule pressure, as the developers felt it was impossible to deliver the promised functionality unless each developer worked alone. Abandoning pair programming was also specifically discouraged by team leadership. Refactoring was often ignored due to fear that it would introduce new faults into the project. Several developers expressed concerns about collective code ownership, and stated that they had observed a decrease in the willingness to take responsibility for poorly written code. Further, there was a perception among the developers that the focus on continuous delivery caused them to lose sight of the overarching goals of the project. Unanimously, all of the developers felt that they were not working at a sustainable pace; the project incorporated fixed and extremely optimistic deadlines, and the development team had no power to reduce scope. As a result, the development team was forced to work consistent overtime in order to meet the project deadlines. (Layman, et al., 2004b., pp. 5 – 7)
CHAPTER 4: CONCLUSIONS

INTRODUCTION

This paper has explored the performance of the agile software development method Extreme Programming compared to traditional software development methods through a comparison and contrast analysis of three published case studies. The purpose of exploring these case studies has been to determine if there are measurable differences in performance between agile development methods and traditional methods.

The first sub-question is: Are there significant and measurable differences in cost performance between software development projects using traditional development methods, compared to software development projects using agile development methods? The null-hypothesis for this question is that there are no significant differences between development costs of traditionally developed software projects and projects developed using agile methods.

The second sub-question is: Are there significant and measurable differences in schedule performance between projects using traditional development methods, compared to projects using agile development methods? The null-hypothesis for this question is that there are no significant differences in schedule performance between traditionally developed software projects and projects developed using agile methods.

The third sub-question is: Are there significant and measurable differences in quality between projects using traditional development methods, compared to projects using agile development methods? The null-hypothesis for this question is that there are no significant differences in quality between traditionally developed software projects and projects developed using agile methods.

The fourth sub-question is: Are there significant differences in stakeholder satisfaction between projects using traditional development methods, compared to projects using agile development methods? The null-hypothesis for this question is that there are no significant differences in stakeholder satisfaction between traditionally developed software projects and projects developed using agile methods.

SUMMARY

The first sub-question asks if there are significant and measurable differences in cost performance between software development projects using traditional development methods, compared to software development projects using agile development methods. In the case of these three case studies, the null-hypothesis for this question, specifically that there are no significant differences between development costs of traditionally developed software projects and projects developed using agile methods, is not supported by the evidence; significant and measurable improvements were found.

The second sub-question asks if there are significant and measurable differences in schedule performance between projects using traditional development methods, compared to projects using agile development methods. The null-hypothesis for this question, specifically that there are no significant differences in schedule performance between traditionally developed software projects and projects developed using agile methods, is not supported by the evidence; significant and measurable improvements were found.

The third sub-question asks if there are significant and measurable differences in quality between projects using traditional development methods, compared to projects using agile development methods. The null-hypothesis for this question, specifically that there are no significant differences in quality between traditionally developed software projects and projects developed using agile methods, is not supported by the evidence; significant and measurable improvements were found.

The fourth sub-question asks if there are significant differences in stakeholder satisfaction between projects using traditional development methods, compared to projects using agile development methods. The null-hypothesis for this question, specifically that there are no significant differences in stakeholder satisfaction between traditionally developed software projects and projects developed using agile methods, is not supported by the evidence; significant and measurable improvements were found.
The fourth sub-question asks if there are significant differences in **stakeholder satisfaction** between projects using traditional development methods, compared to projects using agile development methods. The null-hypothesis for this question, specifically that there are no significant differences in stakeholder satisfaction between traditionally developed software projects and projects developed using agile methods, is not supported by the evidence; significant and measurable improvements were found.

