World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:8, No:9, 2014

Release Management with Continuous Delivery: A Case Study

A. Maruf Aytekin

***Abstract*—**We present our approach on using continuous delivery pattern for release management. One of the key practices of agile and lean teams is the continuous delivery of new features to stakeholders. The main benefits of this approach lie in the ability to release new applications rapidly which has real strategic impact on the competitive advantage of an organization. Organizations that successfully implement Continuous Delivery have the ability to evolve rapidly to support innovation, provide stable and reliable software in more efficient ways, decrease the amount of resources need for maintenance, and lower the software delivery time and costs. One of the objectives of this paper is to elaborate a case study where IT division of Central Securities Depository Institution (MKK) of Turkey apply Continuous Delivery pattern to improve release management process.

***Keywords*—**Automation, continuous delivery, deployment, release management.

I. Introduction

BUSINESSES need to respond ever faster to new opportunities. They also need to evolve rapidly in order to maintain a competitive advantage. For many organizations, their ability to do so is reliant on the software that supports their business. In fact, most CEOs were looking to use technology to gain both efficiency and differentiation simultaneously [1].

In the fall of 2011, Forrester Consulting conducted in-depth surveys with 325 business and IT professionals. Forrester found that most of these companies are not able to deliver new custom software solutions as fast as business leaders need them. According to the survey a major reason for this is that they have a low level of maturity when it comes to software delivery processes. As a result, most of the companies are not able to use their software development capacity to drive their business, and they are not able to release new applications to support their businesses as fast as they would like [2].

IT organizations face two different and conflicting pressures. They need to respond rapidly to a changing business environment and deliver high quality software and services fast. This is the capability of an IT organization and is called "performance" which is the ability to deliver more with less.

On the other hand, IT organizations need to meet the Service Level Agreements (SLA) and regulatory requirements, support complex and interdependent systems, protect the stability of the live environment, and manage risks to the business sufficiently. Thus the other key capability of an IT organization is "conformance" which is the ability to adequately manage risks to the business.

The apparent conflict between these two requirements — performance and conformance— is most strongly felt during releases. Releases are often painful and risky affairs, at worst, they can result in service interruptions and end in a roll back, at best, people lose their evenings, mornings, or weekends [3].

In this paper, we discuss the release management model with continuous delivery to enable IT organizations to achieve both performance and conformance by mitigating the risks so that releases become a routine, push-button event.

This paper has the following structure: Section II will briefly discuss different background concepts and definitions. In Section III we introduce process of release management from the development to production followed by a concrete case study in Section IV which also describes how we applied this approach in our practical software development life cycle. An evaluation of this approach and future work are discussed in Section V.

II. Background

For many organizations, release management is not critical only for development and deployment practices; but also for doing business. In most organizations, the delivery of software is a time consuming, stressful and costly process. As soon as an application goes live, issues pop up, forcing the IT organization into another costly release cycle. A software release is typically a risky, unreliable procedure that costs businesses both time and expense. The common causes of this problem is infrequent releases, manual delivery process, manual data migrations and database updates, manual configuration changes, poor collaboration between teams and team members, and lack of integration tests.

Frequent integration has proven so successful over time that it is now a mainstream development practice known as Continuous Integration (CI). However, since CI is focused on development, it can only benefit a fraction of the end-to-end release process, which remains a high-risk, labor-intensive affair in a majority of IT organizations [4]. Continuous Delivery (CD) is a method that automates the delivery process of the release, lowers the risks, and minimizes manual affairs and requires the creation of an automated deployment pipeline to release software rapidly and reliably into production.

In the following subsections, we will give brief definition of Release Management, CI, and CD processes.

A. Maruf Aytekin is with Release Management Team atCentral Securities Depository Institution (MKK), Istanbul, Turkey (phone: +90-212-334-5700; fax: +90-212-334-5757; e-mail: maruf.aytekin@ mkk.com.tr).

A. Release Management

Information Technology Infrastructure Library (ITIL) defines release management as the process responsible for planning, scheduling and controlling the movement of releases to test and live environments. The primary objective of release management is to ensure that the integrity of the live environment is protected and that the correct components are released [5].

B. Continuous Integration

CI is a software development practice where members of a team integrate their work frequently. Usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build 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 [4].

