Final thesis

Continuous Integration – A comparison between theory and practice

by

Martin Sandberg

LIU-IDA/LITH-EX-G--15/005--SE

2015-02-27
Final thesis

Continuous Integration - A comparison between theory and practice

by

Martin Sandberg

LIU-IDA/LITH-EX-G–15/005–SE

2015-02-27

Supervisor: Johan Glännfjord
Examiner: Kristian Sandahl
Abstract

To develop software in a larger scale, some kind of software development method is needed to organize the development. Many different software development methods have emerged since the beginning of software development, and Continuous Integration (CI) is one of them. Many companies have applied, or tried to apply CI in their business. Some companies were successful, some were not. This thesis examines what CI is in the theory, and compare it to how it works in the practice in the department "Product Development Traffic Control" (PD TC) which is a part of the "Product Development Unit LTE Multistandard Radio" (PDU LMR) within Ericsson. The theory of CI is examined mostly through literature studies. CI in practice was examined through interviews with developers and employees working with the CI-machinery at PD TC. The comparison between the theory and the studied company revealed that it is difficult to adapt CI perfectly to a large organization. This does not imply that CI is implemented in the wrong way, but the benefits of implementing CI in large projects may not be as enormous in comparison with smaller projects.
## Contents

1 Introduction  
1.1 Motivation ........................................... 1  
1.2 Purpose .............................................. 1  
1.3 Method ................................................ 1  
1.4 Questions ............................................ 2  
1.5 Limitations .......................................... 2  
1.6 Prerequisite ......................................... 2  

2 What is CI?  
2.1 The basics with CI .................................. 4  
2.2 Example of a CI-system .............................. 5  
2.3 Advantages .......................................... 6  
2.3.1 Reduce Risk ....................................... 6  
2.3.2 Less work ......................................... 6  
2.3.3 Deployable software at any time ............... 6  
2.4 Disadvantages ....................................... 8  
2.4.1 Takes time to start up and to maintain ....... 8  
2.4.2 Extra expenses ................................... 8  

3 What is needed?  
3.1 Version Control System ............................ 9  
3.2 Developers .......................................... 10  
3.3 CI-server ............................................. 11  
3.4 Build script .......................................... 11  
3.4.1 Code-inspection .................................. 11  
3.4.2 Compiling ......................................... 11  
3.4.3 Database integration ............................ 11  
3.4.4 Testing ............................................ 12  
3.4.5 Deployment ....................................... 12  
3.5 Feedback mechanism ............................... 12
CONTENTS

4 PDTC's CI-system .................................................. 13
  4.1 VCS .................................................................. 13
    4.1.1 Rebase .......................................................... 14
  4.2 Developers .......................................................... 15
  4.3 CI-server .............................................................. 15
  4.4 Build-script ......................................................... 15
    4.4.1 Code-Inspection .............................................. 16
    4.4.2 Integration ..................................................... 16
    4.4.3 Testing .......................................................... 16
  4.5 Feedback mechanism ............................................. 17

5 PDTC's CI workflow .................................................. 19
  5.1 Phase 1 .............................................................. 20
    5.1.1 Review .......................................................... 20
    5.1.2 Verification ..................................................... 21
  5.2 Phase 2 .............................................................. 22
    5.2.1 Mainline Delivery Queue ................................... 22
    5.2.2 Builds and tests .............................................. 22
    5.2.3 SDC ............................................................... 23
  5.3 The end? ............................................................ 23

6 Experiences among the developers ................................. 24

7 Analysis .................................................................... 26

8 Improvements .......................................................... 28
  8.1 No daily basis ....................................................... 28
  8.2 More automation .................................................. 29
  8.3 Feedback ............................................................ 29

9 Discussion ............................................................... 30
  9.1 Ethic & environment .............................................. 31

10 Dictionary ................................................................ 32

11 Interviews .............................................................. 34
  11.1 Interview 1 ........................................................ 34
  11.2 Interview 2 ........................................................ 35
  11.3 Interview 3 ........................................................ 35
  11.4 Interview 4 ........................................................ 35
Chapter 1

Introduction

1.1 Motivation

Continuous Integration (CI) has become a very important concept in development of the LTE system at Ericsson. Implementation of CI-mechanisms has mainly been carried out in order to improve the quality of the product, but also to increase efficiency of the development and reduce the delivery time to the market. CI-mechanisms are crucial to the organization, and to be able to gain full potential of the mechanisms and to meet future demands it is fundamental to improve the understanding of the mechanisms.

1.2 Purpose

The purpose with this thesis is to study how Continuous Integration is supposed to work according to the theory, and investigate how it works in the practice at PD TC. These differences will be analyzed to evaluate whether the current way of working is the proper way, or if there are any possible improvements to perform.