In the light of the fact that significant and measurable differences were found in every instance, is it safe to conclude that agile software development methods are demonstrably superior to traditional methods? No, it is not. Firstly, despite the consistency of the findings across the three case studies, a sample size of three is too small a sample to be conclusive; more studies and more varied studies need to be evaluated and included into the body of knowledge before definitive conclusions can be drawn. Secondly, the three studies examined utilized one specific agile method, Extreme Programming. Further studies are needed to determine if the improvements found using XP are generally applicable. Finally, the Sabre-P project exhibited evidence of stress on the XP processes used. As the Sabre-P project was the largest of the three projects examined, this stress is indicative that conventional wisdom may be correct as it relates to agile methods used in non-agile problem domains. Further study is needed to determine the efficacy of agile methods in large projects and for safety-critical systems.

**FURTHER RESEARCH**

As discussed by Leedy and Ormond, (2010, p. 7) research is rarely conclusive; in exploring one set of research questions, additional questions and areas of exploration are discovered. The author presents the following as areas of further interest:

1. The XP evaluation framework is useful in normalizing data drawn from differing XP development projects, thus enabling meaningful analysis. It would be a boon to the industry to expand this tool into a family of tools supporting additional agile development methods. If additional methods were supported, developers could use the tool to see if the conclusions drawn from the evaluated XP development projects apply to agile methods generally. Such a tool family would be especially useful considering the popularity of other agile methods like Scrum and Feature Driven Development.

2. The polar chart developed by Boehm and Turner (2003), used in the developmental category of the XP-cf, uses a plot of five project risk factor categories to determine a project’s optimal method. As pointed out by Williams, Krebs, & Layman (2004, p. 13), it is in the best interests of the industry to apply this classification model to as many and as varied development projects as possible, in order to both validate the model, and determine the model’s usefulness in creating custom hybrid methods. Experience has shown that the implementation of agile practices can be beneficial even when only a subset of possible practices is used. (Williams, Krebs, et al., 2004, p. 8)

3. Conventional wisdom within the software development community is that agile methods are inappropriate for developing safety-critical applications. (Lindvall, et al., 2002, p. 7) A possible cause for this belief is found in the origins of agile methods. Before the agile movement had a name, these methods were referred to as “lightweight” methods; this terminology was dropped in favor of the word “agile,” in part because it was felt that no one would take lightweight methods seriously. (Beck, et al., 2001) On the surface, it is obvious that lightweight methods should not be used for developing software on which lives depend; however, is this judgment backed by empirical evidence, or is it an example of expectation bias? If we accept the idea that “all great truth begins as blasphemy” (Walsch, 2004, p. 5), then a valuable addition to the body of knowledge concerning software development would be the initiation of a safety-critical application...
project to be developed using agile methods; evaluation of the results of this project could begin to answer questions as to the correctness of conventional wisdom in this instance.

4. Due to the relative newness of agile development methods, the majority of developers currently in the workforce learned traditional development methods before they learned agile methods. There is a tendency in human behavior to “fall back” to the most familiar habits when under stress. This tendency was observed in the Sabre-P study when the project was stressed due to unrealistic project goals; the project team abandoned XP processes in favor of the more familiar traditional processes. An interesting addition to the body of knowledge would be to observe a stressed agile development project, where the development team is most familiar or exclusively familiar with agile processes, thus removing this fall-back tendency from the evaluation.

CONCLUSIONS
Software development is a developing discipline. (Siakas & Siakas, 2007, p. 2) This study adds to the body of knowledge concerning software development. It indicates that agile methods are useful under certain development conditions, and that at least for projects that are within the agile boundary set, agile methods are measurably more effective in producing high-quality software and happy stakeholders. Despite these findings, additional research is necessary in order to clearly delineate the outer edges of the agile boundary set, assuming they exist. It is certain that global dependence on high-quality software will increase in the future. In light of this dependence, it becomes increasingly more urgent that the most effective methods possible for software development be discovered, proven, and implemented. Epstein (2004) suggests that software development can be used as spiritual metaphor, and that the various software development methods correspond to different spiritual paths. (Epstein, 2004) Just as it is important that each individual finds the spiritual path that empowers them to be happily and usefully whole, it is important that each software development project be developed using the most appropriate development method, so that the desired software can be created with the best possible results.
REFERENCES


