Continuous Delivery - effects of iterative implementation

Mattias Gustafsson and Jesper Olsson
{dat13mgu, dat12jol}@student.lu.se

***Abstract*— Previous studies have established that Continuous Delivery (CoDe) is a benefit for agile teams. In this essay we will compare the differences between two teams that have iteratively develop their own CoDe pipeline to two teams that have gotten a pre-existing set of CoDe tools set up in advance. Through measuring how the teams deployments went and their understanding of CoDe we compared the two. We will find that implementing CoDe implies an overhead knowledge making it difficult to set up iteratively and having time to use the benefits it gives in the short timespan of six week.**

1 Introduction

As children we've all been told "stop doing that" as a response to an action that hurts. That way of thinking haunts many people to this day. In software development there are many things that hurt, that slows developers down. Integrating code, writing extensive unit tests, updating manuals and technical documentation, there are different ways of coping with challenges like these.

Some companies might try to integrate as seldom as possible to avoid developers getting hindered in their problem solving, paying off the technical depth when the customer demands a release. That type of mentality that sweeps problems under the rug will lead to developers needing to go in the the infamous crunch.

Kent Beck has a different view over development which he outlines in his book "extreme programming explained - embracing change". Two principles he puts out in contrast to the earlier examples are continuous integration and 40 hour work week. By working iteratively in small steps and integrating often with the main code base we find problems earlier and can address them before they grow.[1]

CoDe, short for Continuous Delivery, takes continuous integration and ups the ante. By forcing developers to do the things that hurts often, suddenly a new need occurs. They can do the work manually, finding stable branches, test extensively and try to export it to shippable state. Or they can focus on making the processes automated and repeatable thus making the things that hurt, not hurt.

In this essay we examine the benefits of letting a team with a limited time budget implement several steps towards a fully automatic release and integration process. We will discuss whether it is worth dedicating development time to establishing a pipeline or if it is better to buy an existing set-up or not have Continuous Delivery (CoDe) at all. Specifically we will look closer on how continuous delivery impacts the team's releases.

The essay has four parts: in the background section CoDe as a concept is explained and the teams participating in the study described. Next in the method section we discuss what and how data was collected and how we will use this data. The data itself is then presented in the result section before, in the final section, we draw our conclusions.

2 Background

In this section we will discuss the use of CoDe and the context in which this study was performed. We will also mention earlier studies and talk about our reasons for choosing to let the participating students implement the solutions on their own.

2.1 The PVG course and teams

The Computer Science program at Lunds Tekniska Högskola has a course "Programvaruutveckling i grupp" (Software Development in Teams)[2] in which the students take part in a larger software development project. This is done in teams of ten people, led by one or two senior students, coaches, who have taken the course before. The students have a basic understanding of programming but have mostly worked only on small assignments before. Of the teams, we have coached two, referred to as team 05 and 06 in the text.

In the course, the students are introduced to Extreme Programming (XP), an agile development method which is used during the project. XP has twelve core principles, including Test Driven Development, Pair programming and Shared code ownership. Following these principles takes some getting used to and the project is laid out in such a way that conflicts and issues are likely to arise (for the sake of learning of course). There will also be deployments to a customer at several points, usually resulting in frantic, chaotic work leading up to them.

The project takes place in the span of six weeks during which there is an eight-hour lab session each week, as well as a two-hour planning session and four hour

“spike” time where each student gets assigned tasks that benefit the team but does not involve committing any code to the team repository. The project has in advance been split up into different user stories. The stories gets estimated during the planning sessions and implemented during the programming sessions.

