Continuous Delivery: How hard can it be?

Anders Buhl dat13abu@student.lu.se
Mikael Jarfors nat13mja@student.lu.se
Dept. of Comp. Science, Lund University

The fast development rate of modern software projects and high expectations on the time to market and quality of software demands faster and more reliable software development methods. Continuous delivery is the extreme combination of continuous integration and frequent release where every new task will be made into a new release. The customer will always have access to the latest version which provides fast and continuous user feedback.

1 Introduction

This report will present the results of our research on the effects of continuous delivery (CoDe)on a small scale project developed by undergraduate students. In addition to the project setup and the tools available to all development teams, the teams in the case study also had tools to support continuous delivery. The students did not have any prior knowledge about the system and had to get to know the system during the project. Previously, the team had tried using a continuous integration system before allowed them a basic understanding of the principles behind it.

The purpose of the research is to evaluate how the release process is affected by adopting CoDe. The main research questions are; how time spent on creating a release can be decreased by adopting CoDe, how the number of failed releases can be decreased by adopting CoDe and if a student project of this size will be affected by CoDe. Additional interesting areas of research are what technical problems that can occur, what benefits CoDe bring to the team and the project, if the release process can be made less chaotic, if it is possible for a student team to keep the system releasable at all times and what effect CoDe will have on the developers confidence.

Continuous delivery is an approach to software development, which adopts continuous integration and frequent releases to the extreme and makes every new story into a release. This means that the system has to be kept in a releasable state at all times and the customer will at any given moment in time be able to tryout the latest version within a couple of minutes.

The in-depth study is conducted on two teams studying the course software development in a team (Programvaruutveckling i grupp, EDAF45). The course is mainly directed to students studying the second year on computer science but is optional for other students studying at LTH. The teams consists of ten developers and each team is supposed to follow the XP developing methodology. One of teams (referenced to as team04) was a homogeneous group of male students studying the second year on computer science, except one, who was studying the third year of computer science. The other team (referenced to as team03) was a more heterogeneous group of students who had chosen to study this course as an optional course. This team was a combination of second, third, fourth and fifth year male and female students. The study was done as a part of the parallel course coaching of programming teams (EDA270), where older students acts as coaches to the development teams.

The main goal of the course EDAF45 is to learn how to develop software in a team environment through practical experience learn how to develop software in a team. Each week there is focus on one or more of the XP principles (also called team goals), e.g. test-driven development, pair-programming etc. One week corresponds to one sprint and starts of with a planning meeting. During the planning meetings, the teams reflected on previous sprint and plan for the next sprint. The customer presented new stories each planning meeting which were in turn estimated and prioritized. After the planning meeting each student were supposed to spend four hours on a spike assignment which was to be presented in the beginning of the programming session. The programming session was eight hours and usually there was a release with a deadline six hours into the programming session.

2 Background

This in depth study is based on a few yet relevant aspects, where the studies provided inspiration from earlier in depth studies as well as from the principles of eXtreme Programming. This section will discuss how previous studies and XP led to setup for this study.

2.1 Related work

Earlier attempts to adopt continuous integration and CoDe in the students projects have been made. One in-depth study focused on using Jenkins as a continuous integration tool and looked at all benefits. The result was higher test coverage and fewer failed builds. The coaches chose to provide the team with build script and a pre-configured Jenkins server [9], this inspired us to have a similar approach in our setup. Our research was more directed to the release process and we chose to focus on how the release process would be made simpler and how it would change when using CoDe. Since test coverage would be higher and there would be fewer failed builds each release would be more stable and the chance of a failed release because of a failed build would decrease.

Eriksson and Gärtner used an iterative approach in teaching their team to adopt CoDe. In this study they focus on teaching the students how to adopt CoDe and let the team discover the benefits of CoDe. Also letting the team develop the CoDe system including build script and build server would give the team more insight into how the system works and make the team appreciate CoDe more. The team manage to setup a working CoDe system within the time frame of the course [7]. These results are interesting to see how a student team responds to CoDe and how it can be implemented. It also gives an indication of the effects on adopting CoDe on a student project. Our focus will be more on the benefits and challenges of adopting CoDe and to see how the team responds to CoDe when using a pre-configured system.