1.3 Method

The structure of the work behind this thesis could be described with following phases: First, literature studies within CI was performed to get a basic understanding of what CI is. After this phase, the examination of CI in PD TC was performed. It was achieved through workshops, discussions and interviews with developers as well as employees working with the CI-machinery. I had very good contact with a cross functional team (XFT), that the thesis is mainly based on. I reconnected to the theory continuously while examining the CI in practice to get a deeper understanding of the differences between practice and theory.
1.4 Questions

- What is Continuous Integration according to the theory?
- How does Continuous Integration work at PD TC?
- What are the differences between PD TC’s adaption of CI and the theory?
- Are there any possible improvements that could be considered at PD TC?

1.5 Limitations

Following limitations have been decided due to the limited scope of this thesis:

- There will be no investigation of how Continuous Integration is implemented in other departments within Ericsson, or companies outside Ericsson.
- When investigating the CI-machinery, focus will be on the part of LTE’s CI-machinery that PD TC is using.
- There will be no implementation of eventual improvements found.
- I have chosen not to include deeper theories or ideas about CI, but the overall description of what CI is.

1.6 Prerequisite

This thesis is written with following prerequisites:

- Basic knowledge in computer engineering.
- Basic knowledge of what a version control system is.
Chapter 2

What is CI?

CI stands for Continuous Integration, and was initially taken from the Daily-build method by Microsoft in the 90’s [1]. It was later included in the software development method eXtreme programming (XP), in the end of the 90’s as one of it’s original methods [2]. The name eXtreme programming comes from the idea of taking all "best-practices" to an extreme level [3]. The goal with CI was to prevent the occurrence of major problems in the course of the software integration, through integrating more frequently [3]. In a software development project where several developers are involved, the integration process really is something that can contribute with major problems. Integration is a very unpredictable step in the development process, and can easily be more time consuming than the actual programming process [3, p. 49].

Due to this, it is significant to test the whole system frequently already from an early stage. But often this is not done. Instead, each individual part is developed until it is finished before testing of the whole system takes place [4].

Here is where CI comes in. CI is basically a concept to easily integrate all code that is written, and also compile and test it more frequently. And this should be done more or less automatically. Through this approach, developers should submit the code to the CI-server on daily basis. The CI-server integrate and test the code automatically. It doesn’t imply that this process solves all the issues, but when integration problems occur, developers find out this already a short while after a faulty line of code is written. Thus, CI provides much faster feedback than if this should be done manually in the end of the development process [4].
2.1 The basics with CI

The basics with CI could be described in short as [4]:

- **To integrate more often.** The faster the code integrates, the faster the developer finds out if it works with the rest of the code implemented by other developers. With integration means that the different components of the code shall be put together, think of it like a puzzle.

- **To improve efficiency by testing all the code more frequently.** Through this, the search area gets smaller once problems arise.

- **Reduce the manual work** by automating as much as possible in terms of compiling, testing, database integration and more.

- **Be able to produce functional, executable software at any time** that has already been tested and is ready to be delivered.
2.2 What could a CI system look like?

Below follows one example of how a CI system could look like. It is possible to adapt CI in different ways, depending on the different needs, but this is the fundamental idea [4].

1. A developer submits his or her code to the repository in the version control system. It is of great importance to write new or change old test cases for the added or changed functionality. These test cases shall be submitted together with the updated code to the repository.

2. After code submit, the CI-server discovers that a change has occurred, and fetches the new code from the version control repository. After this the CI-server executes a build script that contains compiling, database integration and the execution of the updated test cases. What is included in this step could vary, but according to the theory, this is the fundamentals.

3. When the build\(^1\) is finished, the CI-server sends direct feedback to the involved project members.

---

\(^{1}\)What the definition of a build is could vary, but in this thesis a build includes not only compiling and integration, but also testing

![Figure 2.1: Illustration of a CI-system [4, p. 40]](image-url)
2.3 Advantages

2.3.1 Reduce Risk

The fundamental idea with CI is that every developer shall submit their code at least on daily basis. This leads to many integrations every day. At every integration, several tests should be executed to ensure that everything looks ok. With the help from the CI-server’s feedback mechanism it enables the developers to get rapid feedback on their code, which results in [4]:

- The developer gets to know about the faults that get introduced in the code already when the code is submitted to the version control repository, and not in a later phase of the project [4, p 24].
- It becomes easier to get an overview on the status the software has in real-time [4, p 31].
- If the code is tested with the same process against the same scripts all the time, there will be less assumptions about the software quality and/or the reliability of third party modules [4, p 23-24].
- The feeling about the quality of the software gets better. This is because it is possible to at any time see what a certain version of the software has been tested against, and if it has passed or failed these tests. Without CI-mechanisms it becomes more difficult to say to what extent a new piece of code will affect the rest of the software [4, p 31-32].

2.3.2 Less work

It becomes much easier when having a CI-server that takes care about much work that usually has to be done manually. It can be tasks as compiling, database integration, testing, code inspection and feedback on the quality of the code. In this way the project members get more time to do other important work. It also becomes less risky to miss some step in the process because everything is run automatically at every code submit [4].