C. Continuous Delivery

CD is a software deployment the continual delivery of code to an environment once the code is ready to ship. CD requires creating a "Continuous Delivery Pipeline" which sometimes called "Deployment Pipeline". With deployment pipeline delivering the code to various environments, feeding the business logic tests, and providing feedback to the developers become a push button event.

III. Software Release Management

An effective release management process lowers the risk and creates a reliable procedure to protect the integrity of production and deliver high quality software and services fast and also ensures businesses get valuable feedback quickly. The key to achieving these goals is to improve the process of software release by using continuous integration and continuous delivery [6], [7].

A. Continuous Integration Revisited

The traditional workflow of CI process can be summarized as follows. Project source code and all resources are being managed in a Source Code Management (SCM) tool to be able to trace the changes made to them over time. A member of development team checks out the copy of the software. He makes some changes to this local copy, makes a local build and runs local development tests. After completing this local verification and test process, he commits the code back to the SCM tool. A CI server is used to integrate all changes from members of development team. CI server makes a build to integrate changes from development team to see whether the software builds successfully or not. The build process might contain automated integration tests and code analysis and validation checks to ensure that committed code conforms to common code conventions. If there are build errors or integration test failures, build process is stopped and a notification being sent to development team. It is the committer's responsibility to fix the erroneous code as soon as possible.

Martin Fowler defines the key practices that make up effective CI as follows [7]. An effective CI system must use a single source code repository and a release opening strategy must be established such as Branch for Testing Model (BTM) [8]. Everyone in the development team should commit to the mainline every day and each commit should build the mainline on the CI server. Builds and tests must be automated and be fast. Tests should be executed in a clone of the production environment. Latest build artifacts and executable must be available to everyone. Development team members and counter parties must be able to see what's happening. It is essential that all manual processes need to be avoided in any of CI steps.

B. Continuous Delivery Pipeline

Deployment pipeline is an automated manifestation of your process for getting software from version control system into hands of your users [9]. Deployment pipelines are the central part of Continuous Delivery [10]. A deployment pipeline creates visibility, provides control and complete traceability over the delivery process of the software, and enables collaboration by providing instant feedback on the production readiness of the software. It also provides control over the delivery process by enabling developers, testers, and operations teams to perform push-button releases of any chosen version of an application into the managed environments. Thus the delivery process can be audited from check-in to release.

The first stage of a deployment pipeline (see Fig. 1), known as the commit stage, is triggered every time a team member commits a change into the SCM. CI server checks out the committed code and compiles it, runs unit tests and integration tests, and makes code analysis using related tools on it in the commit stage and builds the application binaries and packages. The commit stage provides binaries for later stages. If this stage passes, further stages can be triggered automatically such as automated functional acceptance or capacity tests.

The next stage of a deployment pipeline is the deployment stage. A selected build can be deployed into testing or live environments at this stage. Deployment of the binaries, configuration changes, and database updates to the environments expected to be executed automatically without a manual intervention. Deploying into live environment is usually the final stage in a pipeline [11].

IV. Case Study

The case study presented in this paper describes continuous delivery practice we use to manage release and deployment process as a part of the software development life cycle. As IT department of MKK, we need to continuously push new features into multiple test environments, around 40, every day to support our agile software development [12] lifecycle and be able to make changes to the live environments. Release management process plays a critical role in order us to deliver high quality software and services, lower the development costs, minimize the risk of release failures, and provide faster feedback loops to the business.

In the following subsections, we briefly describe different activities involved in our release and deployment management process.