Previous work presents the result of adopting CoDe in a real company. It presents the benefits and challenges of introducing CoDe into a team which is unfamiliar with the principles of CoDe. The benefits presented were accelerated time to market, increased user feedback, improved productivity and efficiency, reliable releases, improved product quality and improved customer satisfaction. Challenges they faced when adopting CoDe was operational challenges, process challenges and technical challenges. These benefits and challenges are results to be expected when adopting CoDe. We expect the benefits and challenges to be similar in a student project [6].

2.2 XP

Extreme programming, XP, is an agile development method. Agile development is a methodology where the main idea is to embrace change and not plan ahead more the absolutely necessary. Planning, analysis and design is done iteratively in small steps for each cycle in development. Development cycles are shortened to be able to easier adopt to changes. Changes in software can be very costly and the cost of changing software increases over time. By embracing change and not plan ahead the cost of change is reduced [3].

Each development cycle, sprint, is divided into two main parts: planning game and development. During planning game the development team estimates the cost of implementing features. These features are described by the customer as small user stories, called stories in XP. The customer prioritizes which stories should be implemented during the next iteration according to the estimations done by the development team. In the development phase of the iteration the developers will work in pairs and picks a story to implement according to the customers priority [3].

XP have 12 principles (see figure 1) which describe the central parts of XP. Continuous integration and frequent release are the most important practices when adopting CoDe. Continuous integration is the principle that new code should be integrated as soon as it is developed. For each integration the system will be built and all tests will be run. Releases will be created frequently with small changes. CoDe combines these two principles to the extreme that every implemented task will result in a new release.

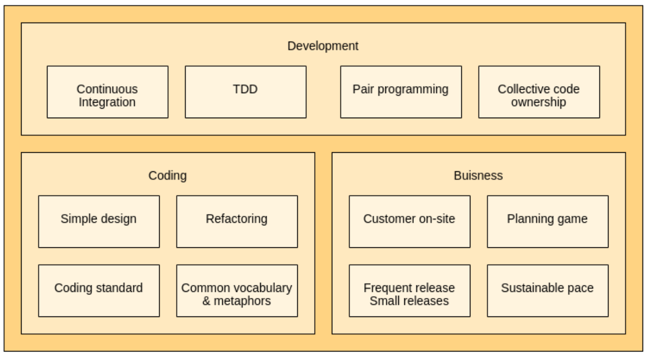

Figure 1: *The 12 practices of XP [3]*

2.3 Project setup

In order for our two teams to adopt continuous delivery from the start, a Jenkins integration server was provided. The system was comprised of a Jenkins server running on a home computer, such that it had access to a public IP address. The server initiated builds through a web hook that was set up through the admin tools in Bitbucket. Initially the build pipeline was a simple maven script that ran all the tests and then generated a jar-file from the contents of the repository.

Moreover, the Jenkins system was chosen for this project for a few reasons, the primary reason was that it was the most powerful free tool that was a available at the time of starting this project. An alternative that presented itself was Bitbucket pipelines, which seemed a viable alternative, however due to the 30 day trial period and the project running for 6 weeks, it was disregarded as an alternative. Furthermore, Jenkins has a tremendous user base, which in turn means that there are plugins for all the major version handling tools such as git, Bitbucket and SVN.

3 Continuous Delivery

The fundamental idea of continuous delivery is to be able to fast, efficiently and reliably deliver high quality software. To achieve this the releases have to be frequent and automated. This is supported by the XP principles continuous integration and frequent release (see section 2.2). The release process has to be automated to be repeatable and fast [8]. Continuous integration requires a system to automate building and testing the system. By adding an additional step to produce and package executable binaries, documentation and external libraries etc. The integration and delivery pipeline will run at every commit and produce a new release.

3.1 Continuous Delivery related principles

CoDe requires developers to follow certain principles when developing the system. These principles are a part of the XP principles and are important to be able to guarantee reliable and high quality software. CoDe relies heavily on the quality of tests, fast feedback and the ability to keep the repository releasable a all times. In addition to the three XP principles, clean repository is also an important principle to be able to keep the repository releasable.

- **TDD**
  An important design and architecture activity in agile development to keep the design simple and efficient. Unit tests written during TDD are used for regression testing to guarantee that existing functionality is not removed or broken. TDD is used to form a safety net through test cases that ensure functionality is present, and thus protects existing functionality during new commits in a CoDe environment.

- **Continuous Integration**
  It is important to continually integrate new functionality into the personal workspace to reduce the risk of merge conflicts and workspace diverging in different directions. To be able to create a new release as fast as possible it is important that new features and fixes are integrated often. Continuous Integration is needed as each task in CoDe is created into a release, if the code is not being continuously integrated, no releases will be created.

