The Deployment Pipeline

Dan Mikita
danmikita@gmail.com

Gerald DeHondt
dehondtg@gvsu.edu

School of Computing and Information Systems,
Grand Valley State University
Allendale, MI 49401 USA

George S. Nezlek
gnezlek@gmail.com

Abstract

The ultimate goal of a software development process is the deployment of a quality piece of software. A deployment pipeline includes many aspects of software development that are rarely focused on by a development team. The proper development process should include as little manual testing and configuration as possible, while still confirming all of the functional and non-functional requirements that the system must satisfy. The business users, who are the most knowledgeable about the functional and non-functional requirements, also need to be directly involved in writing those tests. All dependencies should be known and managed consistently, with the objective of having a releasable application after every commit. This provides for a stable application, reliable development and production environments, consistent releases, and a product that meets a larger number of business goals.

Keywords: deployment pipeline, behavior-driven development, continuous integration, configuration management

1. INTRODUCTION

An important, yet often overlooked, portion of software development is the deployment pipeline. A properly defined deployment pipeline can help improve the quality, speed, and robustness of a project. Improper developmental quality has a direct and measurably negative effect on a system's life cycle costs; it takes longer to understand and maintain system code, and architecture drift is harder to discover. It is more laborious to test such systems, and chances are higher that modifications will introduce bugs that are more costly to address (Buschmann, 2011).

This paper will discuss the many aspects of the deployment pipeline, and suggest how to achieve the many benefits it offers. No project is too small to use the methods suggested here, nor are all the ideals mentioned necessary to achieve a better overall state for a project. It is important to note that any one of the dimensions of deployment pipelines may be used separately from the others to add value to a project, but the greatest overall benefit will result from incorporating all of them.

The term ***deployment pipeline*** refers to how software gets from the development phase to

the release phase (Humble & Farley, 2011). Although this may seem relatively straight-forward, there are many things to consider when delivering an application consistently and confidently each time it is released. Having a solid deployment pipeline also enhances how developers, testers, and build-and-operations personnel work together effectively. The range of topics may be vast, including configuration management and version control, to testing methodologies and requirements gathering; each piece is a necessary component of a larger picture. The larger and more complicated the system becomes, and the more complicated the composition of the development teams, the more obstacles are placed in the way to the release of a software system (Kraut and Street, 1995). Since most software systems are developed by teams, effective coordination and communication are crucial to the success of software projects.

2. CONFIGURATION MANAGEMENT

Before approaching the topic of testing itself, there is an extremely important aspect of every project that needs to be in place first: configuration management. Configuration management refers to the process by which all artifacts relevant to a project, and the relationships between them, are stored, retrieved, uniquely identified, and modified (Humble & Farley, 2011). In effect, a configuration management strategy will determine how to manage changes within a project. Humble & Farley (2011) offer some insightful questions to ask to determine if the current configuration management strategy is in good condition:

- Can I exactly reproduce any of my environments, including the software versions and their configurations?
- Can I easily make an incremental change to any of these individual items and deploy the change to any, and all, of my environments?
- Can I easily see each change that occurred to a particular environment and trace it back to see exactly what the change was, who made it, and when they made it?
- Can I satisfy all of the compliance regulations that I am subject to?
- Is it easy for every member of the team to get the information they need, and to make the changes they need to make?

If the answer is "yes" to all of these questions, then the project is probably in a good state. But if the answer is "no" to any one of them, it will provide a good idea of where to start to fix the current process. There are three topics that will be discussed in the next section that are all key components to a successful configuration management strategy: version control, environment set-up, and dependency management.

Version Control

A version control system (VCS), also known as source control, is a staple of every project. VCS is a way to keep a history of any file placed within it. VCS provides the ability to track changes and revert them if necessary (Ruparelia, 2010). It is also a means for multiple users to make changes to the same files at the same time (Louridas, 2006). Some examples of the most common VCS's consist of SVN, Git, and Mercurial. Although the topic of this paper is not version control, there are some very important concepts to keep in mind with regard to VCS use.