2.3.3 Deployable software at any time

Due to the early and continuous integration, testing and deployment, it is possible to deliver the software as a “whole product” to an end-user or customer at any time. The product may not contain all functionality in the early part of the project, but it has at least been tested and deployed. This makes it a lot easier to ship the product to the customer. If not using CI, and integrating occurs for the first time in a later part of the project, it is likely to encounter unexpected problems. Suddenly, when integrating, intractable issues emerge, that puts the deadline at stake due to unpredictable time...
2.3. ADVANTAGES

CHAPTER 2. WHAT IS CI?

consumption to solve the issues. In worst case, this could lead to loosing a customer. This is not preferable for any project [4].
2.4. Disadvantages

2.4.1 Takes time to start up and to maintain

To be able to obtain a fully functional CI-machinery time to set it all up is needed. If there is a need to change the way of working and a high number of processes to obtain CI this can be quite time consuming. There is also some time needed to maintain CI [4]. In very small projects this might not be suitable.

2.4.2 Extra expenses

To preform CI in a good way, separate hardware to run the integration server on is needed. This can be seen as an expensive investment [4].
Chapter 3

What is needed to implement CI?

3.1 Version Control System

First of all, a version control system (illustrated in figure 3.1) is needed. A version control system manages a central or distributed storage space, where the developers fetch and submit their code to [4][7]. This space is called a repository [4]. First, the developer fetches the code from the repository. Then the developer implement his or her changes to the code. After this, when the developer is finished, he or she submit the updated code back to the repository which makes it possible to other developers to fetch the new changes. The version control system record all changes to the files it manages in the repository [5]. It comes with possibilities such as to [5]:

- Compare a file with an older version of that file to see what changes that has been made.
- See who made the changes that might introduced a problem to the code
- Revert one or several files to older versions

There are many different types of version control systems on the market, but Paul M. Duvall, the author of Continuous Integration - Improving Software Quality and Reducing Risk advocates the use of Subversion in his book, because it is freely available, and it has a set of useful features [4, p. 8].
3.2 Developers

As a developer, it is very important to assure that the code is working as intended before submitting it to the repository. This is to prevent to break the code in the repository. Therefore, all developers should compile, test and integrate their code on their local workstation before submitting. The tests should be implemented by the developers themselves. It is enough to only execute unit tests\(^1\) of the implemented code before submit. In that way, there is less risk that the code in the repository is going to break. After everything works on the local workstation, the developer submits the code to the repository. It is important to not forget to also submit the newly implemented test cases as well. Further on, a rule of thumb is that each developer should submit at least on daily basis. In this way there will be much easier to troubleshoot when something goes wrong, because the search area gets smaller \[4\].

\(^1\)Unit test = Test of the smallest testable piece in the code, e.g. a class
3.3 CI-server

An integration server, preferred dedicated, is needed to host a CI-server. The CI-server’s mission is to with short intervals pull the version control repository for changes. If a change has occurred, the CI-server fetch the new or updated code and tests and executes a build script [4].

3.4 Build script

The build script is a script that normally executes following tasks:

- Code-inspection
- Compiling
- Database integration
- Tests
- Deployment of the software

The whole process executed by the build script is called a build (yes, a build could be more than just a compile) [4, p. 4]. What is included in the build script can differ, but above tasks is the fundamentals. It is very important that the build script does not take too long time to execute. A recommendation according to the theory of CI is top 10 minutes. The reason for this is that only one developer can build his or her code at a time. If the build script takes too long time, it becomes a bottleneck. The tasks executed by the build script is described in the following text below.

3.4.1 Code-inspection

It is important to have some kind of code inspection in the build script. Maybe the project have decided different rules or criteria that the source code should follow? For instance, automated checks that guarantee these rules and criteria are fulfilled could be included[4].

3.4.2 Compiling

Is one of the basic features of CI. It means that the CI-server should automatically compile the source code submitted to the version control repository, and make an executable file [4].

3.4.3 Database integration

Some people might think that database integration and code integration are two different things. It is, but why not automate that step to? The database integration part can be a simple script where all the data tables are dropped, and created again (but with new data) [4].
3.5. FEEDBACK MECHANISM

CHAPTER 3. WHAT IS NEEDED?

3.4.4 Testing

Maybe the most important part of the build script is the testing part [4]. The developers should (as mentioned in section 3.2) implement and execute unit tests before submit. But hopefully there are a lot more tests that exists together with the rest of the code. If all these tests should be executed this can easily make the build script take more than the suggested 10 minutes [4, p 141]. Therefore it is very important to have some strategy to somehow categorize the different test cases. A good idea to have in mind to speed up the testing is to start small. For instance, in the build script that triggered by every code submit, execute only unit tests. Then a second build script could be implemented, that executes periodically. For instance, once every night. This build script could contain the more time consuming tests like component tests, system tests and functional tests [4, p. 138].

3.4.5 Deployment

Finally we have come to the last part in the build script. After the above steps have passed, it is time to deploy the software. Deployment is the step when the software is executed and tested in an environment as "customer-like" as possible. This can be considered as the last step in the testing part.

