Regis University
ePublications at Regis University

All Regis University Theses

Fall 2009

Integration and Deployment Techniques in Combination with Development Methodologies

Brian E. Nesbitt
Regis University

Follow this and additional works at: https://epublications.regis.edu/theses

 Part of the Computer Sciences Commons

Recommended Citation
Nesbitt, Brian E., "Integration and Deployment Techniques in Combination with Development Methodologies" (2009). *All Regis University Theses*. 42.
https://epublications.regis.edu/theses/42

This Thesis - Open Access is brought to you for free and open access by ePublications at Regis University. It has been accepted for inclusion in All Regis University Theses by an authorized administrator of ePublications at Regis University. For more information, please contact epublications@regis.edu.

Regis University
College for Professional Studies Graduate Programs
Final Project/Thesis

Disclaimer

Use of the materials available in the Regis University Thesis Collection ("Collection") is limited and restricted to those users who agree to comply with the following terms of use. Regis University reserves the right to deny access to the Collection to any person who violates these terms of use or who seeks to or does alter, avoid or supersede the functional conditions, restrictions and limitations of the Collection.

The site may be used only for lawful purposes. The user is solely responsible for knowing and adhering to any and all applicable laws, rules, and regulations relating or pertaining to use of the Collection.

All content in this Collection is owned by and subject to the exclusive control of Regis University and the authors of the materials. It is available only for research purposes and may not be used in violation of copyright laws or for unlawful purposes. The materials may not be downloaded in whole or in part without permission of the copyright holder or as otherwise authorized in the "fair use" standards of the U.S. copyright laws and regulations.

Abstract

Efficient and inefficient pairings of software development methodologies and software integration and deployment techniques exist. Often times the automation of code integration and deployment is chosen but the full benefit of these technologies are throttled by the incorporation of a development methodology. It can be hypothesized that the evolution of software development created this situation along with the latency of implementing development methodologies. This work examines four scenarios comprised of traditional and conventional development methodologies with manual and automated software integration and deployment techniques. Similar web-based software applications were selected from waterfall (traditional) and agile (conventional) run project development teams. The four scenarios were quantitatively analyzed through the use of a subjective component which took into account the common characteristics of each scenario. It was thought that the use of automation within an agile development methodology would show clear distinction when compared to the other three evaluation scenarios. However as discussed in the analysis, automated integration and deployment technologies benefited both waterfall and agile methodologies. Though due to agile's foundational characteristics of small iterations with constant integration and deployments, the automation of both practices had more of a realized value and benefit.

Table of Contents

CHAPTER ONE

Introduction

Statement of the Problem

The hardware and software sides of technology continually evolve and progress. Technology has evolved to aid software development in addressing business and customer requirements. Methodology frameworks have been conceptualized and practiced in the attempt to meet software requirements and deliver a product within a given timeframe. There are many optimal matches of software and hardware technologies. There also exists an optimal pairing of a software development methodology for web-based applications and that of development technologies. All too often organizations decide to implement development technologies for software integration and deployment in traditional development methodologies. When this occurs, the full potential of integration and deployment technologies can be undercut or unrealized. Vice versa, the pairing of more conventional development methodologies with that of manual software integration and deployment processes can also occur.

This study will examine the leveraging of automated integration and deployment technologies in agile development projects. The examination will focus on the potential for more consistent and reliable delivery of web based software applications. The research and analysis will focus on the development of web applications and the relationship between two development methodologies and two integration and deployment processes. Through this research and analysis, the study's focus on

consistent application delivery and reliability with different development methodologies will be evaluated.

The proposed study will use qualitative methods to evaluate whether or not automated integration and deployment technologies in agile development projects can result in more consistent software delivery. The study will utilize four different scenarios to compare quantitative results and observations. These scenarios will consist of combinations of development methodologies and integration and deployment techniques.

Background

Just as technology and its utilization in business changes so does a technology's innovation and evolution progress. Progress and innovation are a fact of any component's need to survive, improve and become more efficient. Information technology and its integral component of software development is no stranger to change, innovation and progress. It is worth while to briefly discuss the history of computers and how it is tied to the emergence of development methodologies and tools to aid in the control of software complexities.

By simply examining the history of software development it is possible to see its evolution from relays (basic mechanic switch) driven by an electric circuit to people such as Turnig and von Neuman who provided the mathematical foundation for programmable machines (GenerExe, 2002). These programmable machines slowly resulted in program-memory, data-memory, accumulators and central processing units. The read only process of these early programmable machines improved from the use of

fuses and circuits to punch-cards for fast input and then to cathode-ray-tubes for the presentation of results. Later keyboards were associated to machines so as to interpret human-readable text into punch-cards. Data loaded via punch-cards was replaced by magnetic storage devices and shortly later, the fast magnetic memory allowed for increased amounts of variable information to be stored during the execution of a program (GenerExe, 2002). In the late 1950s and early 1960s, software literally meant 'soft' hardware, which essentially equated to pliable electronics (GenerExe, 2002). Computer programs during this time were much akin to programmable calculators.

By 1970 the large complexity of computer systems could be mastered intellectually by one tool only: Abstraction (Wirth, 1999). The conceptual abstraction of computer machine objects and constructs along with the 1975 birth of the micro-computer (the Alto workstation) completely revolutionized and increased the speed of learning and developing software. "The Alto caused nothing less than a revolution, and as a result people today have no idea, how computing was done before 1975 without, highly personal highly interactive workstations" (Wirth, 1999). The evolution of computer hardware capabilities aided in the increase of software (language) capabilities and vice versa. However with the increased capabilities in both hardware and software, the development disciplines to deal with increased software requirements and complexities did not evolve as fast (GenerExe, 2002).

In some ways, software development in its early stages can be seen as a triangle composed of sides representing hardware, software and development methodologies. As discussed before, the rapid increases in the capabilities of hardware and software

along with the lack of attention to a methodology for efficiently meeting an intended goal or requirement created an isosceles triangle. Geometrically an isosceles triangle is a triangle which has two equal sides and two equal angles. In this analogy, the shorter side of the isosceles triangle is effectively the development methodologies meant to efficiently accomplish a set of requirements in a timely manner.

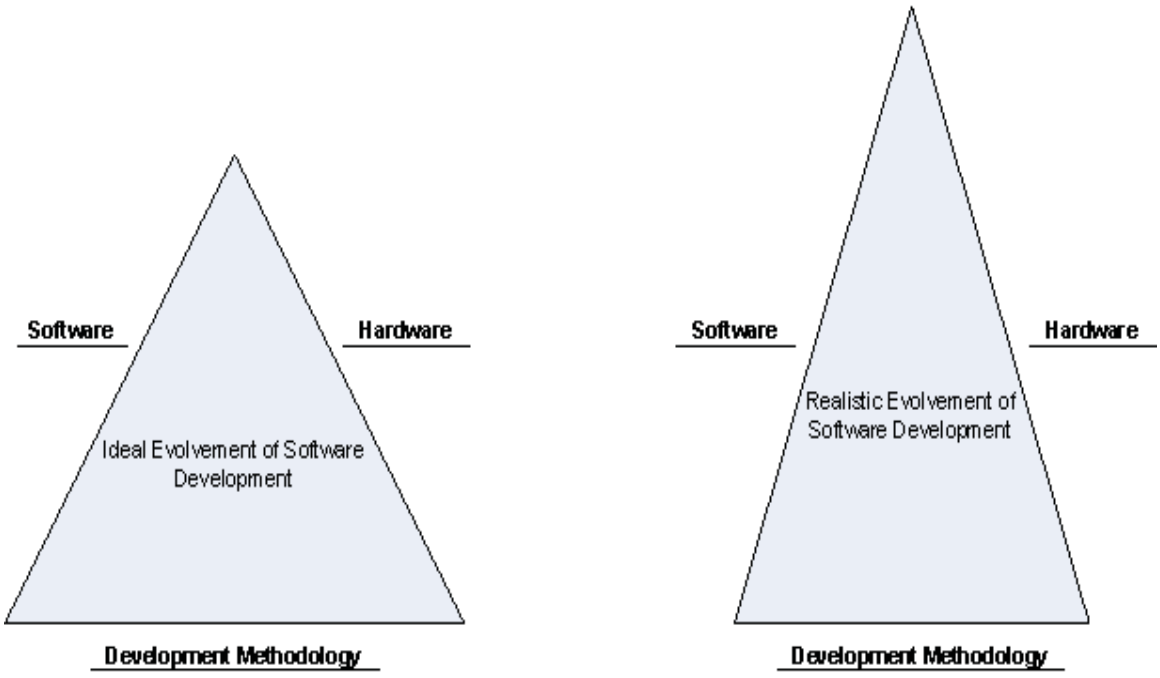

Image 1 (Nesbitt, 2009)

As time has shown, the increased power of hardware and software hardly reflect the signs of great progress (Wirth, 1999). Perhaps the early milestones and growth of hardware and software can be attributed to the constant struggle over developing and delivering expected software within a finite time range. “The increase of power was itself the reason for the terrifying growth of complexity. Whatever progress was made in

software methodology was quickly compensated by higher complexity of tasks" (Wirth, 1999).

The lack of complementary evolvement of development methodologies to that of hardware and software can be seen as an expected outcome in the early begins of software engineering. After all, engineering of all types seems to struggle with increased requests or requirements and time sensitive milestones (Wirth, 1999). Niklaus Wirth considered succumbing to the engineering elements of increased requirements and time pressure results in a "decrease of quality – of reliability, robustness and ease of use. Good, careful design is time-consuming, costly. But it is still cheaper than unreliable, difficult software, when the cost of 'maintenance' is not factored in. The trend is disquieting, and so is the complacency of customers" (Wirth, 1999).