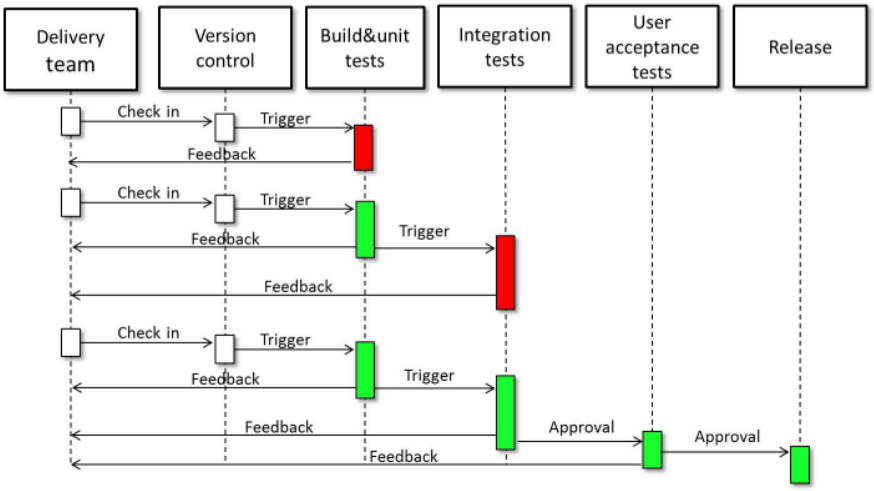

Fig. 1 Continuous Delivery Pipeline

A. CI Process

The CI process is supported by the Jenkins [13] which is an open source continuous integration server and configurable via a Web interface. Its functionality is extendable via plugins and allows integration with SCM systems. Build jobs are at the heart of the Jenkins build process. A Jenkins build job is a particular task or step in the build process. This may involve simply compiling source code and running unit tests or you might want a build job to do other related tasks, such as running integration tests, measuring code coverage or code quality metrics, generating technical documentation, or even deploying application to a web server. CI process of a real project usually requires many separate but related build jobs. A build job can be triggered manually, based on a time trigger, or based on an event, e.g., the completion of another job. Jenkins CI server checks SCM on regular intervals for new commits and triggers the related build job and starts the build when there are detected changes. We use Subversion [14] as SCM tool, FindBugs [15] to detect potential bugs in the code, and Cobertura plugin [16] to report test coverage.

It is our organization's policy that every software project should be managed with Apache Maven [17] which is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

We use single subversion source repository to track code and configuration changes. In the following subsections we will describe the practices of continuous integration [18] we follow in more details:

1. Maintain a Single Source Repository

A single source repository is maintained with subversion in our organization.

2. Everyone Commits to the Mainline Every Day

Developers integrate their code often, at least few times a day, to make sure the changes they have made did not break the state of the software.

3. Every Commit Should Build the Mainline on an Integration Machine

When a member of development team implements a task he runs unit tests locally and commits the changes to SCM. The development team has access to the entire testing tool chain to validate their updated implementation before committing. When the code is committed CI server detects the commit and starts a new build automatically.

4. Establish a Release Opening Strategy

We use Common Branching Patterns [19] as our release strategy. The typical procedure is as follows: Developers commit day-to-day changes to /trunk such as new features, bug fixes etc. When the team thinks the software is ready for release (say, a 1.0 release), the trunk is copied to a "release" branch. /trunk might be copied to /branches/1.0. Team members continue to work in parallel. One team begins rigorous testing of the release branch, while another team continues new work (say, for version 2.0) on /trunk. If bugs are discovered in either location, fixes are ported back and forth as necessary. At some point, however, even that process stops. The branch is "frozen" for final testing right before a release. The branch is tagged and released. When testing is complete, /branches/1.0 is copied to /tags/1.0.0 as a reference snapshot. The tag is packaged and released to customers. The branch is maintained over time. While development team works continuously on /trunk for version 2.0, bug fixes continue to be ported from /trunk to /branches/1.0. When enough bug fixes have accumulated, management may decide to do a 1.0.1 release: /branches/1.0 is copied to /tags/1.0.1, and the tag is packaged and released. This entire process repeats as

the software matures: when the 2.0 work is complete, a new 2.0 release branch is created, tested, tagged, and eventually released. After some years, the repository ends up with a number of release branches in “maintenance” mode, and a number of tags representing final shipped versions.

Maven release plugin [20] is used to manage this process to save a lot of repetitive, manual work and decrease release related errors.

5. Automate the Build

Builds are started on CI server automatically as soon as a change detected on SCM.

6. Automate Testing

Unit tests [21] are executed automatically during the build process by Maven and coverage reports are generated by Jenkins CI plugins.