- **Frequent release**
  The idea to release weekly or daily is taken to the extreme in CoDe and a new release can be created on a minutely basis. Frequent releases support the idea of an accelerated time to market. This means that features that are implemented early in a release cycle are not artificially delayed through release cycles, instead is immediately released into the product.

- **Clean Repository**
  As each subsequent integration is to become a release, it is key that the repository is kept clean.

A clean repository means being able to build the system and have all tests pass.

3.2 Configuration management

Configuration management is key in order to adopt continuous integration, as the developers need to be able to continuously share their changes with other developers as well as integrating new changes to their own workspace. Continuous delivery can be aided or hindered by a few areas of configuration management such as branching.[12] In the project we chose to not focus to much on advanced configuration management methods and this is a part that could be researched further.

3.2.1 Branching

Branching is a key aspect of software development, where allowing your developers the freedom to experiment while not needing to be worried about breaking any of the established functionality of a release. Branching can be used in continuous delivery scenario in order to separate the master branch from any malicious code.

On the other hand, branching poses a few disadvantages such as creating the issue of double maintenance, meaning that the team needs to maintain two separate projects instead of one. This problem could be avoided by introducing feature toggles as a way of introducing branching.

3.2.2 Feature Toggles

Feature toggles can be used in continuous delivery projects in order to avoid double maintenance. In the works by Neely and Stolt, feature toggles are embraced as a method of using continuously pushing subsets of a story without breaking functionality. This was done through incorporating placing unfinished story behind a feature toggle, which could be toggled on at a point where the functionality had been implemented [12].

4 Application

The Jenkins server that was initially set up during the first iteration, went through multiple changes and additions in order to better aid and suit the project needs. The plugins used where the Ant Plugin, Jacoco Plugin, Maven Plugin, Project Based Matrix Security, Bitbucket Build Status Plugin, Slack Notifier Plugin, Bitbucket Oauth Plugin and Bitbucket Plugin.

4.1 Tools

- **Maven Plugin**
  The generation of deployable jars was done through a Maven build script [11], this plugin ran the build script on the Jenkins server
- **Ant Plugin**
  The generation of Jacoco test coverage needed an Ant script instead of Maven in order to function properly [1]. The reason behind using Ant as an aid, was that the Jacoco plugin for Maven struggled with a multiple module set up.
- **Jacoco Plugin**
  This plugin published the Jacoco coverage reports to the Jenkins build page.
- **Project Based Matrix Security**
  Project Based Matrix Security was used to restrict access of users to their specific project, as well as limit their ability to configure and sabotage the Jenkins set up.
- **Bitbucket Plugin**
  The Bitbucket plugin enabled integration between Bitbucket and Jenkins, such that once a push was made to Bitbucket, the Jenkins server was notified by a webhook to pull the latest changes and build the system.
- **Bitbucket Build Status Plugin**
  This plugin handled feedback in the Bitbucket client (As seen in Figure 2).

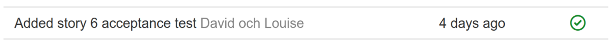
Added story 6 acceptance test David och Louise 4 days ago ✓

Figure 2: *Bitbucket Notifications*

- **Bitbucket Oauth Plugin**
  Bitbucket Oauth Plugin was used to handle secrets and keys in order for the Bitbucket build status plugin to function.
- **Slack Notifier Plugin**
  This plugin pushed notifications to slack about the status of a build after each commit. (As seen in Figure 3).

  Our pipeline consisted of compiling the program and running the tests through Maven, and then rebuilding the system with ant in order to produce test coverage reports with Jacoco. Lastly,

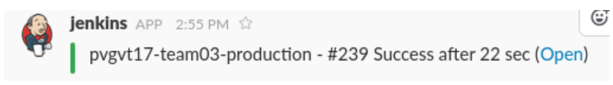
jenkins APP 2:55 PM
pvgvt17-team03-production - #239 Success after 22 sec (Open)

Figure 3: *Slack Notifications*

the tests, artifacts and reports were published on the build page of Jenkins. An illustration of the pipeline can be seen in figure 4.