Perhaps the ideal beginnings of software development would have been analogous to an equilateral triangle. However as discussed thus far, that is not the case and development methodologies still struggle to be in synch with technological capabilities and customer requirements. The area of software development in general seems to acknowledge this 'shorter side of the triangle' or weakness. A weakness that has resulted in the potential short coming of unreliable delivery of an expected product. Information technology's numerous attempts to refine development methodologies can be noted in the section that follows.

Though computer hardware and software have evolved, it is interesting to observe how cost is distributed over an entire information technology solution. In the past, a

majority of the cost for a single computing system centered on the amount of hardware needed to effectively run, in today's terms, a relatively simple program. "The overall cost of computer-based systems is associated to the hardware where the system will be deployed and to the cost of the software development and maintenance. The cost of hardware has systematically decreased over the last few decades. Moreover it represents an initial and well-defined fixed cost. This scenario indicates that the main restrictive factor for the development of computer-based systems tends to be the cost of the software" (Guimarães, 2005, p1). In general, Guimarães' statement depicts how the hardware component has become less of a cost, in terms of an information technology solution, than that of the software component. Guimarães' statement does not specifically note whether hardware has actually become cheaper or if its capacity has increased, rather that cost continues to rise faster for developing and implementing the associated complex software. This notable point shows how the efficient development and implementation of software is one of the most important components in the success of an overall information technology solution.

Purpose of the study

This research and analysis focused on the development of a web application and the relationship between two development methodologies and two integration and deployment processes. The background information or research provided a case for the importance of software development methodologies. This case was developed through the discussion of software history. This research performed a brief evaluation

of the traditional and conventional development methodologies, respectively waterfall and agile. Research also involved methods for software code integration and deployment both manually and automated. This project may provide current and future development teams with a perspective on the characteristics of web development projects that best suit the incorporation of certain methodologies and integration and deployment techniques

Limitations of the Study

The following are identified major limitations of this project's research. The use of the term project refers to this paper. The methodology to pair integration and deployment techniques (manual or automated) occurred in a controlled project environment specific to the financial management industry; therefore, it may not be indicative of other software development environments. The sampled methodologies for both traditional and conventional software development do not necessarily permit valid conclusions about the larger population. The four project scenarios observed and analyzed took place in a work and development environment. Due to time constraints and the limited availability of project scenarios, the sampling was limited. The sampled deployment methods for both manual and automated processes do not necessarily permit valid conclusions about the larger population. An attempt to evaluate the development of a web application over a fixed amount of time with similar application requirements may not be indicative of the larger population of software projects. This limitation simply states that not all project requirements (as reflected as a variable in Appendix A) are the same across all projects. The study selected project scenarios that

share common or dependant variables. These variables are not exact and hence the analysis of the results must acknowledge this aspect.

Assumptions of the Study

The following are identified as major assumptions of this research. The projects evaluated for research were web based software applications. The software applications utilized for research were designed and developed in a team environment. A team development environment is defined as a multiple developer, business analyst and business customer based situation. The metrics recorded for the configuration of the integration and deployment techniques are included in the evaluation of these techniques when paired with development methodologies. The selected traditional software development methodology was waterfall. The selected conventional software development methodology was agile. A similar amount of software application requirements, development time and resources were selected in each project research scenario.

CHAPTER TWO

Review of Literature and Research

Introduction

This chapter will document the approach used to study the pairing of software development methodologies and integration and deployment techniques. The research and documentation of these entities aided in the evaluation of the collected metrics.

Chapter Objectives

This review of literature will accomplish the following:

1. Establish the fundamental issue motivating the research of pairing development methodologies and integration and deployment techniques.
2. Develop the criticality and sensitivity of the integration and deployment process in a team development environment.
3. Implement a literature research and review strategy which exams the relative entities and their characteristics as related to the problem statement.
4. Analyze and document data related to the stated problem motivating this research.
5. Develop a research methodology for which there is a justification and rational for producing results which can be evaluated.

Software Mythologies and Deployment Methods

Software development methodologies in information technology (IT) can be seen as a process for satisfying the many aspects of a business request or required need. This process must take into consideration the given resources of personnel, technology and time. Relating back to chapter one's obtuse/equilateral triangle analogies, the methodology side of each triangle can be further seen as utilizing the software, hardware and resource components of IT to produce a well rounded solution. This so called well rounded solution can be seen as a circle that surrounds each obtuse/equilateral triangle. When the triangle is in balance the business solution is supported and could be conceived as well rounded.

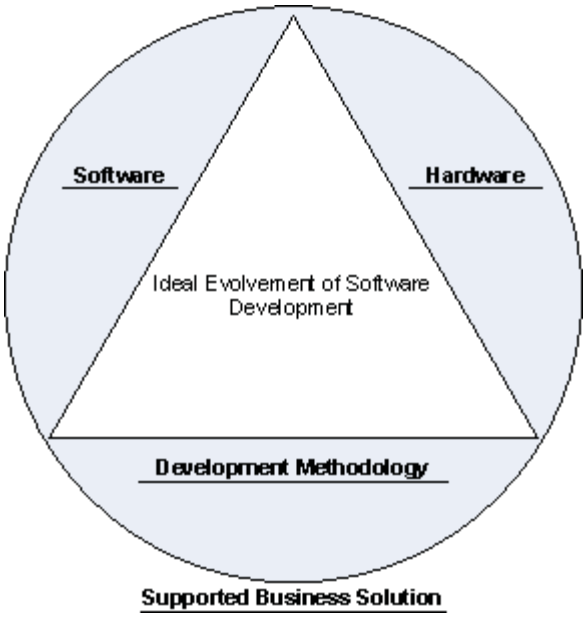

Image 2 (Nesbitt, 2009)

There are many aspects to software development and most of the time certain aspects are overlooked when a project is kicked off or in mid-development. These aspects can include the technology to be used, the business goal or purpose, specific business rules and how the developer understands the required need. The IT industry operated acceptably for many years without a standardized or acknowledged approach to software development. As technologies evolved and requirements became more complex, so did the need for a development framework to give structure to software development. Or another view is the need for reliable development methodologies to provide more balance to the 'triangle of software development.'

Importance of Development Methodology Adoption

Why it is so important for IT organizations to adopt software methodologies? It is clearly evident that a development framework has been historically needed in software development. 15 percent of software projects completely fail and 51 percent of software projects fail to meet the three paramount goals of delivering on time, under budget and meeting customer expectations (SoftwareMag.com, 2004). These are challenging and risky statistics when businesses attempt to under take a software development project. An obvious need exists for a methodology in order to mitigate or lessen these software project risks. A methodology can be seen as a framework, a set of processes or a defined set of rules. These characteristics of a methodology are for the purpose of accomplishing goals and expectations.

In the realm of software development, methodologies are imposed in order provide structure to projects so that they may be delivered on time, under budget and meet business or customer expectations. “Software project management methodologies that have developed in the past couple of decades have done so to address the endemic problem of software project failures caused, in a large part, by lack of planning and poor execution” (Brewer, 2004, p1). Companies and more specifically IT have realized the need for software development methodologies due to the failure rate of past projects. Methodologies attempt to curb the software development epidemic of missed deliveries and not fulfilling customer expectations. There are however a number of different development methodologies which may be applicable to different types of projects.

By addressing these endemic development problems through software methodologies as mentioned by Brewer, it has been possible to realize the benefits of implemented software methodologies. Fitzgerald mentions a number of advantages for the use of methodologies in a software development project. “Methodologies help to cope with the complexity of the software development process. Methodologies reduce risks and uncertainty by rendering the development tasks more transparent and visible. Methodologies may provide a framework for the application of techniques and resources at appropriate times during the development process” (Fitzgerald, 1998, p2). One of the key perspectives of this statement is that methodologies aid in making development tasks more defined or understood. Another key perspective is that with a framework it is possible to gauge when techniques or resources should be applied at a

given point in the development process or processes. The aforementioned time periods in a development framework and the best application of a resource, specifically software code integration, will become clearer later in this paper.

There are a number of software development methodologies which IT project managers can choose from. Some of these methodologies are waterfall, incremental, spiral, sashimi and agile. No single methodology is applicable to any type of software project. The underlying goal of all of these methodologies is to deliver the project on time, under budget and meet customer expectations. It is important to differentiate the traditional software development methodology of waterfall to that of the increasingly popular agile methodology. Though these two methodologies attempt to complete visible development requirements and hence reduce risk, both are different in accommodating changing project variables. Specifically, by comparing these two methodologies it becomes easier to see how time periods (aka. iterations) in the agile development cycle aid in accommodating the implementation of software code integration and deployment.

Traditional Software Development Methodologies

A general explanation of the traditional waterfall development methodology can be seen as collecting all system requirements, foreseeing potential risks and then developing, integrating and deploying the needed system requirements in a set timeframe. “Extensive upfront planning is the basis for predicting, measuring, and controlling problems and variations during the development life cycle. The traditional software development approach is process-centric, guided by the belief that sources of variations are identifiable and may be eliminated by continually measuring and refining processes. The primary focus is on realizing highly optimized and repeatable processes” (Nerur, 2005). The key aspects in this definition by Nerur are that it is possible for a software development team to identify a majority of critical variants, compensate for these variants and better apply these compensations in potential upcoming development phases. While this concept has been shown to be a workable methodology, this tradition just as technology, has had to accommodate increasing variants which effect overall system quality and delivery.

A traditional development methodology or life cycle model dictates the specific tasks and resulting deliverable to come out of each phase. These phases are defined in large chunks such as design, development, quality assurance and deployment. Each of these phases are technically specific and as such are assigned in that manner. “In addition to the end product of working code, these methodologies also produce a large amount of documentation that codifies process and product knowledge” (Nerur, 2005).