2.2 What is Continuous Delivery

“Software release should be a fast, repeatable process.” says Jez Humble and David Farley in their book on Continuous Delivery. Releasing software is hard, integrating code and working in the same directions are the cause of many common problems in software development.[3]

The core idea of Continuous Delivery is to maximize feedback from the code. Focus lies on repeatable and automated procedures that goes from code to an audited release with one press, while excluding as many variables as possible such as manual configurations.[3]

Some of the most common elements in a CoDe pipeline are automation of tests and build tools. For a more advanced pipeline a server that listens to the code base and automatically audits the code and builds a release can be implemented but it is no requirement.[4]

According to agile guru Martin fowler, CoDe increases safety for developers and customers. As problems that arise in the final stages of deployment come up quicker and can get ironed out before clustering up into complex problems. Developers can also get more useful feedback from the customer who can see the features in a more holistic way.[5]

2.3 Related work

A similar study was done by Erik Gärtner and Malcolm Eriksson in 2015[6]. They coached teams into integrating CoDe during the course of their project. They let their team iteratively implement a CoDe pipeline and afterwards measured the team’s understandings of continuous integration compared to other teams. They concluded that letting the team integrate CoDe was good for their understanding but noted that other shortcomings of the team made it less useful than it could have been.

In another study, Hembrink and Stenberg[7] coaches a team to use a Jenkins server for CI. In this case, the coaches set up the system beforehand and provided the team with assistance as they expanded upon it. They concluded that using CoDe is highly suitable for the project and that it increased the test-coverage and minimized the number of failed builds.

From these studies we can see that introducing CoDe is viable and that it does have a positive impact on the projects. Both these studies have been conducted in two very different ways: either the team got a process pre made by the coaches or the team got to develop their own process. We will look at the differences between these methods but we are also interested in whether or not it is a good idea to wait with integrating CoDe until the teams have experienced some of the chaos of their first release.

3 Method

In the following sections we describe the process we used to introduce CoDe to the teams and how we measured their progress.

3.1 Steps for introducing CoDe

1) During the first lab session the team got a chance of familiarize themselves with working in a team environment and during the second, they had to do a release. They were allowed to prepare for the release but it was done manually.
2) We then Introduced CI through spikes. After the first release, some team members were tasked with researching tools for continuous integration to see if they could get them to work. These would then be used on the project so that the students would get warnings or errors if they tried to push commits with failing tests.
3) Next we introduced build scripts. In this step the students had to create a script to build the project to facilitate future releases. This included exporting the code into runnable jar files and including relevant files and folders.
4) With CI and build scripts done, the next step would be to introduce automated acceptance tests. Ideally, acceptance tests provided by the customer should be run on generated builds rather than as unit tests run in the developer IDE. This requires extended scripts and redesigned tests. Thereafter the teams would integrate these new acceptance tests into the commit process so that they are automatically run before a commit is integrated into the master-branch.

We have coached two separate teams to iteratively improve their CoDe process following the previous steps. Two additional teams coached by another pair were provided a working CI solution from the beginning and will be used as comparison.

3.2 Working with spikes

Spikes are home assignments used for researching and experimenting with things that the team does not want to spend time on during actual development. Any task can

be a spike as long as it does not affect the shared code base. Most spikes were done in pairs. All preparatory work for the steps in section 3.1 were done during spike time.

3.3 Why not just set it up in advance?

It would have been easy to prepare a set of tools for the team to use in the project. We are also simulating a situation in which the developers themselves sets up the frameworks which is a common scenario when companies tries to become more agile. Our most important goal is for the students to learn as much as possible during these weeks, therefore we would rather see them get involved in the process.

A fully established pipeline might smoothen out some of the early bumps we want the team to face. Ideally, the teams would first experience the problems of their first release and then see a clear difference when they have a CoDe pipeline in place. To do this we wait with CoDe for the first week and start with it after the first iteration.

With that said, many of the tools used for CoDe are more advanced than what most of the students are used to (they may require knowledge of Ant or Maven, etc.). They are given very little help on the subject as researching these tools was considered home assignments (spikes) for them to do between lab sessions. As a result, it is not certain if this approach is viable for all teams and it may take too long for them to get things to work for it to be useful.