7. Automate CodeQuality Checks

Findbugs, a code analysis tool, is run automatically during the build to locate potentially vulnerable code and security flaws. It looks for instances of bug patterns and code instances that are likely to be errors. Moreover PMD [22], a source code analyzer, is used to scan source code and look for potential problems like; possible bugs, dead code, suboptimal code, overcomplicated expressions, and duplicate code (CPD). If an error is found during the build, CI server stops the build and sends an error-report to counter parties.

8. Keep theBuild Fast

We continuously monitor the build process and make improvements. We configured and installed Jenkins CI to support distributed builds [23] in the "master/slave" mode, where the workload of building projects are delegated to multiple "slave" nodes, allowing a single Jenkins installation to host a large number of projects, or to provide different environments needed for builds/tests.

9. Test in a Clone of the Production Environment

The same architecture with production environments is used as staging environment. The integrations with 3rd parties are being simulated for test environments also. We leveraged a test data management strategy [24] to subset the production data, while masking the sensitive information, to feed test environments on a regular basis.

10. Make it Easy for Anyone to Get the Latest Executable

Anyone involved with a software project can get the latest or previous build artifacts and run it for demonstrations, exploratory testing, or just to see what changed recently. Jenkins CI has a Web interface to provide all the information about the build stats, build history, and build artifacts as well as released artifacts.

11. Everyone Can See What's Happening

Jenkins CI Web interface is used to communicate the state of the mainline build. Everyone can easily see the state of the build system and the changes that have been made to it. We hooked up a continuous display to the build system - lights that glow green when the build successful or red if failed.

12. Automate Deployment

We have development, integration, staging, and production environments to apply CI. We move release artifacts between these environments multiple times a day. We developed custom deployment scripts to manage the deployment process between these environments. These scripts and deployment procedure is used for both test and production environments. Automation and automated deployment is described in more detail in the next section.

B. Automation

In order to build an effective release management system with continuous delivery, deployment step of continuous delivery pipeline needs to be automated. Deployment process involves installing applications, configuring resources and middleware components, starting/stopping components, configuring the installed application for different environments [25].

We will describe automation of deployment activities in the following subsections under three categories; configuration management, change management, and deployment process.

1. Automate Configuration Management

Continuous delivery pipeline relies upon good configuration management. We follow these principles to manage configuration:

a) Get Everything under Version Control

Good configuration management entails having everything required to create and test your application in version control. We have source code, build and deployment scripts, management scripts, automated test scripts, database change scripts, and application configuration files under version control. During the deployment and release, CI server checks out the necessary files from SCM and executes necessary steps.

b) Automate Dependency Management

The dependency management is a mechanism for centralizing dependency information. Good configuration management requires software dependencies, such as libraries and components are managed in an automated fashion. We use maven dependency mechanism [26] to automate dependency management. We have a set of projects that inherits a common parent configuration (POM) and we put all the information about the dependency in the common parent configuration and have simpler references to the artifacts in the child configurations.

We implemented our local-corporate build artifact repository as a proxy by using Apache Archiva [27] and dependencies and libraries are served from this central repository. When a developer declares a new dependency, maven tries to fetch the dependency from local-corporate repository. If the dependency exists in local-corporate repository, it is provided to the developer, if not, repository manager requests the dependency from remote repositories

and delivers it to the developer while catching a copy for the future requests (see Fig.2). This process is managed behind the scene for dependencies and transitive dependencies by Maven automatically.

c) Automate Infrastructure Management

Infrastructure and environments should be provisioned and managed using a fully automated process as a practice of good configuration management process. We use virtualization to create environments from a set of baseline virtual machine images and custom scripts to configure these environments.