One of these concepts is that a version control system is not just for source code. Every single artifact related to the creation of software should be under version control. Developers should use it for source code, tests, database scripts, build and deployment scripts, documentation, libraries and configuration files for any application, the compiler and collection of tools, and so on – so that a new team member can start working from scratch (Humble & Farley, 2011). To put it simply, it is extremely important to store all the information needed to re-create the testing and production environments within a version control system. This allows the team to roll back to the last previously known good state should any system modifications introduce, or re-introduce, errors into the system. Proper version control also ensures that the latest system build is readily available for users to validate adherence to requirements. This level of configuration management ensures that, provided an intact repository, the team will always be able to retrieve a working version of the software (Humble & Farley, 2011).

Every version control system offers the ability to add a small message to every commit that is made. This gives those who look at a project's history an idea of what the change in the commit was for (Spinellis, 2005). Because this

message is required to make a commit, some individuals get in the bad habit of writing something trivial, along the lines of "fixed a bug". It is imperative to include useful information to inform those who will continue to support the application. Typically, an appropriate method is to include a short concise line at the beginning, followed by a more detailed description below, since the first line is what is typically displayed in most views. Appropriate information tagging makes information retrieval and environment recreation easier (Treude and Storey, 2012). Information retrieval is also enhanced by community tagging, allowing a broader understanding of the material attached to the artifacts (Robu, Halpin, and Shepherd, 2009). This documentation feature is beneficial to the developers, testers, and subject matter experts who will continue to support the system after deployment.

Branching is another topic that can cause issues within a project. A branch, in terms of version control, provides the ability to generate an exact copy of the current code base with the purpose of working on a new piece of functionality. This allows the new code to be written without affecting the main line of code. In most cases this can cause more trouble than it is worth (unless a distributed version control system is being used). The reason for this is the difficulty that can arise when trying to merge those new changes back into the main line of code. Frequently, many of the same files that were changed on the branch were also changed on the main line of code, which can lead to a large number of conflicts and trouble. Synchronizing all the changes made is imperative, and managing the different branches may prove unwieldy if this is not properly managed.

Although branching is not recommended, there is one situation where branching is extremely useful: releases. By branching whenever a release is cut, the ability to continue developing the mainline code while still having a stable representation of the application currently in production is undeniably useful.

An expansion of all issues relevant to version control is beyond the scope of this manuscript, but keep these topics in mind as the discussion of the deployment pipeline continues. As previously mentioned, version control can be used for more than just source code, and an example of that is a program's environment.

Environment Set-up

Every application ultimately depends upon the hardware, operating system, software, and all the other aspects of a computer system that allows it to run. Consequently, it makes sense to manage the environment an application will run on as well.
If an application requires a certain version of a driver, or possibly requires a port to be available on the system, the environment will need to be set-up in that way to work properly. The worst possible approach to managing this is on a piecemeal, system-to-system basis. One of the primary challenges with this method is that if a problem occurs with the current configuration, there is no record of the last known good state and hence no method to roll back.

The best way to approach this situation is by automating the entire process. Automation precludes the possibility of having only one individual who knows how to set-up a new environment, as well as the ability to easily revert back to a known good configuration. Also, it allows creation of test environments that mimic a production machine, which is required for manual user acceptance testing, to be discussed later. All configuration specifications can be checked into version control and pushed out to various environments using tools such as Puppet or CfEngine. These tools allow the user to define things such as what access level individual users have and what software should be installed, thus letting users store configurations in version control and initiate the rollout through the same version control system. This environment set-up is simply another piece of the puzzle that leads to the management of a project's dependencies.

Dependency Management

Most applications have many dependencies, and whether they are third party libraries or internal components, it is necessary to have a way to manage them all. The main goal of dependency management is to enforce consistent, repeatable builds. If a tester checks out a project from a version control system and runs the automated build, the exact same libraries should be used (Humble & Farley, 2011). There are two main approaches to managing a project's dependencies. The first is checking all of the project's dependencies into a version control system. The second is to use a tool such as Maven or Ivy that will transitively resolve

dependencies with other projects and ensure that there are no inconsistencies in the project dependency graph. Furthermore, these tools cache the libraries a project needs on the local machine so that consecutive builds are just as fast as if the libraries were checked into version control.