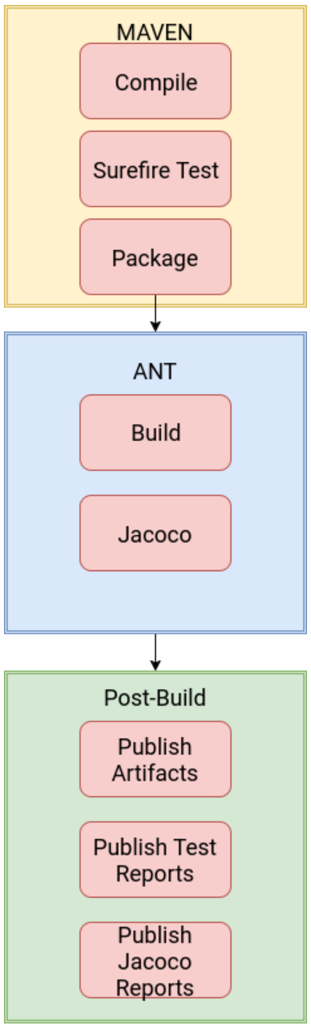

Figure 4: *Visualisation of the Pipeline*

5 Results

This section will present the results of introducing CoDe in a student software development team and also present challenges related to CoDe faced during the project. The results were gathered through observation of the team, a survey sent to all students participating in the course and through discussion with other coaches in the course.

5.1 Benefits

The main benefit from adopting CoDe was continuous feedback. The teams immediately discovered if a commit failed any test cases or if the build failed for any other reasons. Team03 had easier access to the feedback since they were notified through Slack with every commit and whether the build succeeded or failed. The release process was less chaotic and faster for the teams compared to what we were told by other coaches. On the other hand team04 had a big problem with non-functioning releases which resulted in them having to fix the problems and redo the release process.

The teams almost managed to keep their repository clean from failing test cases and when it happened the team was notified very fast. This helped the students understand how important and difficult it is to keep the repository clean.

5.2 Challenges

This section will state the challenges we experienced when utilizing Jenkins and CoDe, it will focus on 3 key areas presented in [6]. The results in this section are some common problems with adopting CoDe and it also contains challenges which could have been avoid if planned for.

5.2.1 Organizational Challenges

We did not encounter any severe organizational challenges since the organization of the team was built from the ground up during the project. The only organizational challenge we encountered was related to the organization of the course and institution. The computers used during the project did not allow the students to install additional software and since Maven was not installed they could not execute the Maven build script locally before committing to the repository.

5.2.2 Process challenges

In CoDe every commit would result in a new release and the software should be kept releasable all the time, but still the release has to be of sufficient quality. This required the development process to have a clear definition when a feature is done and when to commit the feature. Every story was too big to be committed all at once and required to be divided into smaller tasks and test cases. This was a problem in both teams and it was not clear when a feature was committed and often a half finished task (or story) was committed which resulted in a release containing some functionality which was not completely implemented. This forced the team to manually check the release before it was deployed to the customer.

Build scripts are used to make sure that the system is built, tested and packaged the same way every time, i.e. that the build process is repeatable. Developers should use the same build script as the integration server during development when testing new features. This was not possible in this project (see section 5.2.1) and introduced some process challenges since a build could fail on the integration server even though developer had tested the system locally.

5.2.3 Technical Challenges

The largest issues that we encountered during the projects lifetime consisted of technical problems. The first issue that the teams encountered was that Maven required the team to follow a specific packet structure in order for the tool to build correctly.

Furthermore, the versions of JUnit and JDK used posed an elusive issue as it was not clear that this was the issue, however it was solved easily through specifying the version of each of the tools.

Moreover, one of the earlier points of discussions was whether the team members should have access to the Jenkins system, and how this would be done. During the first iteration of the project the matrix based security managed to lock out all of the users from the system and a complete server reset needed to be done through deleting the security settings.

Thirdly, both teams found that at some point in the development process, there was a need to include external files in the JAR file. In the case of Team 03, an automated GUI construction tool was used, which created .FXML files that needed to be included in order for the GUIs layout specified. In the case of Team 04, the team had decided to include an external library called TimeSafe in order to construct their configuration files. In both these cases it proved difficult to include the needed files and libraries in the JAR file.

Lastly, the Jenkins server did not support building more than the main branch of the system and we chose to not add this functionality to the due to lack of time and also we did not want to encourage the students to use branching.