APPENDIXES

APPENDIX A

The Agile Manifesto (Beck, et al. 2001)

**The Twelve Principles**

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain indefinitely a constant pace.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity — the art of maximizing the amount of work not done — is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
APPENDIX B

Comparative features of agile software development processes. (Abrahamsson, et al., 2002, pp. 89 – 90)

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Key Points</th>
<th>Special Features</th>
<th>Identified Shortcomings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extreme</td>
<td>Customer driven development, small teams, daily builds.</td>
<td>Refactoring, the ongoing redesign of the system to improve its performance and responsiveness to change.</td>
<td>While individual practices are suitable for many situations, the overall view and management practices are given less attention.</td>
</tr>
<tr>
<td>Programming</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scrum</td>
<td>Independent, small, self-organizing development teams, 30-day release cycles.</td>
<td>Enforce a paradigm change from the “defined and repeatable” paradigm to the “new product development” view.</td>
<td>While scrum specifically details how to manage the 30-day release cycle, integration and acceptance testing is not detailed.</td>
</tr>
<tr>
<td>Crystal Family</td>
<td>Each member of the family is based on the same core values. Techniques, roles, tools, and standards vary.</td>
<td>Method design principles. Ability to select the most suitable method based on project size and criticality of application.</td>
<td>Only two of the four suggested members are realized; insufficient practical application experience to determine its effectiveness.</td>
</tr>
<tr>
<td>DSDM</td>
<td>Application of controls to RAD, use of time boxing, empowered DSDM teams, active consortium to drive method development.</td>
<td>First truly agile software development process, use of prototyping, several user roles: “Ambassador,” “Visionary,” and “Advisor.”</td>
<td>Proprietary; only members of the DSDM consortium have access to white papers dealing with actual experience with the process.</td>
</tr>
<tr>
<td>FDD</td>
<td>Five-step process, object-oriented (feature) based development. Very short iterations, from hours up to 2 weeks.</td>
<td>Method simplicity, design and build the system by features, object modeling.</td>
<td>FDD focuses only on design and implementation, needs additional supporting processes.</td>
</tr>
<tr>
<td>ASD</td>
<td>Adaptive culture, collaboration, mission-driven component based iterative development.</td>
<td>Organizations are seen as adaptive systems, creating an emergent order out of a web of interconnected individuals.</td>
<td>ASD is more about concepts and culture than about software practice.</td>
</tr>
</tbody>
</table>
## APPENDIX C

*The Shodan 2.0 Input Metric Survey (Williams, Krebs, & Layman, 2004, pp. 26 – 33)*

<table>
<thead>
<tr>
<th>When answering ‘how often’ questions, the scale below may help:</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Fanatic (100%)</td>
</tr>
<tr>
<td>9 Always (90%)</td>
</tr>
<tr>
<td>8 Regular (80%)</td>
</tr>
<tr>
<td>7 Often (70%)</td>
</tr>
<tr>
<td>6 Usually (60%)</td>
</tr>
<tr>
<td>5 Half ’n Half (50%)</td>
</tr>
<tr>
<td>4 Common (40%)</td>
</tr>
<tr>
<td>3 Sometimes (30%)</td>
</tr>
<tr>
<td>2 Rarely (20%)</td>
</tr>
<tr>
<td>1 Hardly ever (10%)</td>
</tr>
<tr>
<td>0 Disagree with using this practice</td>
</tr>
</tbody>
</table>

### Automated Unit Tests (40 of 640)

Automated unit tests (such as JUnit) are an essential part of the development process. They provide a tested environment for verifying the correctness of software and allow code to be safely integrated and disseminated among team members. To what extent do you employ automated unit tests?

- Automated unit tests exist for production code.
- A test is used to measure test coverage.
- There is an automated way to run the entire suite of unit tests for an entire program.
- All unit tests are run and passed when a task is finished and before checking in to source control.
- When fixing bugs, unit tests are used to capture the bug before fixing.
- Unit tests are refactored.
- Unit tests are fast enough to be run all the time.