Another important aspect of dependency management is the use of a personal artifact repository. An example of this is Sonatype's Nexus. These are extremely useful for internal components that multiple projects within an organization may depend upon. The practice of using an internal artifact repository also makes it much easier to audit these libraries and prevent violations of legal constraints (Humble & Farley, 2011).

Having an internal artifact repository leads to the ability of dividing a project's code base into components. There are several reasons why components make the software development process more efficient. One is that they allow a project to be divided into more expressive chunks, as well as clear separate responsibilities (Belguidoum & Dagnat, 2007). This can lead to more freedom when optimizing a build and deployment process.

Implementing these processes allows continuous integration of software to remove the manual tinkering and configuration required for each release.

3. CONTINUOUS INTEGRATION

The topic of continuous integration is at the heart of a good testing strategy. Continuous integration and automated testing mean that it is difficult for developers to deliver poor, low-quality code (Conboy, Coyle, Wang, and Pikkarainen, 2011).

Lacking this, project tests serve much less of a purpose than what they are able to. It is extremely common during the development of a piece of software for the developer to work on just a small portion of the code, and only run and/or unit test that portion. In fact, it can be nearly impossible to run the entire application in a production-like environment for the individual developer. Thus, it is possible that a change the developer makes in one part of the code may leave the overall application in a non-working state. When a software project is composed of dozens of components with complicated dependencies among each of them, a change to one component often has a drastic effect on the others. This is where continuous integration comes in (Kim, 2008). Nerur & Balijepally (2007) state that continuous code integration can improve software design and the code base. Schrodl & Wind (2011) investigated a project demonstrating the applicability of this approach. This iterative approach also allows it to be tolerant of changes in requirements (Beck & Andres, 2004). Agile methods utilizing continuous code integration focus on providing high customer satisfaction through three principles: quick delivery of quality software; active participation of concerned stakeholders; and creating and leveraging change (Highsmith, 2002).

As an example, agile methodologies apply iterations to all phases of a project, from system requirements specification all the way down to system testing (Zhang and Patel, 2011). This is achieved by tight collaboration of different teams through a feature team. Scrum, specifically, is a lightweight, simple-to-understand, yet difficult-to-master Agile framework, through which people can address complex, adaptive problems (Schwaber & Sutherland, 2012). Our proposed model implements a framework in a similar, iterative fashion.

There are three things that are necessary before continuous integration can be achieved:

- Version Control
- An Automated Build
- Agreement of the Team

Version control is crucial for the central location and change tracking that it provides. An automated build is necessary for the ability to continuously monitor each change for a negative impact. The third item is worth noting, because without the full commitment of a team, any continuous integration (CI) plan will likely be unsuccessful.

CI Basics

Utilizing a CI tool such as CruiseControl or Hudson will help get a system set-up in a hurry. The steps for installing and setting up these tools are extremely straight forward and will only take a few moments. After the tool has been configured with the version control repository, compile scripts, and run the

automated commit tests for the application; the CI system will have the ability to determine if the last set of changes broke the software (Haines, 2008). According to Humble and Farley (2011), there are seven basic steps to follow:

1. Check if the CI tool is currently building your application. If so, let it finish.
2. Once it finishes, update your local code in your development environment from your version control system.
3. Make any changes you need to make, then run your tests and build the script locally.
4. If your local tests and build pass, check your changes into your version control system.
5. Wait for your CI tool to notice the changes and automatically start running.
6. If it fails, fix the problem immediately and return to step 3. Otherwise, continue on to step 7.
7. Celebrate!

As can be seen, it is important for everyone on the team to follow these rules, otherwise certain members of the development team will end up having to fix others' bad code.

Essential Practices

For continuous integration to work properly, there are a few essential practices that must be followed. The first and most important practice is checking in regularly to a version control system. Small commits throughout the day will ensure that the team always has a releasable version of the application at the end of the day. These small commits also mean there is a smaller chance of breaking the build, as well as less chance of conflicts with other developers.