Along with the process centric focus of this methodology is the fact that development or resource time is also spent producing documentation. This can have both its attractants and de-tractants. Some of the resulting documentation from this methodology will most likely address the deployment of the software code. This can be seen as a manual task documenting all dependencies on how the software code is to be manually deployed by a human being. A potential for human error can exist in manual task documentation.

An important characteristic of waterfall is the emphasis on completing defined phases of development. These phases must be completed in entirety in order to move on to the next phase of development. Waterfall is a formal top-down development approach. If there is a need to change a software feature developed in a previous phase, a formal and sometimes timely change process must be followed. The issue here is that customer needs may change over the phased development life cycle and hence the waterfall approach could be seen as rigid in accommodating these changing needs (Sorensen, 1995). As a potential consequence of accommodating changing needs, the documented deployment process might be affected. The time to update deployment documentation would also need to be an additional identified variant.

Agile Software Development Methodology

The agile methodology has been gaining more and more popularity over the past six to eight years. In general, agile can be seen as an iterative and evolutionary approach to software development. The development teams are highly-collaborative and operate on a minimal amount of traditional development rules or tasks. These two important characteristics are meant to produce high quality software in a defined time frame. The basic concepts at the heart of agile are that of iterations and releases.

As the agile manifesto mentions, the satisfaction of the customer is met through the continuous delivery of quality software (Highsmith, 2001). The simple approach of agile development differentiates it from some of the more formal processes. Less is more in agile. A big differentiator of agile when compared to the previously mentioned traditional methodology is that it welcomes changes in software requirements. Because the development iterations are short, changes in requirements can be accommodated. It is imperative though that the business (customers) and the development team work together daily on the software project. As mentioned by Nikalaus Wirth in chapter one, engineering of all types seems to struggle with increased requests or requirements and time sensitive milestones. The traditional waterfall development methodology has been mentioned as a top-down almost linear approach to defining requirements and developing them. “Agile methodologies rely on speculation, or planning with the understanding that everything is uncertain, to guide the rapid development of flexible

and adaptive systems of high value" (Nerur, 2005). With the complexities of modern day software development, it is no wonder why the agile software development methodologies have been gaining more and more recent popularity. "Agile methodologies deal with unpredictability by relying on people and their creativity rather than on process" (Nerur, 2005). For that purpose, this project will focus on the metrics and analysis of software code integration and deployments techniques implemented in waterfall and agile development methodologies.

Manual Software Integration and Deployment

At times in the software development arena, technology and processes appear to evolve faster than the final step in the software development life cycle (SDLC). The software deployment process is a crucial step in the SDLC. However though, this step seems to be under addressed. The visualization of a football team getting the football ninety yards down the field with only ten yards to go for the score is analogous to this crucial step in the SLDC. It could appear that the last ten yards of getting the software into the users' hands, fully functional, is an easy distance to travel. This naive reality is compromised by different computing environments and numerous extraneous factors involved in the deployment of software. First of all, in order for the football team to travel those ninety yards of the SDLC, software code produced by individual team members must be integrated or combined.

It is an assumption and a reality that organizational software development most often involves more than one software developer. In order for a development team to efficiently operate without stepping on each other's work, there needs to be the ability for more than one developer to work on the same portion or code of an application. When this need is realized and manual code integration is chosen, software designed specifically for code integration will most likely be utilized. A developer must then decide which sections of code will be combined or disregarded. This can be a time consuming and sometimes flawed process. The overall difference between the software concepts of manual and automated are that manual processes are performed by a developer or server administrator and automated processes are performed by a system or program.

The questions are endless when thinking about the issues that may arise when deploying a new or updated release of software. Some of these questions are as follows: "Will your program only run on your development machine? What is needed to get it up and running on a users system? What about other programs that the user may use? If the user has a different operating system – maybe just a different version – will your software work, too? How do you handle earlier installations of you program" (Weissmann, 2005)?

Dolstra, Bravenboer and Visser's article titles 'Service Configuration Management' emphasizes the important aspects of 'identification' and 'derivation management.' When a software environment, such as a web server or a database machine, and their installed components of compiled assemblies and database scripts

are not all under the control of configuration management, there exists the potential for non-reproducibility. In software deployments, there needs to be the ability for 'identification' or a method for naming the configuration of software components on the associated software environment (Dolstra, Bravenboer, Visser 2005). Along with proper identification, manual software deployments lack the concept of 'derivation management' or the ability to automatically rebuild the software components and deploy them reliably to another software environment (Dolstra, Bravenboer, Visser 2005). Derivation management is based on the concept that "a software service is ideally an automatically constructed derivative of code and data artifacts (Dolstra, Bravenboer, Visser 2005)."

Manual software deployment processes or operations are often times quite difficult. It can become rather time consuming to determine which software components must be installed or copied to a software deployment environment. Also with regards to the chosen software deployment environment is the consideration of any environmental configuration changes for accommodating the components to be installed.

Many software development organizations have pre-production and production environment instances. Pre-production environments can be divided into development, test and stage instances or some combination of each. The management, repeatability and consistency of manual software deployments across these different environments can be challenging. For instance, keeping track of all the identified files to be deployed from a development environment to test and then on to a production environment, along with any environmental configuration changes, over n-number of requested software

deployments can be time consuming. This real-world scenario for manual software deployments can leave an undeniable window for human error. Further more, a manual deployment error has a greater probability to occur at a local environment than consistently across all environments (Dolstra, Bravenboer, Visser 2005). For example if a deployment error were to occur in development and test, the likelihood of the mis-managed deployment could be detected and resolved before it reached a stage and ultimately production environments. This also brings to light the benefits of multiple pre-production software development environments. However, the benefits of multiple software development environments are potentially compromised when there is a lack of repeatability and consistency in the deployment processes.

Automated Software Integration and Deployment

The software development community needs to potentially change its method for software deployments due to the complexity of changing application requirements. Also driving this potential for change is the adoption of more effective development methodologies. “In order to produce a working service, one must typically install a large set of components, put them in the right locations and write configurations” (Ray, 2002). Going forward it will not be practical to deploy and integrate software each time there is an update in system code or an increment in the version. “When information transfers successfully without human intervention it is symptomatic of software integration. What systems integration does is try to reduce overheard costs and effort by making the information flow repeatable, with greater confidence of accuracy and within the timeframe of need” (Ray, 2002).

The traditional and current approaches (manual software integration) to software integration are costly, time consuming, and frequently yield suboptimal results. The characteristics of current system integrations include: They are brittle, i.e. easily fail when faced with slight perturbations to the information transacted (Ray, 2002). They are difficult to maintain as a system evolves, scales or is upgraded (Ray, 2002). Current approaches to software integration are also difficult to scale when the requirement for additional information content or additional constituent systems arise. (Ray, 2002).

Martin Fowler is a renowned author, speaker and architecture on object-oriented analysis, design, development and software development methodologies. He defines continuous software integration as a "software development practice where members of a team integrate their work (code) frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly" (Fowler, 2008). Once a development team has successfully performed the integration process and quality assurance and user acceptance has occurred, the product can be deployed to production.

Markus Weissmann provides a list of steps which he feels covers many potential software deployment issues. Steps 1 through 9 are essentially about getting the deployed software up and running on the users' machine. These steps are as follows:

1. Get the Sources – Provide a URL or DVD-ROM install.

2. Verify the Sources – Verify the source to be downloaded is trusted and secure.
3. Required Patches – Understand your users' install environment and provide potential patches or updates in the source install.
4. Required Build Tools – Pre-compiling the source code into an executable format.
5. Required Libraries and Servers – Install and fully test the compiled executable on servers with the correct libraries which simulate step 3.
6. Parameterization of Build Environment – Automate the executable, patches, libraries etc. into an automated build and deploy process.
7. Required Build Resources – Optimize the build process so that it is as efficient as possible.
8. Detection and Handling of Conflicts – Document and prioritize the build and deployment errors and conflicts so that they are systematically addressed.
9. Installation – Address pre-install areas such as creation of local user accounts, directory or log access for deployment administrators etc. and finally install the software.
10. Upgrade – Either take the software offline so that a successor version can be installed or attempt to concurrently run the predecessor and successor versions simultaneously.
11. Uninstall – Have a strategy to revert back to the previous state or version before the software was installed.

12. Robustness – A general requirement of the software development and deployment process. The process of minimizing the effects of programming errors in the deployment. Steps such as keeping logs of installed or altered tables and data, keeping a backup of system databases all for the event that the environment crashes and needs to be restored. (Weissmann, 2005)

In implementing or considering Weissmann's deployment steps, it's important to acknowledge the value of using an automated build and deployment process. No matter how careful a human can be in deploying binary files, there is still more room for error than if an automated process were to handle the deployment. A human can also observe any issues as a result of the automated deployment process. Also the use of version controlled source code is another helpful technology for a development team to utilize in software development. The use of version control with the integration of an agile development methodology could be an optimal fit for the right software and development team culture.

M. Belguidoum and F. Dagnat take a theoretical approach to automated software deployments. Although their perspectives are theoretical, they provide valuable criteria as to what a deployment process (automated or manual) should encompass. "Administration and deployment of software systems has become increasingly complex. This complexity results from the need for uniform access to applications from heterogeneous terminals through different communication infrastructures" (Belguidoum, Dagnat, 2006). The deployment process is a process which is meant to be carried out

throughout the life cycle of an application. The summarized aspects which aid in supporting automated deployments are:

- Taking into account the evolution of the system (need for autonomy)

- Checking and validating the deployment (need for safety)

- Generality (not concentrated on a particular technology or software medium) so that the deployment approach can be usable in different projects and needs

- Separation of the dependency on other deployment information and intervention (for example, low level deployment mechanisms).