An interesting alternative would have been to simulate the scenario where the processes gets build by the developers but with the help of a specialized consultant. But since neither of us coaches are specialized in tools necessary and we can't afford hire real consultants we chose to not go down that path.

3.4 Collecting data

Our most important questions regarded the viability of implementing our own CoDe pipeline and whether the team preferred doing it or not. We base our result on a few factors:

- How far we came along the steps in section 3.1.
- How many and which stories the team completed.
- What the team members thought about their progress and CoDe.

For the last point we let each team member fill a form after the fifth lab session where they would rate how much they agreed with the following statements:

- We had a clean repository with working tests.
- We had a quick and smooth release process.
- Focusing on CoDe helped the team improve.
- CoDe was worth the time spent on it.
- I would rather have been given a working solution from the beginning.
- I understand the concept of CoDe.

For each statement there would be a scale from 0 to 5. The same questions were given to the teams with a provided set-up, with a few modifications. The point "CoDe was worth the time spent on it." was changed into "Code would have been worth it even if we had to do it ourselves." and the point "I would rather have been given a working solution from the beginning." was changed into "I would rather have gotten to set up the pipeline myself.". During the last planning session we asked the students if they had any other opinions on the matter and discussed how they perceived the benefits of CoDe.

4 Results

In section 4.1 we present how well each of the teams performed. In section 4.2 we summarize the results of our survey. Finally in section 4.3 we mention thoughts the teams had on the process.

4.1 CoDe results

The two teams performed very differently when it came to CoDe. Team 06 quickly got quite far along the steps in section 3.1.

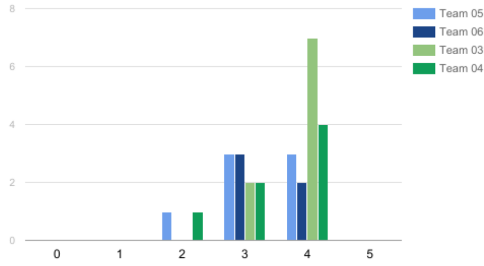
Team 05 on the other hand struggled to get CI to work, even though they spent a lot of time on it. It was not until the fifth week that they managed to set up an automatic integration pipeline that would run all tests with each commit. Note that commits would not be stopped if they failed the tests but the teams would be able to see what and when it went wrong. The team never got build scripts to work and though they had automatic acceptance tests, they were run as unit tests within the IDE and not on built versions of the program.

Team 06 managed to set up the the CI server half way through development. They also made their own script that tested and built the project. There was some trouble getting gui tests to run on the CI server. As a result the team shifted more to shell-scripts that tested and deployed.

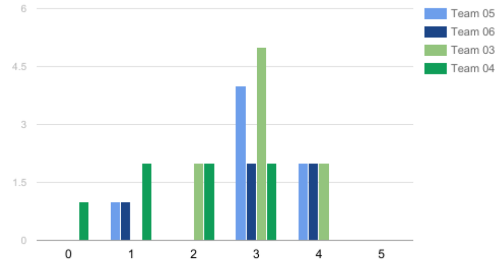
4.2 Survey answers

When applicable, we have placed all four teams in the same diagram, otherwise they are split up in team 05 and 06 (our teams) and team 03 and 04 (teams provided with CoDe tools). Team 05 has the color light blue while team 06 are dark blue. Team 03 are light green and team 04, dark green.

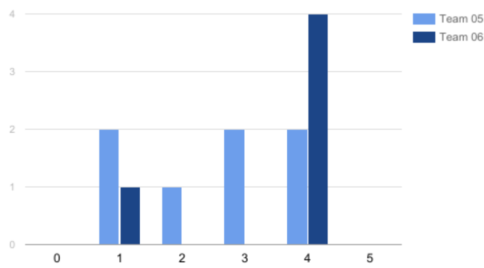
We had a clean repository with working tests.