The second essential practice is having a comprehensive automated set of tests. The point of using continuous integration is being able to consistently run a suite of automated tests every time a commit is made by any developer. This helps ensure current functionality and eliminates "negative work", where more errors are introduced than requirements fulfilled in the latest build. While keeping this concept in mind, it can be seen why it is imperative to have a comprehensive set of automated tests.

4. TESTING

The proper testing process should include as little manual testing and configuration as possible. It should also confirm all functional and non-functional requirements, while allowing the business users (who know those functional and non-functional requirements better than the developer) to be personally involved in writing those tests. Even in situations where the majority of tests are automated, they are often poorly maintained, out-of-date, and require significant manual testing to make up for their deficiencies (Humble & Farley, 2011). It is important when updating an application to ensure that the automated tests remain consistent with application functionality. Zhang and Patel (2011) recommend that the team develop and verify individual test cases first, then add them to the list for automated batch mode execution, execute all the test cases in batch mode overnight, and finally analyze test results to find the root cause of the failed test cases. In this scenario, it helps to ensure that the test cases remain relevant to the current application iteration. Riungu-Kalliosaari, Taipale, and Smolander (2012) have even recommended the use of cloud-based testing services to enhance testing agility and speed the development and deployment pipeline. These methods could result in faster delivery of products to address business needs.

It is also important when writing automated tests to keep in mind that just having a test in place means very little if it does not prove a business function. Each test needs to be directly traceable to validate system requirements. Surveys have shown the lack of automation of software testing tasks in most software organizations (Polo, Tendero & Piattini, 2006). To build a quality application, it is imperative to have automated tests at multiple levels, including: unit tests, component tests, and acceptance tests. Having these tests at multiple levels allows them to be run as part of the deployment pipeline, which should take place every time a change is made to the application or configuration. This is achieved by the Continuous Integration with a version control system that we discussed previously.

Test-Driven Development

Test-driven development is a software development process that is designed to consist of short development cycles. The way the process works is that the developer will first write an automated test case that defines the functional or non-functional requirement defined by the business. These tests are written before

any code relating to the application is developed. Next, the developer will write code for the application that makes these tests pass. Once the code has been written, and the tests pass, the developer will refactor and polish his initial codebase, while ensuring required functionality is maintained.

Using this strategy, a development team can easily achieve all the functional requirements defined by the business. This development practice has led to many other forms of test-driven development, one of which shows much promise, Behavior Driven Development. Before we can understand the usefulness of behavior-driven development, we must first talk about requirements gathering.

Requirements Gathering

When developing an application for a business partner, requirements gathering may be more challenging. Most developers go through a process of requirements gathering before starting any project, but the requirements gathering used for a business application can take on a whole new appearance. The reason for this is that the complexity of business processes can be overwhelming and impossible for a developer to understand on a deep enough level to write an application that will mimic those processes (Holub, 2005). It is in these situations that a subject matter expert, with a thorough knowledge of the business, is imperative to the success of a project.

In an ideal world, the business will assign an individual to the project who understands the concepts of the business processes on a deep enough level to explain them to the developer. In this way, the business user and the developer can work together to write the automated test suite that will allow the developer to achieve the functionality the business requires. Unfortunately, customers have become increasingly unable to definitively state their needs up front while, at the same time, expecting more from their software (Lucia & Qusef, 2010). This leads us to the discussion of behavior-driven development.

Behavior-Driven Development

As we have stated, in an ideal situation there will always be a business user available to the development team to help with any functional requirements the business may have. To make the collaboration between business and developer even easier, the concept of behavior-driven development (BDD) has been created.

Initially created by Dan North in 2003, BDD is now starting to gain widespread acceptance in the field. BDD focuses on allowing business users to actually write the tests themselves (North, 2006). Since these business users will be the subject matter experts, they will have the best knowledge of the required system functionality. Specifically, Conboy et al. (2011) note potential indifference and disengagement on the part of business users when excluded from system development activities. They may believe that the development team knows little about the business side and will be unable to deliver value. In this instance, it is incumbent upon them to write the tests that will determine successful system behavior. In the end, it will be the business users who ultimately accept the completed system, based on adherence to these pre-defined requirements. As the business users are the Subject Matter Experts who will be working with the software, they best understand the requirements necessary for successful software development.