(Belguidoum, Dagnat, 2006)

Getting the components that comprise a web application turned into a running system can often be a complicated process involving compilation, moving files around, loading schemas into the databases, and so on. However like most tasks in this part of software development, they can be automated - and as a result should be automated. Asking people to type in strange commands or clicking through dialog boxes is a waste of time and a breeding ground for mistakes (Fowler, 2008). Automated environments for builds are a common feature of systems. The Unix world has had this for decades, the Java community developed Ant, the .NET community has had Nant and MSBuild and, as utilized in this study, Cruise Control. The automated integration and build technology, Cruise Control, has been developed by ThoughtWorks Studios. With every subsequent build and deploy of code to a development or test environment, Cruise Control makes it possible to configure the automated process so that it aligns with the actual production deployment process.

Configuration Management

Software development projects involve lots of files that need to be orchestrated together to build a product. Keeping track of all of these is a major effort, particularly when there's multiple people involved. So it's not surprising that over the years software development teams have built tools to manage all this. These tools - called source code management tools, configuration management, version control systems, repositories, or various other names - are an integral part of most development projects (Fowler, 2008).

In development methodologies, whether waterfall or agile, software requirements and code evolve over time. Hence new phases or versions of the code become available with the release of new functionality or defect resolutions. Perhaps there are configuration changes or patches in the environment where the system resides and as a result the environment needs to be restarted. Regardless of an intentional deployment of system code or an unintentional one due to environment changes, such deployments need to be easily accessible. By utilizing a version control system, it is possible to access software code at any point within its life cycle.

Test Driven Development

Test driven development is a concept that test code is developed upfront. Test driven development is popular and can commonly be found in agile development communities. Hence, the incorporation of test driven development into agile development releases aids in continued testing as software changes. Test driven development is further realized when compounded up unto a production release. Test

driven development (TDD) is motivated by the fact that thinking about and writing a test prior to coding will make the code more understandable and maintainable“(Nerur, 2005). TDD is comprised of a number of unit tests. A unit test can be seen as a singular instance testing a specific piece of software code such as retrieving a user’s account information from a database or saving the update of a user’s account information. A good way to catch bugs more quickly and efficiently is to include automated tests in the build process. Testing isn't perfect, of course, but it can catch a lot of bugs - enough to be useful. In particular, the rise of Extreme Programming (XP) and TDD has done a great deal to popularize self-testing code and as a result many people have seen the value of the technique. Martin Fowler makes the following point. “Regular readers of my (Fowler) work will know that I'm a big fan of both test driven development and XP, however I want to stress that neither of these approaches are necessary to gain the benefits of self-testing code. Both of these approaches make a point of writing tests before you write the code that makes them pass - in this mode the tests are as much about exploring the design of the system as they are about bug catching.” (Fowler, 2008). For purposes of this study, TDD was applied and implemented in all testing scenarios regardless if the development methodology was traditional or agile.

CHAPTER THREE

Research Methodology

Introduction

This chapter will discuss the quantitative methodologies utilized in the analysis of the four project evaluation scenarios. These scenarios will be explained and along with the rational behind why these methodologies were selected. The use of a component was utilized in the collection of metrics associated with the quantitative methodology. This component will be described along with the rational behind its design. Documented in this chapter are the four project evaluation scenarios for which the evaluation component was applied. Associated with the four evaluation scenarios is a discussion, both literally and graphically, on how the quantitative methodology will incorporate one of the two integration and deployment processes.

Methodologies

A quantitative methodology was selected for the evaluation of the four project scenarios. The quantitative approach is definitive in the evaluation categories selected (Appendix A). The metrics collected by the component in Appendix A were useful in the analysis, conclusion and possible future recommendations.

A quantitative methodology involved the collection of numeric numbers based upon the observation of each of the four evaluation scenarios. Numeric numbers were generated based on various evaluation categories. These evaluation categories have been compiled into the component in Appendix A. The creation of the component's

evaluation categories originated from material about traditional or agile development methodologies. The terms and component categories used were meant to be agnostic of methodologies and common to both traditional and agile. For example, the available number of developers and testers related directly to both methodologies' 'time to available resources'' aspect. When the available development time aspect was considered, the term child development phase was selected. Child development phases related to agile development by the number of iterations rolling up to a major release. For traditional development, child development phases were seen as the number of development life cycles a project underwent in order to reach the intended end-goal at project kick-off.

The component's section on the number of planned and unplanned requirements and hours were meant to capture the initial estimate provided for a requested or required portion of system functionality. This requested system functionality can come in the form of a user story for agile development or simply as a business requirement for traditional development. In the development of the research component, it became clear that the inclusion of the initial estimate to that of the realize effort was valuable. A methodology's ability to accommodate inaccurate estimates could prove to be useful in real world software development. The manual or automated integration and deployment technique category is simply meant to clarify which evaluation scenario the component applied to. Included in the metric for the number of planned software requirements is the configuration for either manual or automated deployments. Overall the component collected metrics for the four evaluation scenarios based upon a similar number of project characteristics (available development and testing hours, number of

requirements, number of available development days etc.). All metrics collected are displayed in Appendix B, C, D and E and originated from a project tracking tool.

As mentioned, four different scenarios were evaluated based upon two different methodologies and two different types of integration and deployment techniques. Similar projects with similar requirements were used as the foundation for collecting metrics with the component in Appendix A. The four different project scenarios that were evaluated are as fallows:

1. A traditional development methodology utilizing no automated integration and deployment processes
2. A traditional development methodology implementing an automated integration and deployment technology
3. An agile development methodology utilizing no automated integration and deployment processes
4. An agile development methodology implementing an automated integration and deployment methodology

The below visuals are intended to depict at a high level how manual and automated software integration and deployment techniques were incorporated into application development. The visuals are methodology agnostic. The integration and deployment specifics with regards to the selected methodology will be discusses later.

Manual Software Integration and Deployment

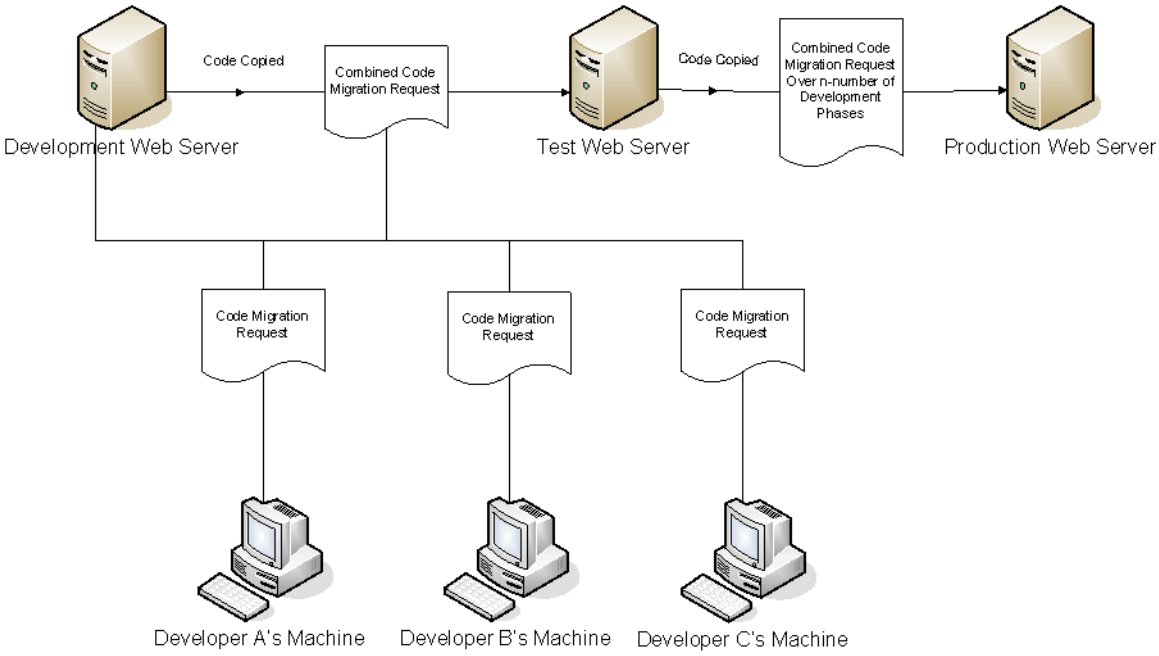

Image 3 (Nesbitt, 2009)

The above diagram titled Manual Software Integration and Deployment depicts a software development scenario in which three developers desire to have newly developed code for the same system deployed to a development web server. In order for this to simultaneously take place, the code individually developed by A, B and C must be integrated or merged. This could be a manual process which occurs on one of the developer' machines, on an integration server or less advantageously on the development server. Regardless of the architectural challenge of developer code integration, is the overall manual process of initializing an integration and deployment request, the time involved to complete such a process and the integrity of that process. The diagram further illustrates that once the code has been successfully integrated onto the development web server; there still exists the manual process of deploying or migrating the system code to the test web server. Furthermore, the diagram shows that

the system code on the test web server must be manually deployed to the production web server.