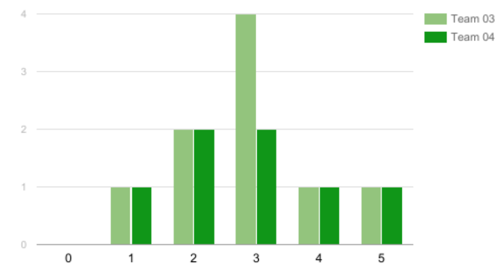
We had a quick and smooth release process.



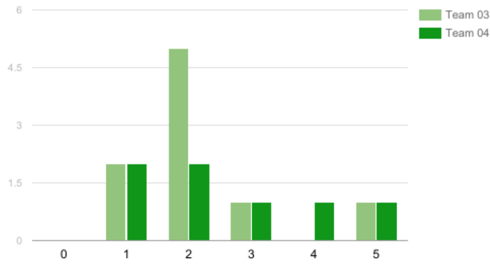
CoDe was worth the time spent on it.



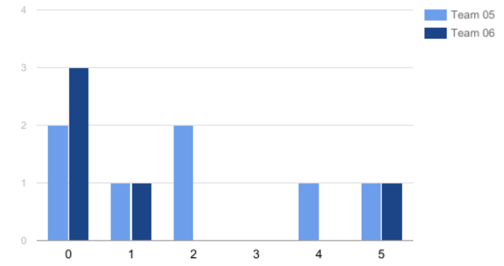
Code would have been worth it even if we had to do it ourselves.



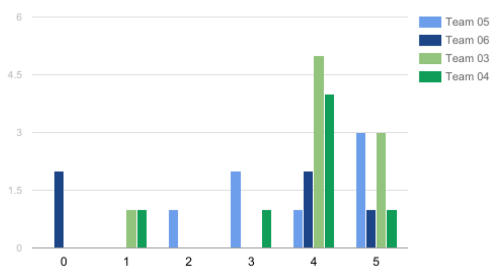
I would rather have gotten to set up the pipeline myself.



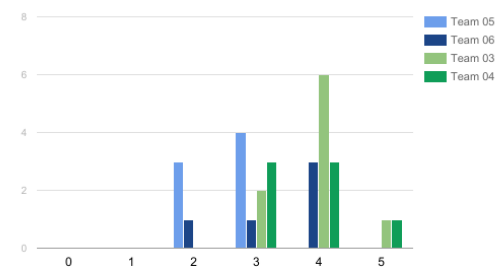
I would rather have been given a working solution from the beginning.



I understand the concept of CoDe.



Focusing on CoDe helped the team improve.



4.3 *Other thoughts*

Team 05 liked CoDe and appreciated the CI server, though they missed not being able to make it build their program and running acceptance tests on it. They agreed that experiencing a release on their own was positive but would have liked if they could have gotten most of the CoDe tools up by the next lab session.

Team 06 managed to set up the CI server for the third lab which was a big success. But when it came to getting acceptance tests working it was harder. In general the team felt positively about feedback from how the build went. Many times builds failed because the team forgot to run all the tests after an auto-merge occurred in git, errors like that got noticed quickly when the CI server responded.

When team 06 for the last iteration managed to get all a script that first ran the unit tests then built and exported executables and then ran all acceptance tests on those executables the atmosphere in the team changed. Seeing all those green tests gave a moral boost, and the mix of unit and black box tests made the team more confident than ever. For the last release we had code pushed within the same five minutes as the release was made without any concern.

5 Discussion

From the above results a few things stand out. First of all, the teams had very varying success with CoDe. The reason for this may come from their different experiences from before the course, we know that team 06 had one very experienced programmer for example. Both teams were very eager though and wanted to learn.