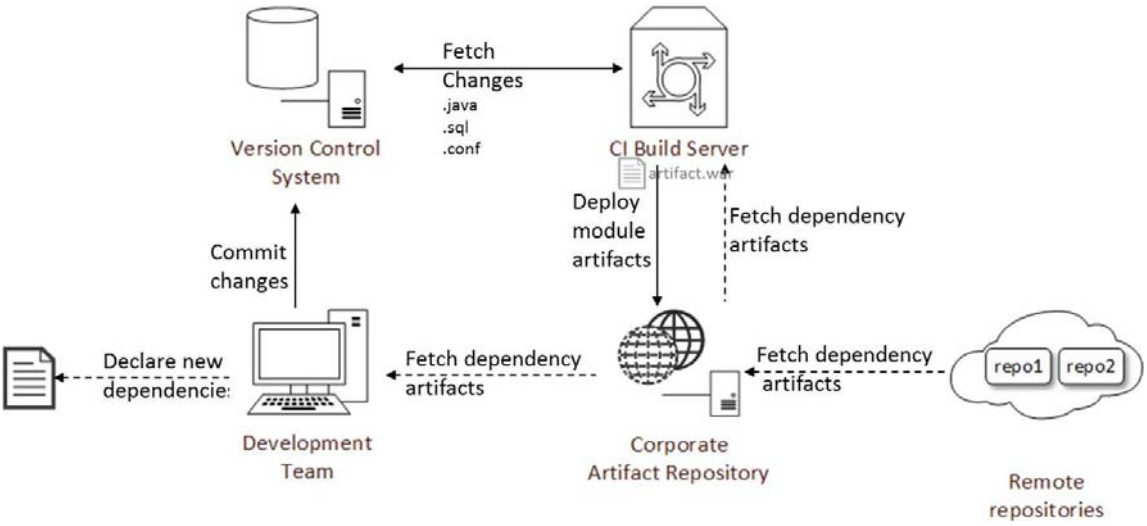

Fig. 2 Dependency Management

d) Automate Database Management

Changes need to be propagated from development to test, and ultimately to production in a controlled and consistent manner. Database is versioned [28] to manage this process. Therefore we have the ability to recreate a database at any point in time. Versioning the database is particularly important when there is a need for creating new environments.

We generated a baseline schema; any changes to the schema require a schema change script. The schema and change scripts are managed with SCM. We automate database updates with dbmaintain [29]. CI server checks out the change scripts and uses dbmaintain to incrementally apply the change scripts to database during the deployment.

e) Automate Application Configuration Management

Application configurations are kept in SCM and managed automatically. We have common configuration files in SCM as templates and generate configuration files for a specific environment from these templates. Generated configuration files are packaged and deployed to local-corporate repository based on application version. These configuration files can then be retrieved based on version number during the deployment.

2. Automate Change Management

Uncontrolled changes are a leading cause of downtime in live environments. It is essential to lock down staging and production environments to ensure that unauthorized changes cannot be made to them. We keep all sources and configuration files in SCM and changes are controlled with pre-commit hook scripts. When a change needs to be introduced procedure followed is described next. Change requests, tasks, or bug fixes come to development team with an issue from issue tracking system JIRA [30]. The issue needs to be assigned to a developer and be fixed for a release version in order to commit a change about this issue to SCM. When a developer tries to commit a change, a pre-commit hook script runs to verify the commit rights of the developer and status of the issue to make sure the changes are allowed to be committed for the specified version of software.

The deployment process is also protected with authorization mechanism. Releases to development and test systems occur on scheduled time intervals or can be triggered by users who are assigned release management role.

Changes are tracked with software versioning model. Software versioning is the process of assigning either unique version names or unique version numbers to unique states of computer software [31]. We have a version numbering schema in our organization that uses a five-sequence identifier; *major.minor[.maintenance[.bugfix[-build]]]*. We version software applications during the development and releases according to this schema and all artifacts are stored in local-corporate repository based on version numbers.

In order to trace released versions of software applications we developed custom scripts to create an “organizational release information page” dynamically. The organizational release information page is accessible by everyone and displays the environment name, release time for every environment, and version number of the current release with a link to the build on CI server and SCM changes. We are able to trace the changes from released versions back to a change set in SCM by using this web page.

3. Automate Deployment Process

We developed custom scripts to distribute deployment artifacts between the stages of deployment pipeline, manage and configure the environments during the deployment automatically. Deployable artifacts are being created once in the build stage of deployment pipeline and the same artifacts are distributed between all stages of the deployment pipeline. This process executes as follows; when a release job on CI server is triggered, it retrieves the application artifacts from local-corporate repository, connects to the database, executes database update scripts, stops the application, installs the new version of the application to target environment, configures target environment, and starts the application. Deployment scripts connect to the server and monitor the log files for errors during the start. If an error found, deployment is stopped and an error report is sent to counter parties, if environment starts successfully, a successful release report is sent.