Automated Software Integration and Deployment

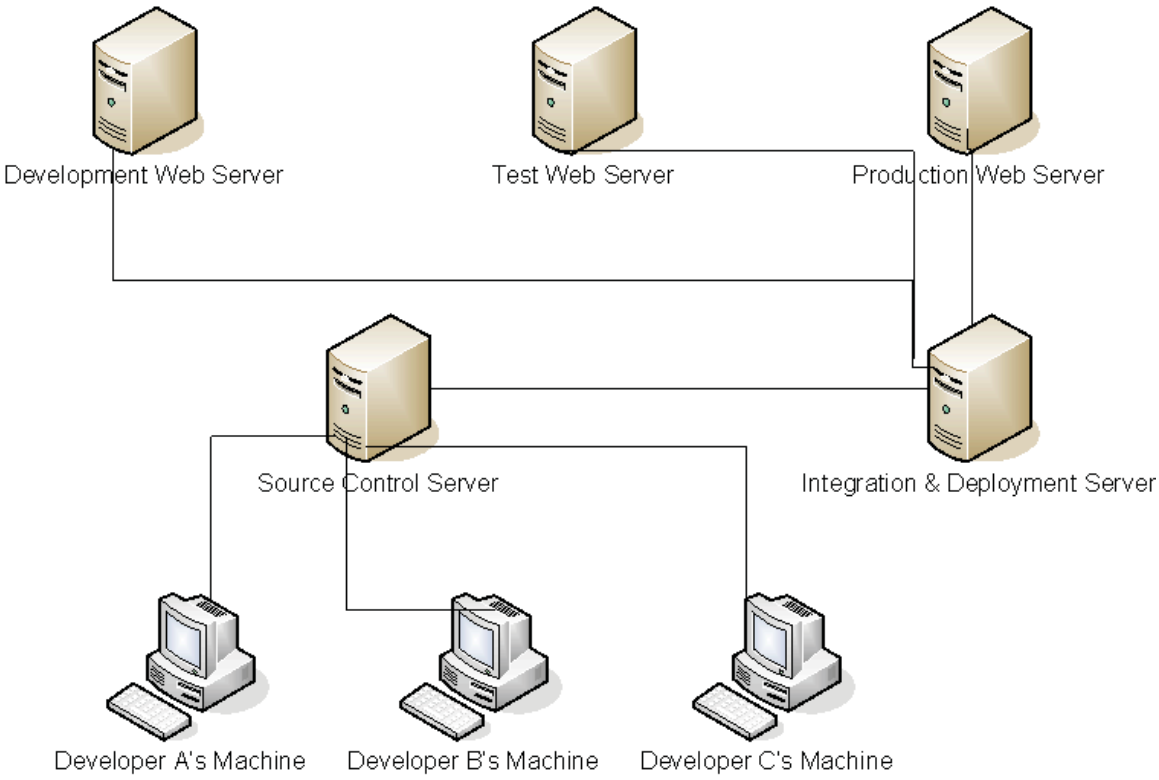

Image 4 (Nesbitt, 2009)

The above diagram titled Automated Software Integration and Deployment depicts a software development scenario in which each developer submits or check-ins newly developed code for the same system. The source control server in this diagram can be seen as a library for maintaining the historical and present state of the system code. In order for Developer A to add file XYZ to the system code baseline, it must be checked-in or registered by the source control server. Once a check-in takes place, the file XYZ will be available to Developers B and C. The Integration & Deployment Server

component of the diagram plays a critical role in the sense that it periodically checks the Source Control Server to verify if any new files have been added or whether any existing files have been changed. If the verification by the Source Control Server is affirmative, the Integration & Deployment Server has the ability to retrieve the latest system code, compile and deploy the code to the Development Web Server. Furthermore, the diagram depicts the ability for the Integration & Deployment Server to compile and deploy system code from the Source Control Server to the Test and Production Web Servers. The most notable aspect of the process depicted is that an automated integration and deployment process can potentially increase the integrity and reliability of correctly integrating and deploying system code to various environments.

Each of the above development scenarios has potential pros and cons. These pros and cons were realized and became more obvious when applied to a software development methodology. The following sections discuss how the preceding integration and deployment techniques will be incorporated into the agile and traditional development methodologies. The incorporation of these integration and deployment techniques with the methodologies created the evaluation scenarios. These evaluation scenarios were then coupled with the study's selected web/internet projects.

Incorporation of Integration and Deployment Technologies in Agile

This section will discuss the application of integration and deployment techniques in an agile development methodology. Scott Ambler presents some helpful agile integration steps for aiding in the complexities of software integration and deployment. However Ambler's integration perspective is applicable to both manual and automated

integration and deployment processes. He suggests a development team needs to identify and understand its deployment audience (Ambler, 2005). Additionally a deployment administrator should document the release notes or process as a requirement in an agile development iteration (Ambler, 2005). Based on this suggestion, each evaluation scenario in this study created a requirement (in traditional methodology terms) and a user story (in agile methodology terms) for integrations and deployments in the system life cycle. This study also aligned to Ambler's suggestion that pre-production and production environments can be utilized to efficiently develop and test software in an agile methodology. Ambler's suggestion was based on the idea that multiple environments helped software integrations and deployments go as smooth as possible (Ambler, 2005).

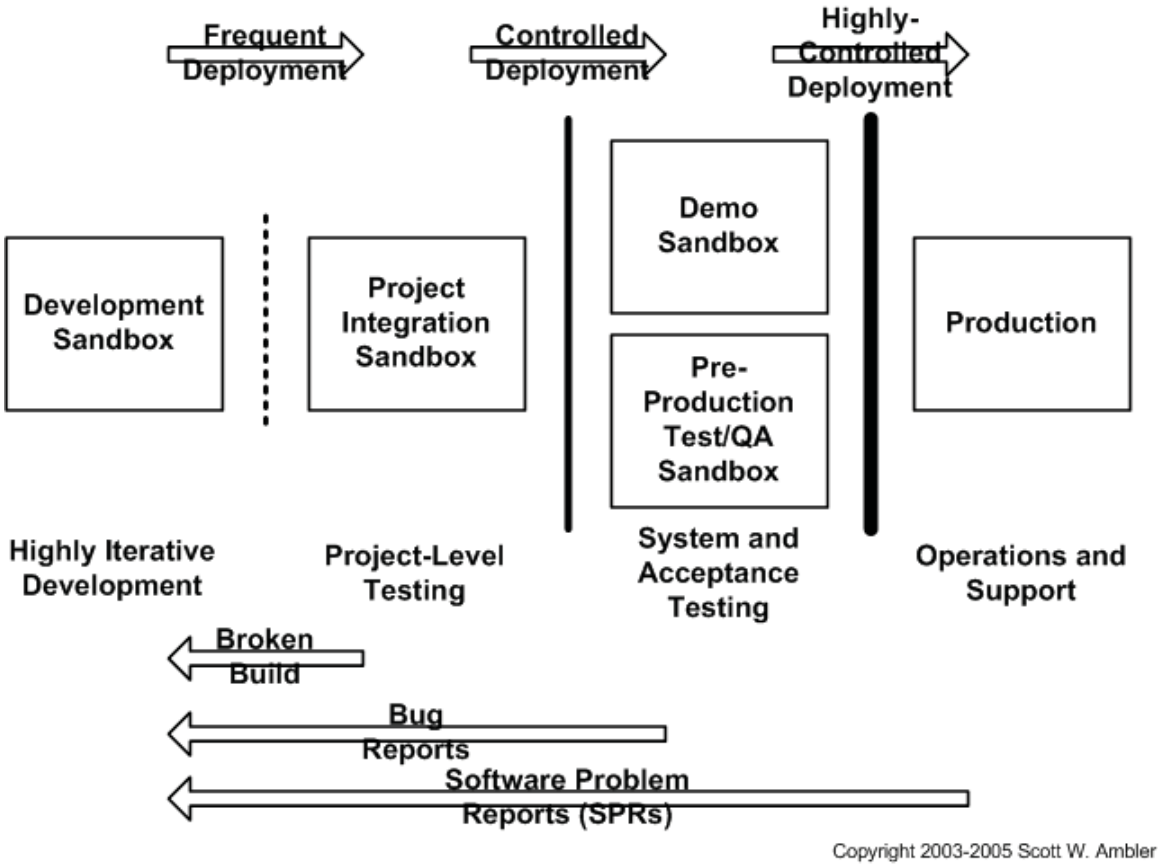

Image 5 (Ambler, 2005).

The above diagram published by Ambler is ideal in depicting how software code can be integrated and deployed in a multi-environment scenario. Each vertical line in the diagram beginning on the left with a dashed line and becoming more bold illustrates the increase in fine tuning the integration and deployment process. This diagram is ideal and applicable to this study's methodology because it is agnostic of integration and deployment processes. Each of the four evaluation scenarios implemented the above process flow in a multiple server environment.

This study also implemented an agile directive and Ambler's suggestion of releasing regularly in a development environment and having a milestone like production release. The execution of the software integration and deployment process in the development environment whether manual or automated resulted in the production of an artifact. This artifact could be a documented deployment plan or an automated build and deploy script. Proper planning and user and environment research was paramount in mitigating deployment issues. By having development releases, the hope was that potential deployment conflicts, as mentioned in Weissmann's step 8, can be realized and addressed before the production release. Development releases could be seen as a litmus test for a true production release. Hence, agnostic of the integration and deployment process used for the release to the development environment, the process was seen to become more tuned through the resolution of potential integration and deployment issues. The below diagram by Ambler illustrates the suggestion followed by this study of regularly releasing to development environments followed by a milestone production release.

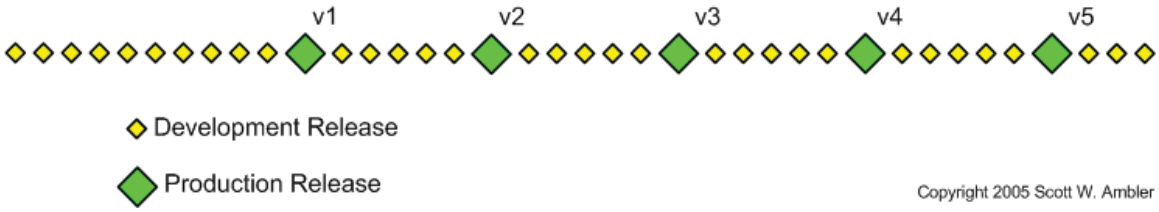

Image 6 (Ambler, 2005).