It could be worth considering to what extent the coaches should help the teams. In this study we were both very lenient and open with how the teams should solve their problems. We told them what they needed to look into (for example, delegated a spike on build scripts) but let them choose and set up their tools on their own without interfering much. We believe this helped with their understanding but it also took more time. As a result, they had little time to actually use the systems once they were fully implemented and worked properly. Though team 05 suffered hardest from this, team 06, who got more things working early, also did not have everything done until the last few iterations. As team 05 noted, it might be good to focus more on CoDe early on during the first two weeks though it is possible to wait with using a CI server and build scripts on the working branch until after the first release to still get the benefits from experiencing that.

On the other hand, one could also question the value of not using CoDe for the first weeks as it is still possible that the team's general inexperience would be enough to still ensure a hectic first release. It would be hard to tell the effects of CoDe in the first weeks because the teams improves quickly at first which would add ambiguity to the results.

We see in the survey answers that the teams that were given a CI server from the start had a cleaner repository, which is not very surprising. We were surprised that they did not have as good a release process and that they felt they had a better understanding of CoDe than our teams had. Perhaps the latter is because their knowledge was more evenly distributed than in our teams where the different parts of CoDe were introduced as spikes, meaning that those not doing the spikes in question felt that they lacked knowledge in those areas. We still believe that doing something yourself grants better understanding than having someone else do it for you so if everyone in the teams had partaken in all spikes they should still have a better understanding. Because that is not possible, this would then require better communication between students after each spike so that everyone had a clear idea of what has been done and how it worked. Another reason for the results could be that our teams did not rely on CoDe to the same extent as the other teams as our processes were unfinished.

Something very interesting is that both team 05 and team 06 appreciated getting to implement CoDe themselves. This holds especially true for team 06, likely because they succeeded to a greater extent. For the other teams the feelings were more mixed, though most of them did not agree with the idea of setting up CoDe on their own, contrary to what our teams thought. This could be explained as a psychological thing where the teams preferred the way they worked over other ways. It is common to think your way is better. This is of course just speculation on our part and any conclusions are hard to extract from it.

6 Conclusion

The teams appreciated getting to implement CoDe themselves though it is questionable if this helps with the understanding of CoDe compared to a team that is provided with a working solution. To improve this, the coaches should make sure the teams can have the bases of CoDe set up by the third lab session so that there is ample time for them to use their new tools.

CoDe is tough and requires knowledge to set up. If this knowledge does not already exist within the team, the coaches should consider helping them. If, in the future, CoDe becomes a more prominent part of the PVG course this would be less of an issue and would increase the viability of an iteratively introduced CoDe process.

7 FUTURE WORKS

We said that we did not want to use CoDe for the first two weeks in order to add contrast and give understanding to the teams why CoDe is good. For a future study it could be interesting to see if there is a difference between introducing CoDe at the start of a project or part way through it. Another way to look at this would be with a larger contrast. What would happen the team was given a working CoDe solution halfway through the project?

Another area that needs more research are the impact of automated acceptance tests. Many teams make complicated unit tests as a substitute but how does runnable acceptance tests as black box tests on the executional impact the development.

REFERENCES

[1] ”Extreme Programing Explained”, K. Beck, C. Andres, second edition, 2005, Pearson Education

[2] “Programvaruutveckling i grupp”, http://cs.lth.se/edaf45/

[3] ”Continuous Delivery - Reliable Software Releases through Build, Test, and Deployment Automation”, J. Humble, D. Farley, 2011, Pearson Education

[4] ”Continuous Integration”, M. Fowler, May, 2006 url: https://martinfowler.com/articles/continuousIntegration.html, fetched 05-03-2017

[5] ”Continuous Delivery”, M. Fowler, August, 2014 url: https://martinfowler.com/bliki/ContinuousDelivery.html, fetched 26-02-2017

[6] ”Coaching an inexperienced agile team towards a continuous delivery methodology”, E Gärtner and M. Eriksson, 2015

[7] ”Continuous Integration with Jenkins”, J. Hembrink and P-G. Stenberg, 2013,