We use the same deployment mechanism to deploy to every environment. This is essential to minimize deployment related issues and provide consistency. In order to make zero-downtime deployments we built high available middleware architecture [32].

4. Monitor and Improve

Delivery process is continuously measured and improved. We gather metrics as part of the release management process, such as; application build times, frequency of deployments, time taken to perform deployments, time to detect an incident, time to repair an incident, what proportion of changes are successful, and so forth.

Using these metrics, we measure the success of any changes we make, analyze successful and unsuccessful changes to look for patterns, determine the root causes of unsuccessful changes, and categorize them.

V. Conclusion and Future Work

We have presented an approach to implement a release management process with continuous delivery in order to decrease the amount of resources needed and software delivery time while delivering software applications and services rapidly.

Without a deployment pipeline implementation, a manual deployment to only one environment used to be completed in about 1 man-day. This includes application installation, configuration changes, database updates, and restart of the application steps. In order to get a successful release, endless emails being sent, tickets being raised, or other inefficient forms of communication required and of course this becomes a major source of inefficiency. We have implemented CI in 2008-2009 periods and completed the implementation and automation of all stages of deployment pipeline in 2010. Hence the problems with releases were completely removed with deployment pipeline implementation and the release time for an environment decreased to around 5-10 minutes. This improved our software development life cycle and decreased deployment related errors that cause release failures radically. The number of bugs and issues raised for applications in years is reciprocally correlated with the maturity level of continuous delivery process. Fig. 3 shows the number of bugs introduced in years for Central Depository System (CDS) application.

Visibility of the software delivery process is increased, everyone can see which builds are available and they have the ability to easily deploy any version of the software into any environment at the push of a button.

Moreover, deployment pipeline allows testers, operations or support personnel to self-service the version of the application they want into the environment of their choice. Testers can select older versions of an application to verify changes in behavior in newer versions. Support staff can deploy a released version of the application into an environment to reproduce a defect. Operations staff can select a known good build to deploy to production as part of a disaster recovery exercise. Therefore, support requests and reported issues in regards to software maintenance also decreased around 95% since collaboration is automated.

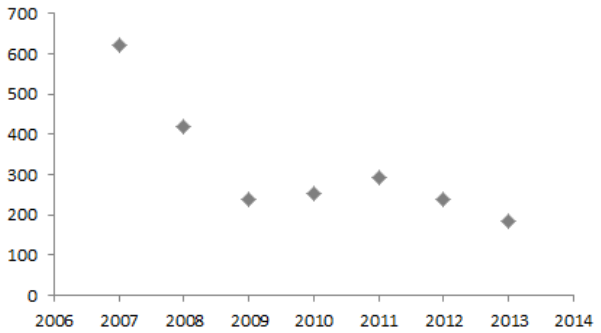

Fig. 3 Number of Bugs Reported for CDS

In the future we will look into more detail at automating infrastructure management, and, in particular creating and configuring lightweight, reproducible, and portable development environments. Number of bugs and issues can be decreased for software applications and caught at early stages of development life cycle by improving test automation.

References

[1] How to drive innovation and business growth, Leveraging emerging technology for sustainable growth, PwC Advisory Oracle practice (2012) (Online) http://www.pwc.com/en_US/us/supply-chain-management/assets/pwc-oracle-innovation-white-paper.pdf Retrieved: 11.08.201.

[2] Continuous Delivery: A Maturity Assessment Model, A Forrester Consulting Thought Leadership Paper Commissioned By Thoughtworks. (March 2013) (Online) http://info.thoughtworks.com/Continuous-Delivery-Maturity-Model.html. Retrieved: 15.08.2013.

[3] J. Humble, Build and Release Principal, Agile Release Management: Towards Frequent, Low Risk Releases, ThoughtWorks Studios. (14 July 2010) p1 (Online) http://www.kn-portal.com/fileadmin/xxx/AgileReleaseManagement-whitePaper.pdf . Retrieved: 20.08.2013.

[4] S. Smith, Introducing Continuous Delivery, (07.04.2014), (Online) http://java.dzone.com/articles/introducing-continuous. Retrieved: 08.04.2014.