### Customer Acceptance Tests (20)

Customer acceptance tests exist to ensure both the developers and the customer know what they want. All acceptance tests must be passed before the product can be delivered to the customer. How important are customer acceptance tests to the development of your product?

- Acceptance tests are used to verify system functionality and customer requirements.
- Customer provides acceptance criteria.
- Customer uses acceptance test to determine what has been accomplished at the end of an iteration.
- Acceptance testing is automated.
- A User Story is not finished until its acceptance tests pass.
- Acceptance tests are run automatically every night.
- A test environment that matches the end-user’s environment is used to test.

### Test-First Design (20)

Test-first design is the practice by which a test case is written before the code is implemented. The implemented code is written to pass the test case. This practice produces higher quality code and higher programmer confidence in their code. What percentage of the time do you employ test-first design?

- Code is only written after a unit test (that fails) has been written first.
- All production code is written using test-first design.

### Pair Programming (80)

Two people, one computer. One thinks strategy, the other thinks tactics. This practice produces higher quality code at the same level of productivity. What percentage of your work (design, analysis, coding) was done in pairs?

- People can go on vacation without regard to what work needs done.
The Performance of Agile Methods: Comparison to Traditional Development Methods

- Drivers and Navigators switch roles often.
- Navigators keep a to-do list.
- People switch pairing partners at least once per day.
- The team has work stations conducive to pair programming.
- Practices are enforced by peer pressure.
- Production code is not written without a pair.
- Repetitive and dull tasks that would not gain from pair programming are automated.

### Refactoring (70)

Rewrite code that 'smells bad' to improve future maintenance and flexibility without changing its behavior. How often do you stop to clean up code that has already been implemented without changing functionality?

- Code contains minimal or no duplication.
- Team refactors often or when applicable.
- There are enough unit tests and/or automated acceptance tests to allow merciless refactoring.
- Any code is open to refactoring.
- Refactoring is done only to improve existing code and not to anticipate future tasks.
- Future refactoring has been identified.
- Long term refactoring is going on now.

### Customer Planning

#### Release Planning/Planning Game (38)

The planning game is a highly interactive process between all stakeholders wherein customers and developers trade items in and out of the plan based on current priorities and costs. Adaptation is favored over following a plan. Do you allow for changes in release plans/requirements after each iteration based on customer feedback and current implementation? How well does planning correspond to the criteria below?

- There is a release plan.
- The whole team, including coach, customer, developer, etc., is present during release planning.
- The customer picks the order of the User Stories in the release plan.
- When stories are added to a release, stories of equal value may be re-prioritized.
- Developers estimate the time needed to complete the User Stories.
- Developers break down User Stories into tasks. Each developer signs up for tasks and estimates the ones he/she owns.
- The release plan is used to determine how much can be done by a certain time.
- Past User Story experience aids in determining how much can be done by a certain time.
- Release points have been identified and communicated to all stakeholders.
- At least one User Story is created for automating acceptance tests.

### Customer Access (32)

The customer is the body for whom the product is being developed and may be either internal or external. Customer access is imperative to developing a product that satisfies the customers' needs as well as clear up requirement ambiguity/inequality. On-site Customer is best, but you can use chat, e-mail, telephone, etc., to quickly verify requirements and get feedback. Ideally, the customer is always available. What % of the time do you get quick interaction with your customers when needed?

- Customer is involved in release planning.
- The developers have direct access (telephone/email/video conference) to the customer.
- The developers have same day responses from customer.
- The customer is on-site.
- Fast and consistent feedback between customer and developer.

### Short Releases (40)
You have frequent smaller releases instead of larger less frequent ones. This lets the customer see the progress of the project and allows the developer to get feedback. **How close are you to having releases at about 6 months with interim iterations of a couple weeks?**

- The customer has identified release points.
- The product is releasable (internally or externally) every six months or less.