5.3 Group Dynamics & Trust

The challenges, related to CoDe, faced during the project affected the students trust in the system. The dynamic of the group was affected when the system did not work as expected and it was hard for the students to accept that some test cases that passed on their local machine could fail when executed on the Jenkins server. Team03 almost missed a release deadline when the Maven build script did not include all required files in the executable JAR-file. This had a great negative effect on the team and morale were really low until the issue was solved by us, the coaches. A similar situation occurred in team04 and it had even greater negative effect on team04, because they were more inclined to doubt the system.

5.4 Survey

A survey was conducted after the third release during the fifth programming session. We chose to conduct the survey at that time to give the students sufficient time to get used to work in the team and to get well acquainted with CoDe. We did not want to conduct the survey after the last release during the sixth programming session since the project would have ended and it would probably give the students a different perspective than they would have during development.

The survey was conducted to be able to compare the release processes of each team. The focus of the survey was the release process, the students perception of their releases and release process and whether the team had adopted CoDe.

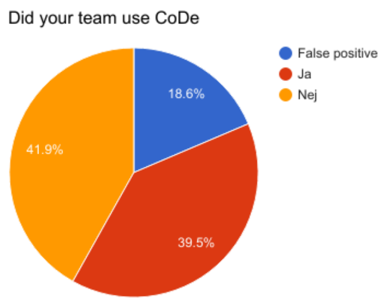

Figure 5: *The use of CoDe in the course. Q: Did you team use code? False positives answer was 'yes', but was a misunderstanding of CoDe*

The survey did have some problems. The first problem was that only 44% (43 out of 98) of the students answered the survey. Also many students (18.6%) had misunderstood the concepts of CoDe and answered that their team used CoDe even though the team did not use CoDe (see figure 5).

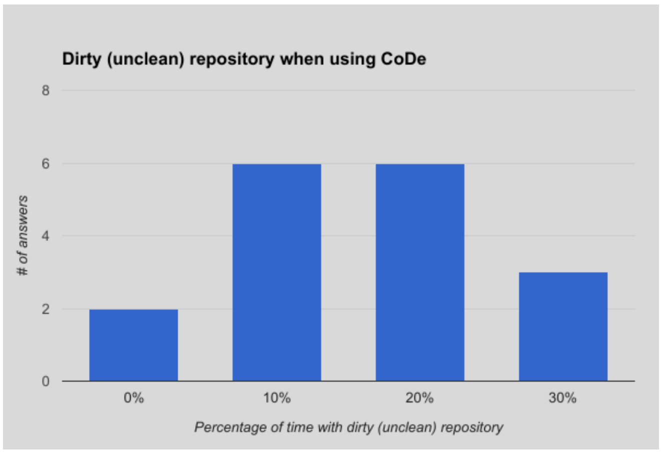

Figure 6: *Part of the project with dirty (unclean) repository when using CoDe (false positives not included)*

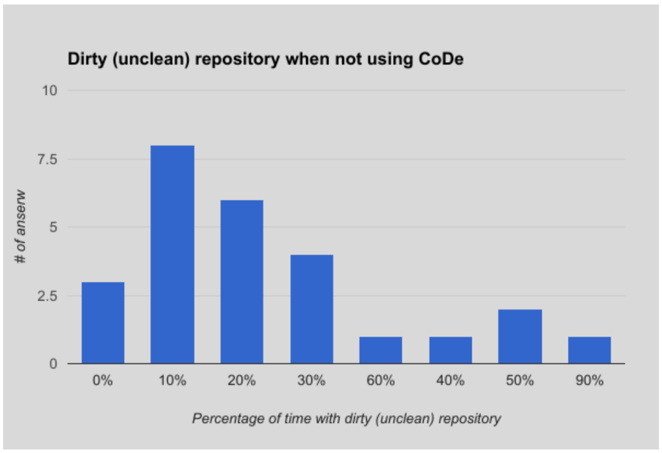

Figure 7: *Part of the project with dirty (unclean) repository when not using CoDe*

Figure 6 and figure 7 presents the amount of development time where the repository contained 'red' code, i.e. code that does not compile or code with failing test cases. Important to note is that this is the students view and may not actually reflect the real numbers. What can be interpreted from the survey is that it is a small improvement to when using CoDe. From our point of view which came as a result of discussion with other coaches, resulted in a different perspective where the improvement was more significant than the result of the survey.