3.5 Feedback mechanism

When the build script has been executed, a result shall be sent to the developer that submitted the code. This can be done through RSS, e-mail or text message to name a few examples. The build script has to pass all the different tasks to pass. It is enough with just one failed task, and the result is set to "fail". When a task in the build script is failed, it is of highest priority to fix this. It is the developer who submitted the faulty code that has the responsibility for this [4].
Chapter 4

PDTC’s CI-system

4.1 Version control system

PD TC is using the version control system Git. Git was created 2005 to manage the source code for the Linux core. Git is known as a rapid and distributed version control system, and probably the best thing with Git compared to other version control systems is its branching model [7].

Branching could basically be described as making a copy of the existing code in mainline, and start a “new project” with it. This is not what actually happens, but it is easier to describe it that way. The mainline is the “code foundation” which should be fully tested and fully working. New code is written upon this code (often in form of a branch) and later on merges back into the mainline to become the new mainline. It is widely used within PD TC, as a way to enable many parallel activities at the same time without messing with the mainline. For example each new feature that is being developed is normally started as a new branch. Figure 4.1 illustrates in a very brief way how branching works from creation of the branch to the integration back into mainline. Commits to the mainline shown in figure 4.1 could be commits from other branches or commits made directly on the mainline.

![Illustration of a branch.](image_url)
4.1. VCS

4.1.1 Rebase

Before integrating the code back to mainline, a rebase is needed to get the latest changes from mainline. A rebase is a command in Git and basically changes the base\(^1\) of a branch, to integrate recent changes performed on mainline into the branch. In this case, eventual integration problems occurs outside the mainline which is very good.

![Diagram of branch merged back into mainline](image)

Figure 4.2: Illustration of a branch merged back into mainline (same as figure 4.1). The branch is based on mainline commit "MC1".

![Diagram of branch rebased](image)

Figure 4.3: Instead of basing the branch on mainline commit "MC1", it is now rebased and based on the latest mainline commit "MC3". What happens here is basically that you add all the code implemented in the branch upon the newer mainline commit "MC3" instead.

![Diagram of branch rebased back to mainline](image)

Figure 4.4: The branch now have changed base to "MC3", and when integrating back to mainline it becomes much easier since there are no more changes performed on mainline since last rebase.

---

\(^1\)A base is the commit in the mainline where the branch was started or previously rebased from.
4.2 Developers

As a developer in a cross functional team at PD TC the responsibilities are both to implement code and test cases (described more in detail in section 4.4). Instead of executing these tests locally before submitting the code to the mainline as described in theory, these tests executes on a dedicated server. At PD TC there are no requirements how frequently the code shall be submitted to the CI-server, this responsibility is on the team. Some teams submit their work more often than others, and usually the integration problems increase at less frequent deliveries.

4.3 CI-server

What is meant to be a CI-server in the theory I rather prefer to call ”CI-machinery” in the practice. The reason for this is mainly because the expression “CI-server” is not very correct, since it implies the use of only one machine, one CI-server. In the parts of LTE’s CI-machinery PD TC is using, there are actually several machines taking care of the functionality of the CI-server. However, when talking about the CI-server or CI-machinery we refer to the actual software used in the server(s).

The first part of the CI-machinery is Gerrit. It is to Gerrit the developers submit their code, and also where the CI-machinery begins. Gerrit is basically a web based code review system where the developers review each others code [10]. It is closely integrated with Jenkins to easily get an automated process chain. Jenkins is a tool that at its website is described as ”an extendable open source continuous integration server” [11]. It is designed to automatically build, integrate and test software [11]. When new code has been submitted to Gerrit, Gerrit communicates with Jenkins to perform the integration and the testing.

In the theory the code is first put in a version control repository and then pulled to the CI-server. At PD TC the code first arrives to the ”CI-server”, and if the build passes, it is sent to the repository. Gerrit could be described as a gate keeper to the repository, protecting it from broken builds.

4.4 Build-script

What is called ”build script” in the theory we can say is composed of the following at PD TC:

- Code-inspection
- Integration
- Tests
4.4. BUILD-SCRIPT

4.4.1 Code-Inspection

The code inspection part at PD TC can be done either through pair programming, or through a code review in Gerrit (explained in section 4.3). When programming in pairs, the developers continuously review each other's code. When using code review in Gerrit, the code first has to be reviewed and approved by the developer who submitted the code, and normally by one of his or her team members too.

4.4.2 Integration

As mentioned in section 2.1, integration is the "puzzling part" in the CI process. First, every piece must work for themselves. Then every piece must work together with their neighbours, and finally with all pieces together. The integration process at PD TC occurs in different levels. To be able to understand this integration process I have to explain the different parts of the code.

First, at the lowest level there are products. A product can either be a software unit, an interface unit or a library unit. An interface unit is a unit where different protocols and interfaces used are defined. A library unit can be explained as a regular C++ library that contains a collection of common classes and functions. Software units contains the implementation of functionality.