**Stand-Up Meeting (5)**

The team takes 10 minutes each day to review what needs to be done each day and assigns user tasks to team members. **Does your team engage in stand-up meetings every day?**

- **Stand up meetings take place every day.**
- **Stand up meeting takes less than 15 minutes (for an average of 10 people).**
- **Meetings are short and to the point, focusing only on what has been done and needs to be done that day.**
- **Team members exhibit courage in discussing concerns and successes.**

**Teaming (60)**

**Continuous Integration (50)**

Continuous integration works in concert with collective code ownership to ensure that developers have the most recent version of code available. **Code is checked in quickly to avoid code synchronization/integration hassles. How often do you synchronize and check in your code on average?**

- Source control/VCS is used.
- A build machine automatically builds at least once per day.
- Unit tests and acceptance tests are run as a part of each build.
- The build machine informs developers when the build fails.
- The build process is fast enough to support continuous integration.
- The team integrates at least once per day preferably several times per day.

**Coding Standards (50)**

**Do you have and adhere to team coding standards?** Besides brace placement, this may include things like logging and performance idioms. **Strong standards make collaboration, refactoring, and collective ownership an easier process. Is there a coding standard in place and how often is it followed?**

- There is a coding standard for all used languages.
- The coding standard is known and used by the whole team.
- A tool exists to format the code in accordance with the coding standard.
- The coding standard is short and covers readability issues and not design issues.

**Collective Code Ownership (50)**

You can change anyone’s code and they can change yours. This allows for stronger knowledge transfer amongst the team and ensures that you don’t get stuck when the expert is busy or on vacation. **People know many parts of the system. Can people change code they did not originally write, and how often do they do so?**

- Some form of VCS is used that allows multiple people to work on the same file at the same time.
- Everyone is allowed to change any code.
- Everyone has the knowledge to change any code or can pair with a person that has the knowledge.
- There are enough unit tests and/or automated acceptance tests to allow people to safely change any code.

**Craftsmanship (60)**

**Sustainable Pace (50)**

People need to be effective over the long haul. Overworking has negative impacts on productivity, morale,
and home life. How well do you pace yourself?

- High rate of productivity is maintained without being overworked.
- Overtime is not worked more than two weeks in a row.
- Vacations or classes are never postponed or canceled due to work.

Examples Score: 10 - I maintain a sustainable pace and the same high rate of output. 5 - I work longer than what I consider a sustainable pace, but still produce at a high rate and feel only a little burnt out. 2 - I work beyond a sustainable pace and feel burnt out. My code isn't at its usual high quality.

Simple Design (65)
Keep it simple at first; do the simplest thing that could possibly work. You don't follow the philosophy of "I'll include this because the customer might possibly need it later" even though the feature isn't in the requirements. Also, you do not spend a lot of time on design documents. How often do you succeed in "Keeping it Simple"?

- Always do "the simplest thing that can possibly work".
- Follow the principle of YAGNI, "You aren't going to need it!" Only build what is currently necessary.
- Refactoring is used to keep design clean.
- There is no unused or commented out code.

Metaphor/System of Names (35)
A single, overarching metaphor is used to describe the system, such as an "assembly line." In the event that this is not possible, the team may use a "system of names" to describe the various components of the project in a consistent manner. For example, all items related to the database are prefixed by Database. The metaphor/system of names is used by developers to help communicate ideas and to explain concepts to customers. How often do you feel this is true of the systems you develop?

- Classes and methods have good, descriptive names.
- Classes and methods have names relative to one another.
- New members of the team do not need to often ask or refer to a document to understand the architecture.
- The customer understands can explain the metaphor.
- The developer understands can explain the metaphor.

Introspection
Lessons Learned (6)
- The team reviews how to get better after every release.

Growth (6)
- Consider the latest tools and practices in addition to skills. If you're not learning, you're falling behind!

Morale (6)
- How often can you say you're enjoying your work? Oh, this is really an output of your process, but it's collected here for validation and to see if XP is enjoyable.