With the use of Jenkins the teams got continuous feedback whether a build succeeded or failed. A build failed if it did not compile or if there were failing test cases. Other teams would have to manually check if

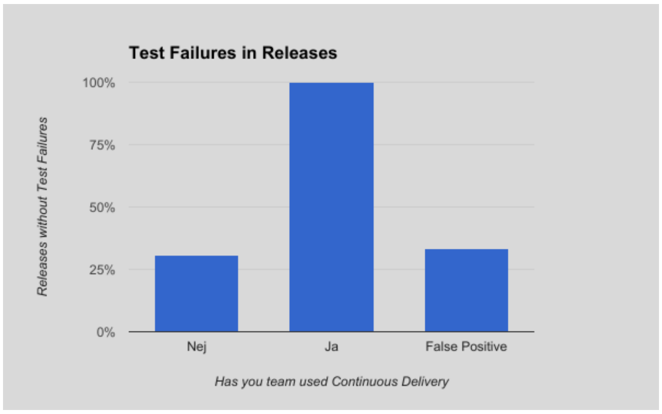

Figure 8: *Part of teams which sent a release with failing test cases comparing CoDe ('Ja') and non-CoDe ('Nej') (false positives could be considered non-CoDe)*

someone pushed a commit with failing test cases and this would probably result in more commits containing failing test cases. The number of releases sent to customer with failing test cases was significantly lower for teams adopting CoDe, which would indicate that number of commits with failing test cases was lower (see figure 8).

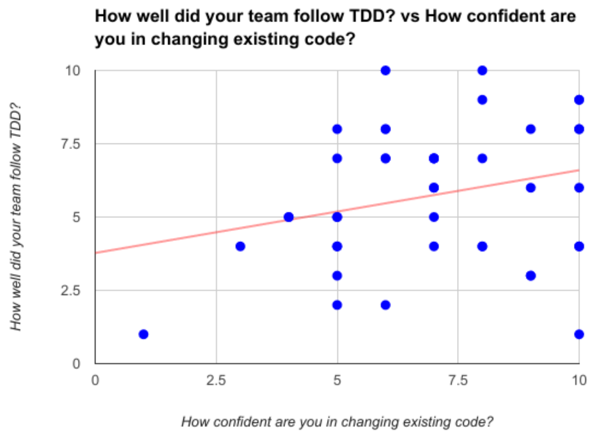

Figure 9: *The students were more confident in changing existing code if their teams followed TDD*

Tests are an important part of CoDe and is also a central part of the XP principles and thereby a central part of the course. It has also been in focus in our research and figure 9 presents how good the students felt their team were at adopting TDD vs. how confident they felt changing existing code. The trend line in figure 9 indicates that there is a correlation between TDD and collective code ownership.

Section 2.1 presents the known benefits of introducing CoDe, e.g. increased user feedback, improved product quality and accelerated time to market. In

Table 1: *Comparison of CoDe vs. non-CoDe*

	Time to create release (0-10)	Confidence in system (0-10)
CoDe	2.35	5.94
Non-Code	2.46	5.96

the student project this would ideally result in faster release process and a higher confidence in the system. The time spent on creating a release is not the measured time, instead it is the students perception of the time to create a release. The result presented in table 1 shows that there was almost no difference in time spent on a release and the confidence in the system.

5.5 Test-driven development

Testing is an important aspect of CoDe which acts as a safety net in order to guarantee the quality of the software. The survey indicated that the students became more confident in changing existing code when adopting CoDe. From the results presented in table 2 it is clear that Team03 had a better testing process, since they had higher test coverage and more test cases. It was noted that Team03 had discussions regarding the testing process and aimed to better adopt TDD and as a result the higher number of test cases indicated that they were better at adopting TDD than team04.

Table 2: *Test coverage*

Team	Number of test cases	Test coverage
Team03	93	80.3%
Team04	38	67.7%

6 Discussion

The result we obtained was inline with our predictions, however we ran into more challenges than expected. The difference when comparing the student teams was not significant which indicates that CoDe does not have great effect on small projects or that it is hard to implement and make the team adopt to CoDe within the time line of the project. We would argue that even if the results did not show a significant difference the problems the teams faced and the reasons for delayed releases were different and both teams experienced the benefits of CoDe.

The survey had a very low response rate and the answers were subjective. Some of the results in the survey could have been collected in better ways, but due to lack of time and resources the survey was constructed and conducted in this form. However, the survey gave some indication on the differences in the teams and most of these differences displayed the benefits of CoDe.