CHAPTER FOUR

Analysis and Lessons Learned

Introduction

This chapter will document the major findings of this study and analyze the data collected from chapter three. This chapter will specifically look at the metrics collected via the component (in Appendix A) from the four evaluation scenarios.

Analysis of Quantitative Results

The metrics collected and represented via the study's component originated from the project planning tool Version One. Version One was utilized for the tracking and project management aspect of web-based financial software applications. The four evaluation scenarios occurred via the development of web-based software applications over a four month time period. The following content will discuss the commonalities and differences of the metrics recorded with each scenario's component.

In quantitative research, the aim is to determine the relationship between one thing (an independent variable) and another (a dependent or outcome variable) in a population (Hopkins, 2008). Quantitative research designs are either descriptive (subjects usually measured once) or experimental (subjects measured before and after a treatment). A descriptive study establishes only associations between variables. An experiment establishes causality" (Hopkins, 2008).

This study's quantitative research is of the descriptive type. Each of the study's four scenarios was observed once and there was no attempt to change the conditions or independent variables and re-observe the outcome variables. Perhaps one aspect that could be further addressed in the study's qualitative metrics is the examination of the greater population. However, an examination of the greater population would need to be selective. Selected projects would need to align with the characteristics of the study's four scenarios. Also, these additionally selected projects would need to have common dependant variables as shared between the study's four scenarios. The independent and dependent variables of this study's scenarios must be noted in order to discuss the analysis of each scenario's results.

The metrics for each of the study's evaluation scenarios can be viewed in Appendix B, C, D and E. Over four months of development, an attempt was made to select project development phases which shared similar dependent variables. Ideally it would have been paramount to have all the dependent variables be the same across the four observed scenarios. However, this was a difficult aspect to obtain from the perspective of real world development.

The dependent variables reflected in the component in Appendix A were number of developers, number of testers, available development time, number of software requirements and number of requirements requiring a full regression test. Though the development and testing resources to the number of software requirements to be developed in an available development timeframe varied, the ratio of the available

development time to requirements and resources was comparable in all four scenarios. The component variables, approximate development and testing hours and available development time aided in the calculation of this ratio. The ratio or average for the available planned resource hours to the available development time was about 45.75 hours per day. This metric identified the average number of total resource hours that could be committed per day towards completing the scenario's software requirements. It was then foreseeable that a limited number of 'unplanned development and testing hours' could be accommodated in each scenario.

The independent variables in each of the scenarios can be seen as indirect results of the dependent variables and their involvement in the chosen development methodology and integration and deployment technique. The independent variables with regards to the component are the integration hours, deployment hours, whether the deployment was successful or unsuccessful, satisfaction of the business customer and on time delivery of the software requirements. The most notable of these independent variables are the integration and deployment hours and the overall success of the deployed software. Much of the quantitative results analysis will focus on these three variables.

Analysis of Automated and Manual Integration

Beginning with the software integration variable, it can quickly be observed that between two scenarios the manual method of integrating and deploying software was

30 and 35 hours. The hours recorded for manual integration (use of an integration software tool) also included documentation for each build to a pre-production environment and the actual time spent executing the deployment document. The two metrics recorded for automated integration scenarios was 40 and 50 hours. The hours recorded for automated integration included the initial and any ongoing configuration that took place over the available development days. It was observed that though the automated integration hours were higher than manual integration, the initial configuration and later adjustments built upon on themselves. It was realized that with each automated run of the integration and deployment technology to a pre-production environment, the process became more efficient and hence more reliable and repeatable by a non-human system. This incremental improvement of the repeatable non-human process is most evident in the independent software deployment variable.

It should be clarified that the component's software integration hours means the time it took to deploy the final code base to the production environment. The deployment metric recorded for manual integration was 12 and 18 hours. This was about 18.75% higher than the time spent deploying software code resulting from automated integration and deployment. The deployment times for the automated scenarios were 3 and 5 hours. The difference in the number of hours spent deploying the software seemed to be attributed to the manual execution of a detailed deployment document. In both manual deployment scenarios, unexpected or faulty application behavior was discovered in the post-deployment test phase. Once the faulty behavior was communicated, the deployment document was examined and it was realized certain

steps were incorrectly performed. These steps were re-executed which in turn increased the deployment time and triggered another post-deployment test phase.

Analysis of Project Delivery

In all four of the study's scenarios, the software deployment to production was considered a success and the software customer was satisfied. Customer satisfaction was attained regardless of the fact that three of the four project scenarios deployed to production in upwards of 5 days late. The only on-time delivery of software to production resulted from the scenario in which an agile methodology was utilized in conjunction with a manual software integration and deployment process. The successful delivery of 17 software requirements (user stories in agile) can be attributed to both the selected methodology and the 13 manual integration and builds in the pre-production environments. This scenario's use of an agile methodology forced the development team to continually integrate and deploy incrementally to pre-production components. The documentation of incremental manually deployed code may have resulted in a more refined deployment document. This refined deployment document was partially realized in the 12 hour deployment variable.

On the contrary, the project scenario that was delayed the most in delivery by 5 days was a traditional methodology paired with a manual integration and deployment process. It could be assumed that the 35 hours recorded for 7 code integration and pre-deployment builds was not repetitious enough to fully document all deployment

nuances. These nuances potentially resulted in 18 hours of recorded deployment time. This is 6 deployment hours more than the project scenario that used an agile methodology with a manual integration and deployment process. One aspect to note in the comparison of these two scenarios (traditional methodology with manual vs. automated) is that the agile methodology required 6 more manual integration and pre-production builds which resulted in less total hours for both the software integration and deployment variables. It could be hypothesized that there are some efficiencies gained from the consistent manual deployment of software to pre-production environments over an available development timeframe.

Analysis of Automated and Manual Deployment

Strictly looking at the differences between manual and automated integration and deployment, regardless of methodology, provides a valuable perspective. The project scenarios that implemented an automated deployment process were almost 400% faster than the manual process. The related automated integration process was around 7% longer than the manual integration process. To reiterate, the automated integration and pre-production deployment process included the configuration of the technology which was also indirectly realized in the software deployment to production variable. It was not recorded how many automated builds resulted once the configuration was completed. This may have been an interesting variable to collect as it would then provide the basis to amortize each integration and pre-production deployment instance. Aside from that aspect, once automated, an integration and pre-production deployment could occur on a daily basis. The question then might arise, is it worth a 7% longer investment in configuration time for a 400% gained efficiency. This question will be referred to going forward as the '7 for 400 question.' The 7% increase is simply seen in the configuration of the technology and does not include the cost of the technology's hardware and software.

To further aid in answering the '7 for 400 question', the project scenario that implemented a traditional methodology with an automated integration and deployment process presents an interesting realization. This realization was that 50 hours were spent configuring and then executing the automated technology. The realization was

not that this scenario took 10 hours longer than the agile methodology based scenario but that by nature, a traditional methodology does not dictate a continuous integration and pre-production deployment edict. Given this realization, the motivation (often times brought on by methodology) to utilize automation in traditional methodology is less than that of a project based in an agile development methodology.

Miscellaneous Observations

The utilization of a version control system along with automated builds and unit tests provided a historical perspective on the code base. The availability of historical versions of source code may be useful in understanding how the software was at a particular point in time. The automation of pre-production and production builds when utilizing a version control system also provided the development team with audit trail capabilities. The version deployed on a pre-production or production server may also provide an audit trail on specific changes logged within the version control instance. The automation of unit tests was further realized and more effective when implemented in an automated integration and deployment process. Automated unit tests and integration technologies could potentially provide the development team with the ability to know when a change had negatively impacted the code base.

CHAPTER FIVE

Conclusion

Concluding Remarks

This paper's research, methodology implementation, component conceptualization and analysis of quantitative results led to many beneficial findings. The thesis statement within the statement of the problem stated that the leveraging of automated integration and deployment technologies in agile development projects can result in more consistent and reliable delivery of web based software applications. It is the conclusion of this thesis paper that this is a true statement.

The support for this conclusion came from the analysis of the results from the selected methodology. One caveat which was mentioned in the research methodology chapter was the application of the component in Appendix A to more web-based software development projects aligning to the four project scenarios. A larger selection pool may have strengthened the conclusion to the thesis statement. However, this aspect of the research methodology was limited due to time.

The lesson learned primarily supporting the conclusion is that of the '7 for 400 question.' Answering yes to the question of, is it worth a 7% longer investment in configuration time for a 400% gained efficiency in pre-production and production deployments shows that automation is highly beneficial to web-based software development projects. Answering yes to the '7 for 400 question' is only part of the conclusion though. When automated integration and deployment is utilized in a web-based agile developed project, it is the most effective combination of the four project

scenarios. As the analysis showed, automation in a traditional web-based development project is also beneficial. However, as a result of the lessons learned, the implemented methodology had a direct effect on the actual use of automation. In an agile development methodology, automation of integration and deployment was more of a benefit because continuous integration and deployment are foundational practices. In a reuse centered technology industry, the reuse of automation has more of a place and benefit in agile methodology. Plus there is a dividend for implementing automation in agile projects. The iterative development and refactoring aspects of agile provide for more potential cost justification and amortization of the technology and its configuration.