At the next level, several of these products integrates to a Load Module (LM). This is how deep this thesis will go, but I will briefly explain the next steps anyway. After the LM's are built there are two separate ways in the CI-machinery depending on the technology developed:

- Technology 1: The LM's gets integrated directly to an Upgrade Package (UP)\(^2\)
- Technology 2: The LM's gets integrated to Load Module Container (LMC), and then integrates with other LMC's to later form a UP.

4.4.3 Testing

There are different tests executed in the CI-machinery. Below I will explain which different tests there are, and what these tests are verifying.

4.4.3.1 A-test

A-test is actually a test framework, but when executing the collection of the following tests below, it is mentioned as executing one A-test.

- **Basic test**: Is a kind of unit test, and is testing only one product independent of any other product. It is the developers responsibility to write these test cases.

\(^2\)Upgrade Package = The final software used in a base station [6]
4.5. FEEDBACK MECHANISM  CHAPTER 4. PDTC’S CI-SYSTEM

- **FS verify**: A file structure verification test that checks that the file structure applies to some given rules.

- **Pure Coverage**: Checks whether the basic tests are "visiting" every line of code. This is a test to verify that new test cases has been written for the submitted code.

- **Purify**: Check for memory leaks in the code

- **Model validation**: Checks that the models\(^3\) that have been created or changed follow a specific set of rules.

- **Lint**: Is a static code analysis test that checks the code for usual mistakes that developers tend to introduce in the code. For instance, if a developer forgets to check if a pointer is null before using it, this test will fail.

### 4.4.3.2 Multi component test

A multi component test is as it sounds testing different components together (in this case when talking about components, we are talking about products). It requires some additional experience and understanding of the environment around the product in focus to implement this kind of tests. This is due to the fact that more than one product is involved in the testing, and you have to understand the dependencies between them. The multi component tests are not necessarily executed in the CI-machinery upon code submit for the current products. However, they have to be executed before the feature is considered as fully implemented.

### 4.4.3.3 Node test

Node test means that tests in an environment with real nodes\(^4\) are executed. Tests like setting up a call, attaching cell phones to a base station are performed in a node test.

## 4.5 Feedback mechanism

The feedback is a very important key factor in the theory of CI. At PD TC the feedback is sent through an e-mail to the developer submitting the code. It is also possible to follow the progress of the build in Gerrit. If the developer want more detailed information about the load module build or the A-test(s) there is a link to the Jenkins build page for the current build. The developers get feedback on following levels, regardless of the result:

- The code review in Gerrit

---

\(^3\)Model = A textual or visual description of a part in a software

\(^4\)Node = A real node in the mobile network. For example a base station
4.5. FEEDBACK MECHANISM  CHAPTER 4. PDTC’S CI-SYSTEM

- The load module build executed by Jenkins
- The A-tests executed by Jenkins.
- If the submitted code reaches the mainline or not
Chapter 5

PDTC’s CI workflow

Figure 5.1: Illustration of PD TC’s part of LMR’s CI-machinery.
5.1. PHASE 1

The CI-workflow at PD TC is divided into two phases: "Pre-integration" and "Mainline integration".

5.1 Phase 1: Pre-integration

Figure 5.2: The pre-integration phase.

The pre-integration phase contains following parts:

- Review
- Builds and tests

5.1.1 Review

When it is time for the developers to submit the code, it is submitted to Gerrit. Gerrit sends a job request to Jenkins. Also, an e-mail is sent to the stake holders involved with the submitted code that a code review needs to be done. The submitted code is not stored in a visible repository where developers can fetch the code from, just a temporarily repository. Gerrit has a grading system where the developers reviewing the code can put a grade on the submitted code. To pass this stage the submitted code needs at least to be approved by two reviewers. If a reviewer reject the submitted code for some reason, the pre-integration phase will fail directly and the submitting developer is notified by e-mail.
5.1.2 Builds and tests

In parallel with the code review in Gerrit, Jenkins receives the job request from Gerrit and starts one or several A-tests and Load Module Builds (described previously in 4.4.3).

Jenkins dynamically chooses a couple of A-tests to run. This technique is called Multi-dependency A-test, and selects the different A-tests depending on what dependencies there are from the submitted code.

In parallel with the Dependency multi A-test, Jenkins start building of products into load modules (LM-builds). If Jenkins fails with either the tests or the LM-builds, the pre-integration phase fails and feedback is sent via e-mail to the developer submitting the code. If both tests and LM-builds passes in Jenkins together with the code review in Gerrit, the delivery is qualified for phase two.
5.2 Phase 2: Mainline integration

When the pre-integration phase is finished, it is time for the second phase, which contains:

- Mainline Delivery Queue
- A-tests and LM-builds
- Source Delivery Check

5.2.1 Mainline Delivery Queue

Before entering the mainline there is a queue-system. Instead of forcing the developers to wait until there is a free time slot before submitting, the developers submit directly. The submitted code is then put in the queue until the resources needed are free.