We did not encourage the teams to use branching, since we did not want the configuration management methods to become too advanced. But we did not stop the teams from using more advanced configuration methods, e.g. branching. Team03 utilized branching in some different scenarios and for one release they created a release branch which was not built on Jenkins and thereby bypassed the continuous delivery system and the release process had to be done manually (see section 5.2.3).

6.1 Cost vs. benefits

The cost of implementing a continuous delivery system was not part of the project and was not developed by the team. In terms of the project the cost was the technical challenges related to the system and the negative effects these challenges had on the team.

Compared to the benefits presented in section 5.1 we think that the benefits outweigh the cost. Further more the cost and challenges faced could be reduced and prevented if the system och processes is developed properly.

6.2 TDD vs non TDD

Team03 had higher adoption rate of TDD in comparison with Team04 during release 2, where both teams believed that they had a functional release. However, during the final test of the runnable files in the release showed that Team03 had a correctly functioning program whilst team04 found faults in their program. Reflections during the following planning game showed that some of the team members came to the conclusion that the program lacked tests for essential features that revolved around output of files, which proved to be the failing factor in the release as well. In this specific yet real world scenario one could see that test driven development rewarded the team that embraced it.

7 Future work

This is an area where a lot more can be researched and there are some aspects we discovered during the project. What could have made it easier for the students to adapt to CoDe and make it possible for the students to manage the system, was if they were to develop the build scripts by themselves. The integration server could be provided pre-configured to the students with the possibility to run the build scripts developed by the students.

7.1 Branching

During the process of the student project, the concept of adapting a development branch, a master branch and a release branch was often a matter of discussion. During a future iteration of a similar project, one could adopt other scenarios, such as letting the students work with a development branch that only merged into the master once a build was successful.

7.2 Feature Toggles

Feature toggles could be useful features to adopt in a similar project, i.e implementing features that can be parallelized through architectural patterns such as template och strategy methods are perfect examples of functionality that can be hidden behind feature toggles.

8 Conclusions

In conclusion, continuous delivery is costly process in any size of project, however the student developed project saw greater benefits as the time to create the continuous delivery pipeline did not consume development time. In contrast, the students often found themselves antagonizing the continuous delivery tools, which could be solved through the students developing the tools themselves.

Furthermore, teams that use a continuous delivery scenario need to be skilled in the art of test driven development as shown in our case studies, development teams that do not fully adopt test driven development may be left with dysfunctional releases.

Lastly, continuous delivery was found to streamline the release process in terms of that the students were able to release at will. However the success of the release itself came down to the teams adoption rate of the XP-principes and values.

9 Acknoledgements

We would like to thank teams 03 and 04 from the EDAF45 course of 2017 as well as the peer reviews that were generously done by Jesper Olsson, Mattias Gustafsson, Joel Klint and Oscar Rydh as well as our mentor Lars Bendix.

References

[1] http://ant.apache.org/ [2017-02-17]

[2] U. Asklund, L. Bendix, T. Ekman, Software Configuration Management Practices for eXtreme Programming Teams, in proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques - NWPER'2004, Turku, Finland, August 17-19, 2004

[3] K. Beck, Embracing Change with Extreme Programming, IEEE Computer, Oct 1999, p. 70-77

[4] L. Bendix, T. Ekman, Software Configuration Management in Agile Development, in: Agile Software Development Quality Assurance, Information Science Reference, Feburary 2007

[5] J. Bosch, D. Ståhl, 2016, Cinders: The continuous integration and delivery architecture framework, Information and Software Technology, Volume 83, March 2017, p. 76-93

[6] L. Chen, Continuous Delivery - Hugh Benefits, but Challenges Too, IEEE Software, Volume 32 Issue 2, March 2015, p. 50-54

[7] M. Eriksson, E. Gärtner, Coaching an inexperienced agile team towards a continuous delivery methodology. 2015 p. 1-11.

[8] D. Farley, J. Humble, Continuous Delivery, Upper Saddle River, NJ : Addison-Wesley, cop. 2011.

[9] J. Hembrink, P. Stenberg, Continuous Integration with Jenkins. 2013. p. 1-8.

[10] https://jenkins.io/ [2017-02-17]

[11] https://maven.apache.org/ [2017-02-17]

[12] S. Neely, S. Stolt, Continuous Delivery? Easy! Just Change Everything (well, maybe it is not that easy), in proceedings of Agile 2013, Nashville, Tennessee, August 5-9, 2013