Appendix A -

Original Development Methodology to Integration/Deployment Method Evaluation Form

Development Methodology:	
Integration & Deployment Method:	
Number of Developers:	
Number of Testers:	
Number of Business Customers:	
Available Development Time:	
Child Development Phases:	
Days per Child Development Phase:	
Number of Software Requirements:	
Number of Requirements Requiring Full Regression Testing:	
# of Unplanned/New Requirements:	
Percent Completion of Planned vs. Unplanned Hours:	
Total Variance (Planned vs. Unplanned Hours):	
Planned Development Hours:	
Unplanned Development Hours:	
Planned Testing Hours:	
Unplanned Testing Hours:	
Software Integration Hours:	
Software Deployment Hours:	
Successful or Unsuccessful Deployment:	
Satisfaction of Business Customer:	
Software Requirements Delivered On Time:	

Appendix B -

Development Methodology to Integration/Deployment Method Evaluation Results for Evaluation Scenario 1

Development Methodology:	**Traditional** (Hybrid of Waterfall)	
Integration & Deployment Method:	**Manual**	
Number of Developers:	**5**	
Number of Testers:	**2**	
Number of Business Customers:	**2**	
		Comments
Available Development Time:	**22 days**	Work days
Child Development Phases:	**n/a**	
Days per Child Development Phase:	**n/a**	
Number of Software Requirements:	**16**	
Number of Requirements Requiring Full Regression Testing:	**9**	
# of Unplanned/New Requirements:	**2**	
Planned Development Hours:	**Approx. 520**	
Unplanned Development Hours:	**Approx. 25**	
Planned Testing Hours:	**Approx. 380**	
Unplanned Testing Hours:	**Approx. 30**	
Software Integration Hours:	**35**	approx. 7 Dev Builds
Software Deployment Hours:	**18**	Producing 1 Deploy Document from Dev Build Docs
Successful or Unsuccessful Deployment:	**Successful**	1 requirement omitted
Satisfaction of Business Customer:	**Satisfied**	More development needed
Software Requirements Delivered On Time:	**Delayed by 5 days**	

Appendix C -

Development Methodology to Integration/Deployment Method Evaluation Results for Evaluation Scenario 2

Development Methodology:	**Traditional** (Hybrid of Waterfall)	
Integration & Deployment Method:	**Automated**	
Number of Developers:	**6** (4 full time @ 6 hrs/day, 2 half time)	
Number of Testers:	**3** (3 full time @ 6 hrs/day)	
Number of Business Customers:	**2**	
		Comments
Available Development Time:	**24 days**	Work days
Child Development Phases:	**n/a**	
Days per Child Development Phase:	**n/a**	
Number of Software Requirements:	**20**	
Number of Requirements Requiring Full Regression Testing:	**13**	
# of Unplanned/New Requirements:	**4**	
Planned Development Hours:	**approx. 690**	
Unplanned Development Hours:	**approx. 15**	
Planned Testing Hours:	**approx. 410**	
Unplanned Testing Hours:	**approx. 30**	
Software Integration Hours:	**50**	Config of Automated Integration/Build Included As Requirement
Software Deployment Hours:	**5**	
Successful or Unsuccessful Deployment:	**Successful**	2 Requirements Omitted from Release
Satisfaction of Business Customer:	**Satisfied**	More development needed
Software Requirements Delivered On Time:	**Delayed by 3 days**	Test Deployments with Build Tool

Appendix D -

Development Methodology to Integration/Deployment Method Evaluation Results for Evaluation Scenario 3

Development Methodology:	**Agile**	
Integration & Deployment Method:	**Manual**	
Number of Developers:	**5** (4 full time @ 6 hrs/day, 1 half time)	
Number of Testers:	**3** (3 full time @ 6 hrs/day)	
Number of Business Customers:	**1**	
		Comments
Available Development Time:	**20 days**	Work days
Child Development Phases:	**2**	2 Ten day sprints
Days per Child Development Phase:	**10**	
Number of Software Requirements:	**17**	
Number of Requirements Requiring Full Regression Testing:	**12**	
# of Unplanned/New Requirements:	**2**	
Planned Development Hours:	**approx. 505**	
Unplanned Development Hours:	**approx. 10**	
Planned Testing Hours:	**approx. 320**	
Unplanned Testing Hours:	**approx. 15**	
Software Integration Hours:	**30**	approx. 13 Dev Builds
Software Deployment Hours:	**12**	Producing 1 Deploy Document from Dev Build Docs
Successful or Unsuccessful Deployment:	**Successful**	
Satisfaction of Business Customer:	**Satisfied**	
Software Requirements Delivered On Time:	**Yes**	

Appendix E -

Development Methodology to Integration/Deployment Method Evaluation Results for Test Scenario 4

Development Methodology:	**Agile**	
Integration & Deployment Method:	**Automated**	
Number of Developers:	**6** (4 full time @ 6 hrs/day, 2 half time)	
Number of Testers:	**3** (3 full time @ 6 hrs/day)	
Number of Business Customers:	**2**	
		Comments
Available Development Time:	**20 days**	Work days
Child Development Phases:	**2**	2 ten day sprints
Days per Child Development Phase:	**10**	
Number of Software Requirements:	**16**	
Number of Requirements Requiring Full Regression Testing:	**11**	
# of Unplanned/New Requirements:	**3**	
Planned Development Hours:	**approx. 690**	
Unplanned Development Hours:	**approx. 25**	
Planned Testing Hours:	**approx. 410**	
Unplanned Testing Hours:	**approx. 10**	
Software Integration Hours:	**40**	Config of Automated Integration/Build Included As 1st Sprint Story
Software Deployment Hours:	**3**	
Successful or Unsuccessful Deployment:	**Successful**	1 Requirements Omitted from Release
Satisfaction of Business Customer:	**Satisfied**	
Software Requirements Delivered On Time:	**Delayed by 2 days**	Test Deployments with Build Tool

Appendix F -

Images presented throughout the research paper.

ANNOTATED BIBLIOGRAPHY

1.) - Gray, J.P., Ryan B. (1997).Applying the CDIF Standard in the Construction of CASE Design Tools. *Australian Software Engineering Conference (ASWEC '97)*. *1*, 88.

Gray and Ryan's article different integration development approaches to the construction of CASE designs for relational database development. The article shows various illustrations of the real life implemented CASE designs for tables, sequences, stored procedures, triggers, functions etc. This article could be helpful in the illustration of the design of not only course specific coding for procedures, functions and packages but also for later documentation in the MSCI program capstone project.

2.) –
Liu, Yu David (2006).A formal framework for component deployment. *Conference on Object Oriented Programming Systems Languages and Applications Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. *1*, 325-344.

David Yu Liu discusses the fundamental principles behind deployment unit design. The article shows how established technological frameworks such as Java, .net and COBRA have deployment procedures which are specific to the respective technology. The article discusses the deployment components of when, where, and how. Also covered in the article are the deployment aspects of versioning, version dependencies and hot-deployment of components. This article is especially applicable to the course writing assignment of deployment issues and the release of code to clients.

3.) –
Jansen, Slinger (2006).Evaluating the release, delivery, and deployment processes of eight large product software vendors applying the customer configuration update model. *International Conference on Software Engineering Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research*. *1*, 65-68.

Slinger's article focuses on the fact that software vendor's release or deployment process is lengthy and complex. Due to the complexity of the release process, the room for error ultimately affecting the customer is highly probable. Slinger proposes a model (customer configuration updating) (CCU) which could potentially help vendor's reduce the negative effects of deployment issues. The CCU proposes involving customers in the release, delivery and deployment processes to improve product quality and experience. This article is especially applicable to the course writing assignment of deployment issues and the release of code to clients.

4.) –
Veranga, Leo (2008, January 22). 10 Benefits of Using Coding Standards to Software Development Team. Retrieved January 22, 2008, from Articlesbase Web site: http://www.articlesbase.com/programming-articles/10-benefits-of-using-coding-standards-to-software-development-team-312610.html

Leo Veranga's article provides 10 notable benefits for coding standardization within an organization. These benefits are directed towards those on a software development such as the developers, quality assurance personnel and project managers. The main noted benefit is around the ease of code maintenance as the application or code base progresses in its lifetime. This article will be helpful for providing support for the coding standards in an organization paper.

5.) –
Brewer, Jeffrey L. (2005).A Study of Software Methodology Analysis: "Great Taste or Less Filling". Department of Computer and Information Technology Purdue University. 1, 2.

Jeffery Brewer's article provides a critique on the traditional software development methodologies such as waterfall, sashimi and spiral in comparison to agile development. He provides the basis for the need for a development methodology. In his critique of traditional methodologies he shows how traditional ones are not always applicable to certain types of software projects. He provides a case with ample support for how an agile development methodology could possibly aid an organization in accomplishing the challenging project goals of delivering on time, under budget and satisfying customer expectations. This article is applicable for not only this course's look at development methodologies but also for capstone reference material.

6.) –
Landre, Einar (2007).Agile enterprise software development using domain-driven design and test first. *Conference on Object Oriented Programming Systems Languages and Applications Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*. *1*, 983-993.

This journal article focuses on the development and application experience of using the agile development methodology in the software department at a large oil and gas company in Norway. The methodology was used in the altering of object oriented applications, domain driven design and Oracle databases. The article goes into detail on the changes made at the data layer and the concerns around implementing business logic through PL/SQL. The article also covers the IT department's experience with a test first design techniques. It focuses on not only agile but the aspects of test first and business logic through PL/SQL.

7.) –
Tuohey, William (2002).Benefits and Effective Application of Software Engineering Standards. *Software Quality Control*. *10*, 47-68.

Wiliam Tuohey's article supports the enforcement of software coding standards. He shows the benefits not only through the documentation of them but also of the ill effects of not having coding standards. The article documents the negative costly effects of bad or non-existent coding standards. This article will be helpful in the writing of the coding standards paper.