5.2.2 Builds and tests

In the mainline integration phase, there is a need to build and test the submitted code again. This is because the mainline the pre-integration was based on for the different deliveries might have changed. This is performed by doing a rebase (explained in section 4.1.1, and submit a job request to Jenkins that performs the Dependency multi A-test and the LM-build again. If it fails, we go back to first phase again. If the build is approved, it is submitted to the "Source Delivery Check".
5.3. THE END?  

CHAPTER 5. PDTC’S CI WORKFLOW

5.2.3 Source Delivery Check

In this step, the system executes multi component tests with real nodes involved. If the delivery pass this stage, it is submitted to the mainline. If it fails, the build is discarded. In either way feedback is sent to the involved developers.

5.3 The end?

No, the CI-machinery does not end here. After the delivery has made it to the mainline, the load modules are integrated to an Upgrade Package (UP), and additional testing occurs. Here the more time consuming testing is executed several times a day. But this is beyond the scope of this thesis.
Chapter 6

Experiences among the developers

To be able to get the employees perspective on what CI is and what it has done to their work I set up several interviews. There were a total of seven formal interviews. The interviews were held with developers from an XFT, working mostly with feature development, and also employees working with the CI-machinery. In addition to these interviews, there were also several informal interviews, workshops and coffee-break-chats that also took place. All this material was gathered and documented, and in this chapter I will summarize the experiences of Continuous Integration presented in this material.

Maybe the most important aspect when implementing a new software development method like CI into a software developing company is how the developers working with the code experience the changes. Based on the interviews held, I think the knowledge about what CI is, is rather good. All the participants in the interviews described CI as a technique to integrate and test more frequently, which is comparable to the definition of CI. But how big change has it been since the introduction of CI? Are there any major differences experienced by the developers that deliver the actual code? Based on the interviews done with the XFT at PD TC, they all can agree that they notice some improvement, but not a huge one. Compare this to the examples in the book, where the introduction of CI has become more or less like a religion for some. Why do the developers not think it have been a huge improvement then? In the interviews I asked how the process looked like before CI. The answer I got from the developers that had worked since the "transformation into CI" was that the process almost looked the same. But there was actually one big difference: Before CI, the developers had to execute the A-tests, MCT’s and other tests manually and locally on their own computer. The first disadvantage with this approach is the different user environments that exist. Secondly, and even more crucial - If this was
forgotten, unverified code slipped through, and the time taken before anyone noticed an eventual fault was longer. After CI was introduced, these tests are now executed automatically at code submit. This might seem as a small change to the developers, but I think the change is vital to the organization. Every step done manually could be a risk. If it is possible to automate, this could save time, reduce assumptions and so on. If it is possible to proceed to next step without testing, this WILL happen - humans make mistakes. Another reason to why the changes made didn’t seem to affect the developers that much could be that the changes are not that visible to the developers writing the code. Instead the actual change by implementing CI was the machinery “behind the fence” from the developers perspective.
Chapter 7

Analysis

In the theory of CI the most important task is to “fix the broken build”. This means, when integrating with the mainline and the code stops working - the highest priority is to fix this. Meanwhile, no deliveries shall be done, and no one should fetch code from the repository with the broken build. This approach should not be feasible within PD TC, where hundreds of developers are involved. You can’t stop deliveries from hundreds of developers when someone breaks the build. That would be very expensive. Instead, PD TC use branches to separate different feature developing teams from each other. If one developer submits broken code to the team’s feature branch - only the team (or the people involved in the feature) will get affected. Before even having the possibility to deliver to the mainline the code must go through a safety net that consists of various testing to ensure the delivery doesn’t break the code. This is a very good idea compared to the theory where it’s the developers responsibility to test the code before delivery.

The theory of CI also advocates that all code shall be integrated to the mainline at least once a day. This works perfectly fine in smaller projects. But when the number of developers increases it becomes a problem. Imagine how it would look like if these hundreds of developers in PD TC should deliver to the same mainline every day. I would not say it is going to be impossible, but it is very difficult. You cannot integrate hundreds of code pieces into one piece without any problems. First of all, if implemented perfectly according to the theory, integration should not take more than 10 minutes. This means a total of 16 hours integration time for a hundred developers. Then we do not include any broken builds, everything has to work perfectly. We clearly see that the equation does not add up - it is not possible to integrate every piece of code individually.

What PD TC has done in order to adapt CI to fit their process better is mainly to use branches. Every feature or task are mainly developed in a branch. The team responsible for the branch decides when they do rebases, when they deliver, and what to deliver to the mainline. They also decide
CHAPTER 7. ANALYSIS

when the individual developers should deliver to the branch. Some teams deliver often, and solve integration problems continuously. Other teams can stay outside mainline on a branch for months without doing any rebases before integrating to the mainline. There are no doubts about that there will be more integration problems for the later example. It does not mean that all teams that are "on branch" for a long time are not aware of the risks. Sometimes when major changes are to be done it might not be possible to do rebases or integrate with mainline before this change is fully performed.

The branching technique is a good approach, because it solves the problem with impossible high number of deliveries into mainline. Instead of integrating hundreds of deliveries from individual developers, we are now talking about tens of branches. But still, we have the fact that daily deliveries into mainline from each branch are rare.