[5] Erenkrantz, J. R.(2003) Release Management Within Open Source Projects. In: Proceedings of the 3rd Open Source Software Development Workshop. Portland, Oregon, USA, May 2003, S. 51–55.

[6] Introducing Continuous Delivery in The Enterprise, Xebia Nederland (Online) http://continuousdelivery.xebia.com/sites/default/bestanden/nl/Whitepaper%20Xebia%20Continuous%20Delivery.pdf. Retrieved: 15.08.2013.
[7] M. Fowler, Continuous integration, (2006, May) (Online) http://martinfowler.com/articles/continuousIntegration.html Retrieved: 20.08.2013.
[8] D. Sujoy, R. Amit Kumar, and B. Uttam, Release Management for Parallel Development: A Case Study, Lecture Notes on Software Engineering, Vol. 1, No. 1, February 2013.
[9] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment, Addison-Wesley Signature Series, Automation 2011 p105-106.
[10] M. Fowler, Deployment Pipeline, (30 May 2013) (Online) http://martinfowler.com/bliki/DeploymentPipeline.html Retrieved: 20.08.2013.
[11] J. Humble, D. Farley, Continuous delivery: reliable software releases through build, test, and deployment automation. ISBN 978-0-321-60191-9. – Chapter 1.
[12] Abrahamsson, P., Salo, O., Ronkainen, J., &Warsta, J. (2002). Agile Software Development Methods: Review and Analysis. VTT Publications 478.
[13] Jenkins CI – Meet Jenkins, Available at https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins.
[14] Subversion. Available at http://subversion.apache.org/.
[15] FindBugs, Find Bugs in Java Programs, Available at http://findbugs.sourceforge.net/.
[16] Cobertura, A code coverage utility for Java, Available at http://cobertura.github.io/cobertura/.
[17] Apache Maven, Available at http://maven.apache.org/.
[18] M. Fowler, Practices of Continuous Integration. (01 May 2006) (Online) http://martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration. Retrieved: 18.02.2014.
[19] Common Branching Patterns, Version Control with Subversion For Subversion 1.7 - Chapter 4, (Online) http://svnbook.red-bean.com/en/1.7/svn.branchmerge.commonpatterns.html. Retrieved: 25.03.2014.
[20] Maven Release Plugin, Available at http://maven.apache.org/maven-release/maven-release-plugin/index.html. Retrieved: 25.03.2014.
[21] Junit, A programmer-oriented testing framework for Java, Available at http://junit.org/
[22] PMD, Source Code Analyzer, Avaliable at http://pmd.sourceforge.net/
[23] Distributed Builds with Jenkins CI, (Online) https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds. Retrieved: 26.03.2014.
[24] 5 Best Practices for Test Data Management, Kimberly Madia, (July 26, 2013) (Online) http://ibmdatamag.com/2013/07/5-best-practices-for-test-data-management/ Retrieved: 26.03.2014.
[25] R. van Loghem , So What Is A Deployment Really, (July 8, 2009) (Online) http://blog.xebia.com/2009/07/08/so-what-is-a-deployment-really/. Retrieved: 27.03.2014.
[26] J. Casey, V. Massol, B. Porter, C. Sanchez, J. V. Zyl, Better Builds with Maven, The How-to Guide for Maven 2.0, 2008 p61
[27] Apache Archiva, The Build Artifact Repository Manager, Available at http://archiva.apache.org/.
[28] K. Scott Allen, Versioning Databases – The Baseline, (February 1, 2008), (Online) http://odetocode.com/blogs/scott/archive/2008/01/31/versioning-databases-the-baseline.aspx. Retrieved: 26.03.2014.
[29] Dbmaintain, Tool for automating the deployment for relational databases, Available at http://www.dbmaintain.org/overview.html
[30] JIRA- issue tracking tool, Atlassian, Available at https://www.atlassian.com/get-jira .
[31] Wayne A. Babich. Software Configuration Management. Addison-Wesley, 1986. 162 pp.
[32] Clustering/Session Replication, Apache Tomcat Version 7.0.52, Feb 13 2014, (Online) https://tomcat.apache.org/tomcat-7.0-doc/cluster-howto.html. Retrieved: 31.03.2014.