This is where BDD comes in. With tools such as JBehave and RSpec, writing automated tests no longer requires any knowledge of programming. This type of tool allows the use of "statements" that are attached to code. Take for example:

Scenario 1: Refunded items should be returned to stock
- Given a customer previously bought a black sweater from me
- and I currently have three black sweaters left in stock
- when he returns the sweater for a refund
- then I should have four black sweaters in stock

By using keywords such as Given, When, and Then, the tool is able to determine how to set up the test (North, 2006). The key words are then matched to annotations within java code that is written by the developer. In this way, specific tests written by the business can be used to determine the completeness of an acceptance test suite. As the functionality of software expands, and the underlying code gradually becomes more complex, tools like these will need to become more prevalent to continue to deliver quality software (Crowther & Clarke, 2005).

Types of Tests

Unit Tests

Unit tests and component tests are both exclusively written and maintained by the developer to verify that a piece of code works the way it should. Unit tests validate small pieces of code, usually one method or even a small part of a method. It is important to remember that unit tests do not involve making calls to a database, using the file system, talking to external systems, or any other external component. Thus, in most situations, unit tests rely heavily on simulated, mocked data. The use of mock data may actually serve a better purpose than the use of production data. Production data may exercise the most common situations and leave unanticipated holes in the tests and code coverage incomplete. Using mock data can help ensure a greater level of code coverage and that all paths of the application are tested.

Component tests are similar to unit tests, but they test a much larger portion of the code. They are typically slower, and involve the external resources that unit tests do not, such as: database calls, use of the file system, or talking to external systems (Humble & Farley, 2011).

Acceptance Tests

Acceptance testing ensures that the acceptance criteria for a story are met. Acceptance tests should be written, and ideally automated, before development starts on a story (Ambler, 2007). Acceptance tests, like acceptance criteria, can test all kinds of attributes of the system being built, including functionality, capacity, usability, security, modifiability, availability, and so on (Humble & Farley, 2011). Conboy et al. (2011) mention that the developers in their study had technical skills in abundance, but no business acumen whatsoever. It was very tough for the users to get the business angle across to the developers. Because of this, it is critical for the acceptance tests to be written by the business users. Acceptance tests come in two categories, functional tests and non-functional tests.

Functional Tests

Functional tests are one of the most important set of tests that can be written for an application. When run, these tests will answer the question of, "Am I done?" and "Did I deliver what the customer wanted?". These tests also provide the opportunity to determine whether a change that was made to one part of the code broke anything in other areas of the software.

Non-Functional Tests

Non-functional tests are largely component tests that are directed at the qualities or attributes of the software. To be more specific, qualities such as capacity, availability, security, etc. are what non-functional tests are used for (Ambler, 2007). In many projects these tests are not used at all, although this is a large mistake. If these non-functional requirements are not mandated by the business, it is possible to see software with many flaws that may not meet user requirements. For example, if performance is not tested, a database call that returns hundreds of thousands of results can take many seconds. By having a test in place to monitor performance, it will allow a developer to create a fast and responsive system that does not lag on database calls or calls to other external systems. Another example is security. Testing for security is also extremely important, and is oftentimes overlooked by business users. It is important to consider the non-functional requirements when designing a suite of tests.

5. THE DEPLOYMENT PIPELINE

Everything that has been discussed thus far can be summarized into one overarching category: The Deployment Pipeline. Although the discussion up to this point has been about each part individually, it is important to take a step back and look at the whole picture to see how it all fits together as illustrated in Figure 1. Each piece is extremely useful by itself, but when put together, a project can take an enormous step forward in productivity and quality.