Artifact Reduction (7)
With agile methods, you have fewer/thinner versions of artifacts from classic techniques. This saves time, which can be invested in better tests, new code, refactoring, etc. To what extent have you been able to:

- Have fewer code reviews (Pairing instead)
- Thinner design specs (1st First Design)
- Lighter comments/internal docs (Simple Design, Refactoring)

Perspectives
Please list up to three practices that most concern you, i.e. those that you think will be difficult to adopt and/or non-effective. (Optional)

Please select up to three XP practices that seem to hold the most promise to you. (Optional)
### Combined EF-om results for the three case studies

<table>
<thead>
<tr>
<th></th>
<th>Sabre-A Old (Relative Scale)</th>
<th>Sabre-A New (Relative Scale)</th>
<th>Sabre-P (Compared to Industry Averages)</th>
<th>IBM Old (Relative Scale)</th>
<th>IBM New (Relative Scale)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response to Customer Change (Ratio of user stories in + out / total)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.23</td>
</tr>
<tr>
<td>Internally-Visible Quality (test defects / KLOEC of code)</td>
<td>1.00</td>
<td>0.35</td>
<td>Similar</td>
<td>1.00</td>
<td>0.50</td>
</tr>
<tr>
<td>Externally-Visible Quality (released defects / KLOEC of code, post release)</td>
<td>1.00</td>
<td>0.64</td>
<td>Higher[a]</td>
<td>1.00</td>
<td>0.61</td>
</tr>
<tr>
<td>Productivity - KLOEC / person-month</td>
<td>1.00</td>
<td>1.46</td>
<td>Similar</td>
<td>1.00</td>
<td>1.70</td>
</tr>
<tr>
<td>Productivity - User stories / person-month</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1.00</td>
<td>1.34</td>
</tr>
<tr>
<td>Productivity Function Points / person-month</td>
<td>N/A</td>
<td>N/A</td>
<td>Higher</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Productivity - Putnam Productivity Parameter</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1.00</td>
<td>1.92</td>
</tr>
<tr>
<td>Customer Satisfaction</td>
<td>N/A</td>
<td>High</td>
<td>N/A</td>
<td>N/A</td>
<td>High</td>
</tr>
<tr>
<td>Morale (via survey)</td>
<td>N/A</td>
<td>68.20%</td>
<td>N/A</td>
<td>1.00</td>
<td>1.11</td>
</tr>
</tbody>
</table>

Notes:
1. Post release time periods vary across the three studies. Sabre-A is 4 months, Sabre-P and IBM are 6 months.
2. KLOEC were translated into function points for this metric. Function points were estimated from lines of code using the 1996 version of the Programming Languages Table. This estimation technique, known as “backfiring,” has been shown to have an accuracy of ±20%.
3. The Putnam productivity parameter is a macro measure of the total development environment. The PFP is calculated via the following equation: PFP = (SLOC)/(Effort/6B) \(^{1/2}\) \(*\) (Time)\(^{1/3}\). Putnam based this equation on production data from a dozen large software projects. Effort is the staff years of work done on the project. B is a factor that is a function of system size, chosen from a table constructed by Putnam based on the industrial data. SLOC is source lines of code, and Time is number elapsed years of the project. (Putnam & Myers, 1992)
4. Relative scales are used to protect proprietary data.
5. A different version of the XP-ET framework was used for the IBM study, compared to the Sabre studies. One result is a relative measure for morale for the IBM study, vs. an absolute measure for morale for the Sabre-A study. Despite this difference, both metrics indicate an overall improvement in morale due to the implementation of XP practices, as reported by team members via survey.
6. The original study reported a lower total defect density and a similar pre-release defect density, which implies that a decrease in post-release defect density drives the overall lower total defect density findings. Lower defect density correlates to higher quality.
## APPENDIX E