8.) –
Davey, Mark (2006, December 1). Outsourcing Strategies - Catering For A Refined Palate. *The Banker*, *1*

Mark Davey's article discusses the evolution of outsourcing when becoming involved with financial companies in the United States. Davey's article illustrates how companies in general are demanding more of outsourcing companies and especially those which are offshore. The article provides examples of how the past practicalities of off shoring were mainly for cost cutting and tactical responsibilities. Now, outsourcing to offshore must also include non-technical skills and cultural adaptability.

9.) –
Panchak, Patricia (2006, July). Next-Generation Outsourcing. *Industry Week*, 11.
Panchak's article provides general definitions and terminology when examining the outsourcing and off shoring services available to onshore companies.

10.) –
Klopper, R., Gruner, S., and Kourie, D. G. 2007. Assessment of a framework to compare software development methodologies. In *Proceedings of the 2007 Annual Research Conference of the South African institute of Computer Scientists and information Technologists on IT Research in Developing Countries* (Port Elizabeth, South Africa, October 02 - 03, 2007). SAICSIT '07, vol. 226. ACM, New York, NY, 56-65. DOI= http://doi.acm.org/10.1145/1292491.1292498

11.) - Jiang, L. and Eberlein, A. 2008. Towards a framework for understanding the relationships between classical software engineering and agile methodologies. In *Proceedings of the 2008 international Workshop on Scrutinizing Agile Practices Or Shoot-Out At the Agile Corral* (Leipzig, Germany, May 10 - 10, 2008). APOS '08. ACM, New York, NY, 9-14. DOI= http://doi.acm.org/10.1145/1370143.1370146

12.) - SoftwareMag.com (2004). Project success rates improved over 10 years. Retrieved September 6, 2004, from http:// www.softwaremag.com/L.cfm?Doc= newsletter/2004-01-15/Standish

13.) - Sorensen, Reed (1995, January). A Comparison of Software Development Methodologies. Retrieved January 13, 2008, from STSC Crosstalk Web site: http://www.stsc.hill.af.mil/crosstalk/1995/01/Comparis.asp

14.) - Ambler, Scott (2007, April 16). System Deployment Tips and Techniques. Retrieved February 22, 2008, from Ambysoft Web site: http://www.ambysoft.com/essays/deploymentTips.html

Ambler's article discusses 23 suggested tips on successfully deploying software along with integrating these tips into the agile development methodology. The System Deployment Tips and Techniques article strongly conveys the need for planning for the system deployment. Planning can occur through the use of simulated development releases in preparation for the regular agile production deployment. The article also depicts how a development and quality assurance environment should be architected to support such a deployment simulation and preparation.

15.) - Weissmann, Markus (2005).Software Deployment. *Georg Simon Ohm University of Applied Sciences*. *1*, 1-11.

Weissmann's article outlines the issues that arise from poor software planning and deployments. The presentation of the resulting effects of poor deployments presents the need for deployment approaches to mitigate these effects. The article goes on to present 12 synchronous steps in solving the software deployment issues. Weissmann finishes the article with a discussion on the Darwin Ports.

16.) - Fitzgerald, B. 1998. An empirical investigation into the adoption of systems development methodologies. *Inf. Manage.* 34, 6 (Dec. 1998), 317-328. DOI= http://dx.doi.org/10.1016/S0378-7206(98)00072-X

Brian Fitzgerald's article centers on the perspectives and arguments that support the use of a methodology in the software development process. Fitzgerald's article is unique in the sense that he presents the attacking arguments for not using methodologies as lead ins for constructing his case for supporting the use of methodologies. The article also possesses interviews with different types of developers and practitioners from various organizations.

17.) - Guimarães, L. R. and Souza Vilela, P. R. 2005. Comparing software development models using CDM. In *Proceedings of the 6th Conference on information Technology Education* (Newark, NJ, USA, October 20 - 22, 2005). SIGITE '05. ACM, New York, NY, 339-347. DOI= http://doi.acm.org/10.1145/1095714.1095793
Guimarães and Souza Vilela propose a systematic way for comparing software development models. This is done though formal techniques they propose in their research paper. The study shows the validity of their techniques by presenting a case study of involving the comparison of two development models. The advantages and disadvantages of these two models are provided.

18.) - Nerur, S., Mahapatra, R., and Mangalaraj, G. 2005. Challenges of migrating to agile methodologies. *Commun. ACM* 48, 5 (May. 2005), 72-78. DOI= http://doi.acm.org/10.1145/1060710.1060712
The 'Challenges of Migrating to Agile Methodologies' details the benefits and challenges of implementing agile in an organization that has practiced traditional methodologies such as waterfall. The article also approaches the implementation of agile from the aspect of object oriented development. It contains very applicable points concerning current IT environments and situations.

19.) - Dolstra, E., Bravenboer, M., and Visser, E. 2005. Service configuration management. In *Proceedings of the 12th international Workshop on Software Configuration Management* (Lisbon, Portugal, September 05 - 06, 2005). SCM '05. ACM, New York, NY, 83-98. DOI= http://doi.acm.org/10.1145/1109128.1109135

Dolstra, Bravenboer and Visser's article provides supporting arguments for the practice of automated build management, software deployment and service deployment as a single notable process. This article also shows how the build and deployment of software components are a time consuming and error prone process. Topics such as co-existent versioned deployments, single component upgrades and rollbacks are also touched upon by implementing a reliable repeatable automated deployment process.

20.) Belguidoum, M. and Dagnat, F. 2006. Analysis of deployment dependencies in software components. In *Proceedings of the 2006 ACM Symposium on Applied Computing* (Dijon, France, April 23 - 27, 2006). SAC '06. ACM, New York, NY, 735-736. DOI= http://doi.acm.org/10.1145/1141277.1141445

Belguidoum's and Dagnat's article takes a very theoretical approach of the automated deployment of software in a proposed framework or model. The article does not so much detail the specific characters, factors or entities involved or needed in an automated deployment technology but the high-level need and sub-goals for a successful deployment model through automation. By discussing these sub-goals a reader or technologist can realize the complexities behind software deployments. These sub-goals, ultimately realized gains are what will be helpful in supporting the technology and proposal of this project capstone paper.

21.) Ray, Steven (2002, November 15). The Future of Software Integration: Self-integrating Systems. *National Institute of Standards & Technology*, Retrieved October 10, 2008, from http://www.nitrd.gov/subcommittee/sdp/vanderbilt/position_papers/steven_ray_the_future_of_software.pdf

Ray Steven's article on what lies ahead for software integration makes the case for the need or rather unavoidable realization of the benefits of continuous software integration. His article is short but concise with description of the current solutions and the problem that results as the environment for software development changes and evolves. The article is theoretical and opinionated but very applicable considering it was written in 2002. No technological solution is presented for based on the documented situation and potential benefits of continuous software integration.

22.) Fowler, Martin (Fowler, 2008).

http://www.martinfowler.com/articles/continuousIntegration.html Martin Fowler is a renowned author, speaker and architecture on object-oriented analysis, design and development and software development methodologies. His website contains insightful documentation from experiences and findings on his involvement in agile development methodologies and software deployments.

23.) GenerExe. (2002). *A Short History of Software Development* (1.0 ed.) [Brochure]. Amsterdam-Netherlands: GenerExe.

GenerExe is an organization of independent developers who design and develop information solutions which are more reliable, affordable and understandable. Their article provides a decent summary of the beginnings of hardware and software. It outlines and provides interpretative meaning to significant stages of software integration into hardware components.

24.) Wirth, Niklaus (1999). A Brief History of Software Engineering. Retrieved January 19, 2009, from Department of Informatik at ETH Zurich Web site: http://www.inf.ethz.ch/personal/wirth/Articles/Miscellaneous/IEEE-Annals.pdf

Niklaus Wirth was an assistant professor of computer science at Stanford University and a professor of informatics at ETH Zurich. His article on the history of software engineering is rather detailed than brief. He provides exceptional insight into the evolution of hardware and software. This insight aids in illustrating how software engineering and programming along with implementing industries' delivery time pressure has created a back-seat situation for development methodologies. He shows

that methodologies are one of the keys to producing quality software for the future and not increasing hardware and software components.

25.) http://www.agiledata.org/essays/differentStrategies.html - Scott W. Ambler

Ambler article discusses the various approaches to implementing agile software development methodologies.

26.) http://www.scottcreynolds.com/archive/2007/12/13/why-use-continuous-integration---beginners-overview.aspx - Why Use Continuous Integration - Beginner's Overview by Scott C. Reynolds

Scott's article provides valuable insight into explaining continuous integration to novice software developers or persons with little knowledge of configurations management.

27.) Highsmith, Jim (2001). Principles behind the Agile Manifesto. Retrieved February 1, 2009, from Manifesto for Agile Software Development Web site: http://agilemanifesto.org/principles.html

28.) http://www.versionone.com/Resources/AgileHallmarks.asp - Agile Hallmarks

This site by the developers of the Version One project management tool was used to provide examples of project metric tracking.

29.) http://scalingsoftwareagility.files.wordpress.com/2007/09/mastering-the-iteration-an-agile-white-paper.pdf - Iterations fleshed out

30.) Hopkins, Will [G.] (2008, July). Quantitative Research Design. Retrieved March 5, 2009, from SPORTSCIENCE Web site: http://www.sportsci.org/jour/0001/wghdesign.html

Hopkins article provides a context for the types of quantitative research. This context aids in providing the lessons learned and analysis section of the thesis paper with structure. Structure in the sense of analyzing the metrics collected through the use of the derived component applied to the four test scenarios.