To describe exactly how all of this fits together, it is necessary to start from the beginning and walk through each step on the journey through the deployment pipeline. When a developer makes a change to the source code, the automated unit test suite will be run on the local machine before being committed. If everything passes, the changes can then be committed to the version control system; if not, the developer must go back and fix the unsuccessful tests. After the changes have been committed to the version control system, the continuous integration tool (CI) will notice this new commit

and begin running the acceptance test suite which includes all the unit tests, functional tests, and non-functional tests for the system. If there is a test failure, the CI tool will inform the team and the individual who made the change will need to fix what has broken the build. If all the tests pass successfully, the user acceptance test phase can begin. Before that can happen, it is imperative to deploy the application to an environment that replicates production. The authors recommend the use of multiple replicated environments through which software will pass. These would include a separate development environment and testing environment, prior to production deployment. The development environment would serve as a "sandbox" where developers would make changes to their code. Once promoted to the test environment, no further changes could be made, and the sole purpose of this environment would be to verify software changes prior to promotion to the critical production environment.

It is important to note here, that no matter how many automated acceptance tests exist or how thoroughly they test the software, nothing can replace a final run-through with manual user acceptance testing. Finally, if everything passes successfully, the application can be deployed to production.

6. CONCLUSION

There are many pieces that fit into the deployment pipeline puzzle, with each serving a critical function along the way. Having one central location for all code, libraries, and environmental configurations to be stored is imperative to any project, big or small. Not only does it provide a consistent location to find the code, it also delivers the ability to literally go back in time if a costly mistake was made. To reliably and confidently release an application, it is necessary to know that each version of a project is using the same versions of external libraries by using a tool such as Maven to simplify dependency management.

Continuous integration helps to always have a releasable version of code at the ready. By consistently running a set of acceptance tests, it is easy to know that the software is always satisfying the functional and non-functional requirements detailed by the business. Balijepally, Mahapatra, & Nerur (2009) note that with the increasing acceptance and popularity of agile methodologies, there is a need to investigate the efficacy of core practices which may have large cost and productivity implications for the software development community. They continue that given the increasingly social nature of software development approaches, it is critical to have a good grasp of the factors that affect group performance in a software development context, thus impacting the acceptance of a proposed method.

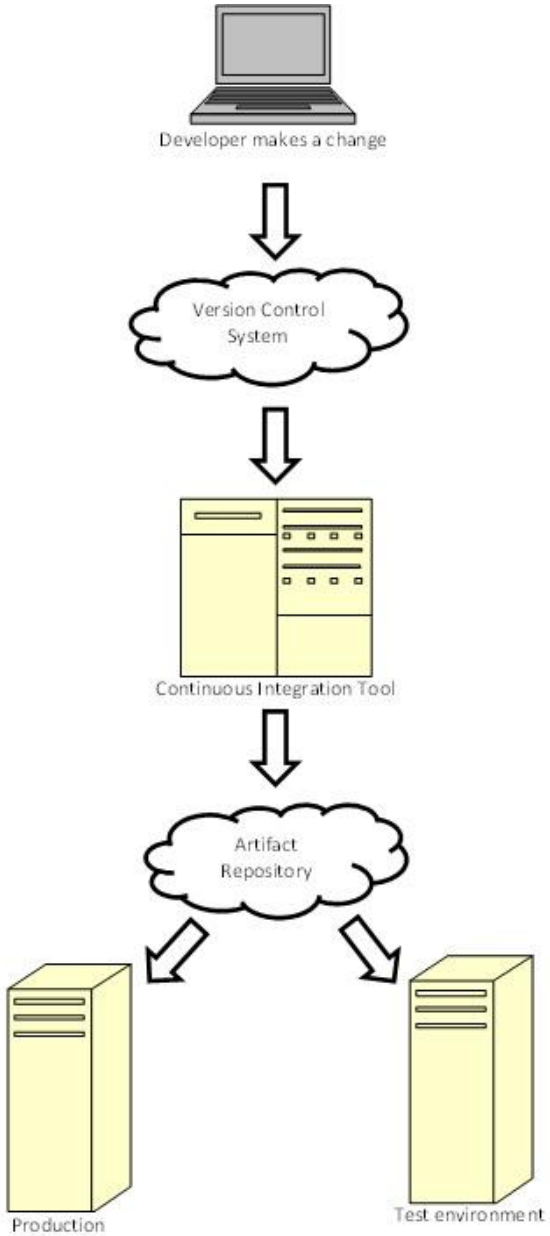

Figure 1

Finally, engaging the business users is to the developer's advantage with tools such as JBehave, which can greatly increase the number and quality of the tests within the automated acceptance testing suite. By using the types of tools mentioned above, producing a consistent, highly reliable release can be as easy as pressing a button.