### Combined EF-am results for the three case studies

<table>
<thead>
<tr>
<th></th>
<th>Sabre-A Old</th>
<th>Sabre-A New</th>
<th>Sabre-P</th>
<th>IBM Old</th>
<th>IBM New</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective Measures</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Coding</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pairing Frequency</td>
<td>0%</td>
<td>50%</td>
<td>70%</td>
<td>11%</td>
<td>48%</td>
</tr>
<tr>
<td>Inspection Frequency</td>
<td>60%</td>
<td>20%</td>
<td>0%</td>
<td>2%</td>
<td>3%</td>
</tr>
<tr>
<td>Solo Frequency</td>
<td>100%</td>
<td>50%</td>
<td>30%</td>
<td>87%</td>
<td>49%</td>
</tr>
<tr>
<td><strong>Testing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test Coverage</td>
<td>N/A</td>
<td>33%</td>
<td>8% of lines</td>
<td>30% of lines</td>
<td>46% of lines</td>
</tr>
<tr>
<td>Test Run Frequency</td>
<td>N/A</td>
<td>100%</td>
<td>40%</td>
<td>&lt; 10%</td>
<td>11%</td>
</tr>
<tr>
<td>Test Class to Story Ratio</td>
<td>N/A</td>
<td>322%</td>
<td>2.25%</td>
<td>N/A</td>
<td>45%</td>
</tr>
<tr>
<td>Test LOC to Code LOC</td>
<td>5%</td>
<td>30%</td>
<td>0%</td>
<td>&lt;30%</td>
<td>42%</td>
</tr>
<tr>
<td><strong>Planning</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Release Length</td>
<td>18 months</td>
<td>3.5 months</td>
<td>3 Months</td>
<td>10 months</td>
<td>5 months</td>
</tr>
<tr>
<td>Iteration Length</td>
<td>None</td>
<td>10 days</td>
<td>10 work days</td>
<td>Weekly</td>
<td>Weekly</td>
</tr>
<tr>
<td>Requirements Added / Removed to Total Shipped Ratio</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.23 (13 add, 1 rem, 60 shipped)</td>
</tr>
<tr>
<td><strong>Subjective Measures (Shodan Adherence Survey)</strong></td>
<td>Sabre-A Old</td>
<td>Sabre-A New (mean &amp; std dev)</td>
<td>Sabre-P</td>
<td>IBM Old</td>
<td>IBM New</td>
</tr>
<tr>
<td><strong>Coding</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pair Programming</td>
<td>N/A</td>
<td>67.3% (16.2)</td>
<td>61.5% (22.3)</td>
<td>32% (15.0)</td>
<td>68% (14.6)</td>
</tr>
<tr>
<td>Refactoring</td>
<td>N/A</td>
<td>66.4% (18.6)</td>
<td>59.5% (20.4)</td>
<td>38% (11.6)</td>
<td>57% (14.9)</td>
</tr>
<tr>
<td>Simple Design</td>
<td>N/A</td>
<td>78.2% (9.8)</td>
<td>69.0% (21.0)</td>
<td>75% (10.5)</td>
<td>78% (6.9)</td>
</tr>
<tr>
<td>Collective Ownership</td>
<td>N/A</td>
<td>70.0% (8.4)</td>
<td>70.0% (21.0)</td>
<td>50% (14.0)</td>
<td>63% (7.5)</td>
</tr>
<tr>
<td>Continuous Integration</td>
<td>N/A</td>
<td>85.8% (6.0)</td>
<td>89.5% (7.6)</td>
<td>58% (18.8)</td>
<td>78% (12.4)</td>
</tr>
<tr>
<td>Coding Standards</td>
<td>N/A</td>
<td>90.0% (6.3)</td>
<td>80.5% (14.7)</td>
<td>87% (7.0)</td>
<td>82% (3.7)</td>
</tr>
<tr>
<td>Sustainable Pace</td>
<td>N/A</td>
<td>80.0% (11.8)</td>
<td>61.0% (25.9)</td>
<td>57% (12.5)</td>
<td>77% (9.4)</td>
</tr>
<tr>
<td>Metaphor</td>
<td>N/A</td>
<td>56.4% (27.3)</td>
<td>55.0% (25.7)</td>
<td>32% (30.7)</td>
<td>43% (18.9)</td>
</tr>
<tr>
<td><strong>Testing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test First Design</td>
<td>N/A</td>
<td>67.3% (14.2)</td>
<td>60.0% (21.0)</td>
<td>17% (11.2)</td>
<td>55% (22.2)</td>
</tr>
<tr>
<td>Automated Unit Tests</td>
<td>N/A</td>
<td>78.2% (23.2)</td>
<td>74.0% (23.0)</td>
<td>43% (16.4)</td>
<td>67% (22.1)</td>
</tr>
<tr>
<td>Cust. Acceptance Test</td>
<td>N/A</td>
<td>56.4% (20.2)</td>
<td>64.0% (26.6)</td>
<td>63% (25.6)</td>
<td>78% (6.9)</td>
</tr>
<tr>
<td><strong>Planning</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stand up meetings</td>
<td>N/A</td>
<td>92.7% (10.1)</td>
<td>98.0% (4.1)</td>
<td>72% (16.4)</td>
<td>90% (14.1)</td>
</tr>
<tr>
<td>Short Releases</td>
<td>N/A</td>
<td>91.8% (11.7)</td>
<td>75.5% (22.6)</td>
<td>78% (27.3)</td>
<td>77% (9.4)</td>
</tr>
<tr>
<td>Customer Access / On-site Customer</td>
<td>N/A</td>
<td>89.1% (12.2)</td>
<td>70.0% (23.6)</td>
<td>60% (28.1)</td>
<td>87% (4.7)</td>
</tr>
<tr>
<td>Planning Game</td>
<td>N/A</td>
<td>84.5% (10.4)</td>
<td>66.5% (17.3)</td>
<td>75% (21.2)</td>
<td>85% (10.0)</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>76.27% (14.4)</td>
<td>70.26% (19.8)</td>
<td>56.33% (17.8)</td>
<td>73.66% (11.9)</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX F
Developmental factors polar charts from the three case studies