One consideration regarding the differences between the theory and PD TC’s implementation of CI is the integration time. According to the theory an integration should not take more than 10 minutes. When delivering code, the first integration with the mainline takes significantly more than that. If this integration succeeds the delivery is eventually put in a queue to the mainline. If that is the case, a new integration with the mainline has to take place, which also usually takes more time than the recommended 10 minutes. The time taken in this step depends on the number of other deliveries in front in the queue that is integrated with the mainline and could affect the current delivery. This means in worst case scenario, the integration process to mainline could be quite far away from the 10-minute build. On the other hand, this might be acceptable. It is a very large organization with far more than ten developers. And as Daniel Ståhl, an employee at Ericsson mentioned in his research report - "studies that validate the benefits of Continuous Integration in large scale development projects is lacking". However, he also found out that there are proof of some benefits by using CI in large scale development projects, but as the participants in the interviews that were held in this thesis said - "The difference is not huge".
Chapter 8

Possible improvements

When I first started with this thesis, I was sure that there would be a large difference between the practice and the actual theory, with many parts to improve. Later on I realized that the CI-machinery at PD TC actually works very well, and the number of improvements that I consider could be done are few.

8.1 No daily basis

The first thing I experienced as a disadvantage was the low number of integrations made from each developer. According to the theory, every developer should submit code at least on daily basis. All these code pieces should go through the whole CI-machinery and finally the code shall be deployed in a real system\(^1\). I mentioned in the analysis chapter that there are hundreds of developers in PD TC, and it would be difficult to integrate hundreds of pieces of code daily with the mainline. But why not take advantage of the branching technique that PD TC uses? Instead of hundreds of individual developers, we have tens of branches to integrate with the mainline. This would probably work in the theory. But when talking to the developers they say it is almost impossible to submit something to the mainline every day. As one developer said: "Think if a surgeon has to take a break in the middle of a heart transplant to test if that part of the transplant worked". That would probably never work. Exactly like the human body, the system developed within PD TC is a large and complex system. It is difficult to submit what you have done on daily basis. However, one thing that could be done in order to increase the number of deliveries to mainline is to divide the code into smaller components. If this is possible, it is probably an expensive improvement since parts of the structure has to be re-designed.

\(^1\)With a real system it means that the system should be as customer like as possible
8.2 More automation

There are no parts of the CI-machinery I have seen that needs automation, except one part - The code reviews in Gerrit. A lot of code inspections executes automatically, but the review is done manually. Instead of spending expensive time reviewing submitted code, is it possible to automate? If the review process is changed, everything needed is a piece of code instead of teaching all the individual developers the new process. If an automation of the code review seems impossible now, start small and see how much of the code review that could be automated.

8.3 Feedback

The feedback in the CI-machinery is good, but it could get better. E-mail is not the best way of notifying the developers about the feedback. Neither is a website that the developers has to visit to get more information. What could be an improvement is a monitor that in real-time shows all information that the team needs. For instance: Status about the team’s latest build, how big change there is on the mainline since last rebase, how big the mainline queue is and so on. Some information could be gathered in radiators to visualize whether the numbers are considered as good or bad. Another good idea is a sound every time a commit from the team passes all tests and reaches the mainline.
Chapter 9

Discussion

To enter a large company more or less directly from school and investigate a software development method they are using is difficult. I had no experience of using any particular software development method at all before (except waterfall method\footnote{Waterfall method is a software development method where you have strict phases typically in following order: Pre-study, implementation, testing} if that one counts). So I had to confide in the experience among the employees. On the other hand it could be good to let an outside person examine the methods used within a company to see it from another perspective. And I think that it turned out pretty well. I had a very good book \cite{4} as a base while studying the theory. It was easy to understand and easy to read, even for me who is not a good reader. In fact it was so good, that I almost only used that book in the theoretical part. This could of course be a disadvantage compared to reading several different books in the subject. But due to the limited scope of this thesis, in combination with my reading difficulties i decided to mainly go with that book. Through the book I also found interesting persons (like the author and other CI-gurus) that I looked up. Through these people I found more useful information that I used as well. I consider the sources very reliable that I used for the theoretical part. I also tried to bring more focus in the investigation of PD TC’s CI-system, since I knew that could take more time than I planned for. Even if I tried to focus more on the real world investigation, this was actually not enough. I could easily spend all my 10 weeks on this investigation, so we decided to narrow down the scope of the thesis to make it manageable. As I mentioned in the introduction I reconnected to the theory continuously all the time while examining the CI in practice. This was a very instructive way to work, since it enabled fast feedback regarding if the practice at PD TC were close to the theory or not (actually a little bit as one of the advantages with CI).

In the first weeks during the thesis I was seated together with the XFT mentioned in the thesis. This was very good since I quickly could ask ques-
9.1 ETHIC & ENVIRONMENT

CHAPTER 9. DISCUSSION

tions and start discussions. But since a new member entered the team during the time of the thesis, I had to move to another location. It was more difficult to have these spontaneously coffee-breaks where a lot of interesting discussion was ongoing. However, I do not think this affected the result of my thesis since I had got a grip of most parts in the CI-machinery at PD TC by then. Even if I was not seated together with the XFT I had the material from the interviews which were very useful since that was the best source of information I had in the examination part.