7. REFERENCES

Ambler, S. W. (2007). Agile Testing Strategies. *Dr. Dobb's Journal*, 32(1).

Balijepally, V., Mahapatra, R., Nerur, S., & Price, K. (2009) Are Two Heads Better Than One for Software Development? The Productivity Paradox of Pair Programming. *MIS Quarterly*, 33(1), 91-118.

Beck, K. & Andres, C. *Extreme Programming Explained: Embrace Change, Second Edition*. Boston, Massachusetts: Addison-Wesley, 2004.

Belguidoum, M., & Dagnat, F. (2007). Dependency Management in Software Component Deployment [Electronic version]. *Electronic Notes in Theoretical Computer Science*, *182*, 17-32.

Buschmann, F. (2011). Gardening Your Architecture, Part 1: Refactoring, *IEEE Software*, 28(4), 92 - 94.

Conboy, K., Coyle, S., Wang, X., and Pikkarainen, M. (2011). People over Process: Key Challenges in Agile Development, *IEEE Software*, 28(4), 48 - 57.

Crowther, D. C., & Clarke, P. J. (2005). Examining Software Testing Tools. *Dr. Dobb's Journal*, *30*(6), 26-33.

Haines, S. (2008). Continuous Integration and Performance Testing. *Dr. Dobb's Journal, 33*(3), 36-38.

Highsmith, J. (2002). *Agile Software Development Ecosystems*. Boston, Massachusetts: Addison-Wesley.

Holub, A. (2005). Requirements Gathering. *Software Development Times*, 35.

Humble, J., & Farley, D. (2011). *Continuous Delivery*. Boston: Pearson Education, Inc.

Kim, S. (2008). Automated Continuous Integration of Component-Based Software: an Industrial Experience. *Proceeding ASE '08 Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. doi:10.1109/ASE.2008.64

Kraut, R. and Streeter, L. (1995). Coordination in Software Development, *Communications of the ACM*, 38(3), 69-81.

Louridas, P. (2006). Version Control. *IEEE Software*, *23*(1), 104-107.

Lucia, A. D., & Qusef, A. (2010). Requirements Engineering in Agile Software Development. *Journal Of Emerging Technologies In Web Intelligence*, *2*(3), 212-220.

Nerur, S., & Balijepally, V. (2007). Theoretical Reflections on Agile Development Methodologies. *Communications of the ACM,* 50(3), 79 – 83.

North, D. (2006). Introducing BDD. *Better Software*. Retrieved from http://dannorth.net/introducing-bdd/

Polo, M., Tendero, S., & Piattini, M. (2006). Integrating Techniques and Tools for Testing Automation. *Software Testing, Verification And Reliability*, *17*(3), 3 - 39.

Riungu-Kalliosaari, L., Taipale, O., and Smolander, K. (2012). Testing in the Cloud: Exploring the Practice, *IEEE Software*, 29(2), 46 - 51.

Robu, V., Halpin, H. and Shepherd, H. (2009). Emergence of Consensus and Shared Vocabularies in Collaborative Tagging Systems, *ACM Transactions on the Web*, 3(4), 1-34.

Ruparelia, N. B. (2010). The History of Version Control. *Software Engineering Notes*, *35*(1), 5-9.

Schrödl, H. & Wind, S. (2011) Adoption of SCRUM for Software Development Projects:

An Exploratory Case Study from the ICT Industry. *AMCIS 2011 Proceedings - All Submissions*. Paper 256.

Schwaber, K. & Sutherland, J. (2012) The Scrum Guide. Available online at: www.scrum.org.

Spinellis, D. (2005). Version Control Systems. *IEEE Software*, *22*(5), 108-109.

Treude, C., and Storey, M. (2012) Work Item Tagging: Communicating Concerns in Collaborative Software Development, *IEEE Transactions on Software Engineering*, 38(1), 19 - 34.

Zhang, Y. and Patel, S. (2011). Agile Model-Driven Development in Practice, *IEEE Software*, 28(2), 84 - 91.