Sabre-A Old

Sabre-A New

Sabre-P

IBM
ABOUT THE AUTHOR

F. Davis Cardwell fell in love with Agile methods during classes for his master’s degree. A seasoned software developer, Davis is a CSC financial analyst who builds productivity tools for the company’s corporate Service Center and analyzes data for the Center. Based in Sterling, Virginia, Davis recently earned his M.S. in information systems, software engineering management from Strayer University. In addition to software development, his passions include cooking, classic cars, building computers, and writing short stories. His current learning challenge is teaching himself how to weld using the GTAW process.
About CSC

The mission of CSC is to be a global leader in providing technology-enabled business solutions and services.

With the broadest range of capabilities, CSC offers clients the solutions they need to manage complexity, focus on core businesses, collaborate with partners and clients and improve operations.

CSC makes a special point of understanding its clients and provides experts with real-world experience to work with them. CSC leads with an informed point of view while still offering client choice.

For more than 50 years, clients in industries and governments worldwide have trusted CSC with their business process and information systems outsourcing, systems integration and consulting needs.

The company trades on the New York Stock Exchange under the symbol “CSC.”

DISCLAIMER

The information, views and opinions expressed in this paper constitute solely the author’s views and opinions and do not represent in any way CSC’s official corporate views and opinions. The author has made every attempt to ensure that the information contained in this paper has been obtained from reliable sources. CSC is not responsible for any errors or omissions or for the results obtained from the use of this information. All information in this paper is provided “as is,” with no guarantee by CSC of completeness, accuracy, timeliness or the results obtained from the use of this information, and without warranty of any kind, express or implied, including but not limited to warranties of performance, merchantability and fitness for a particular purpose. In no event will CSC, its related partnerships or corporations, or the partners, agents or employees thereof be liable to you or anyone else for any decision made or action taken in reliance on the information in this paper or for any consequential, special or similar damages, even if advised of the possibility of such damages.