Actually, while examining the CI-machinery I found a disadvantage with the theoretical part compared to CI at LTE. From a CI perspective if the code breaks when submitting to the repository, the highest priority is to fix this. This can happen if one or two statements are true: 1. The developer has not tested his or her code, 2. The developer has not tested his or her code good enough. At LTE, instead of submitting broken code to the repository the developer is forced to go through this safety net with tests before the code is automatically sent through in the CI-machinery. I totally agree with LTE’s technique here. If there is any sign of possibility to break the code when submitting, this must be prevented in all possible extent. I don’t think the developers should have the responsibility to test the code themselves; it should be included in the CI-machinery.

9.1 Ethical and environmental aspects

With the use of Continuous Integration, there are a much wider use of automation than before. This result in two disadvantages regarding the ethical and environmental aspects:

- **In the ethical aspect;** when automating, the same number of employees might not be needed. This can lead to reductions among the personnel working with the development because the number of hours spent is less than before. It is very difficult to say if this could happen at LTE, but there might be a small risk. On the other hand, the quality of the tasks performed at the workplace increase since it feels more useful to write a line of code than compile and test it manually from time to time if this can be done automatically.

- **In the environmental aspect;** introducing CI means that more frequent integration and testing should take place. This creates a larger need for new hardware that should take care of this. All use of hardware affects the environment since a lot of energy is needed. However, one of Ericsson’s three core values is ”perseverance”, and they work very hard to minimize the environmental footprint.
## Chapter 10

### Dictionary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMR</td>
<td>LTE Mobile Radio (The department responsible for the development of the LTE technology)</td>
</tr>
<tr>
<td>LTE</td>
<td>Long Term Evolution (a mobile radio communication technology, also known as 4G)</td>
</tr>
<tr>
<td>PDTC</td>
<td>Product Development Traffic Control (A department within LMR)</td>
</tr>
<tr>
<td>XFT</td>
<td>Cross-functional team (A team that shall be able to work with several different tasks)</td>
</tr>
<tr>
<td>XP</td>
<td>eXtreme Programming (A software development method where all good practices from other methods are taken to an extreme level)</td>
</tr>
</tbody>
</table>
Bibliography


[9] Interview with a team member in a cross functional team.


Chapter 11

Interviews

In this chapter I’ve gathered notes from interviews that were held.

11.1 Interview 1

What are you working with? What is your role in the team?
Designer, working with features within LTE. As a designer I’m involved in everything from the pre-study phase to the maintenance phase. Also I am a SCRUM master for the team which can be explained as a link to the outside of our team.

For how long time have you been working for Ericsson?
I’ve been working for Ericsson since 1999 and on LTE since 2009.

What does CI mean to you? What are the differences from ”before”?
It’s just an organizational move. Now we have people that actively work with CI. There is some kind of positive improvement.

How do you think CI is working for you?
It works fine. But it feels like before.

What is good with your ”way of working”?
When the team is getting more mature everybody can do everything which is good. But it takes some time to get there.

What could be improved?
Decisions are sometimes taken without taking everything into consideration.
11.2 Interview 2

What are you working with?
   Designer

For how long time have you been working with that?
   6 months

Do you have any experience about CI before this job?
   Somewhat, yes. I’ve worked for CIRV (Continuous Integration Release Verification). We maintained the test cases that the cross functional teams wrote.

What is CI to you?
   You follow up and execute tests more often to find faults faster.

Do you think you are performing CI?
   Yes. We are writing test cases that should cover the feature being implemented.

11.3 Interview 3

What are you working with? What is your role in the team?
   Designer and product guardian.

For how long have you been working for Ericsson?
   Approximately seven years in different positions in LTE.

What is CI to you?
   CI is something that I send my code to. If everything is working as intended it reaches the mainline. If something goes wrong I get feedback. The difference I see from before is the automation and the increased testing frequency. Also you cannot forget to run your tests since it is executed automatically.

Do you think you are performing CI?
   Yes, in one way or another. But I think it is occurring on different levels. The thing that might be missing is the connection between the written code and the multi component tests.

11.4 Interview 4

What are you working with? What is your role in the team?
I work as a designer with both implementation and testing.

**For how long have you been working for Ericsson?**
Started as a summer worker during 2013, after that I did my thesis at Ericsson and finally got a job in early 2014.

**Do you have any experience within CI before this job?**
No.

**What is CI to you?**
It feels like continuous testing and automated tests outside "me". Without CI there should probably be less integration testing.

**Do you think you are performing CI?**
Both yes and no. I think we are performing CI on different levels. The thing that might be missing to perform CI in a better way is more integration testing. What’s good with your way of working? It is a quite new way of working. However, the organization has not yet found a way to make use of Git’s full potential.