Reaching Software Development Maturity with Continuous Delivery

A Master's Thesis within the Department of Computer Science,
Faculty of Engineering, Lund University

Fredrik Ståhl
Viktor Ekholm

Abstract

Having a mature software development process signifies that it has undergone certain improvements to maximize productivity while the users of the process finds it simple to work with. In this master's thesis we have studied an approach to increase the level of software development maturity on a company by analyzing key areas in three cases of software projects on the company.

Our methodology was to arrive at designed solutions by performing analysis on the domain and determine the root cause to any problems visible on the surface. We were able to find that the most important first step was to develop a template for a common process.

Based on our experiences with implementations of our proposed solutions, we also arrived at our *Software Development Maturity Model*, which is an experimental model on how to determine the current level of maturity, set goals for reaching higher levels and ultimately improve a development process.

Keywords

Software development maturity, software process improvement, software configuration management, continuous delivery, continuous integration, automation, agile development

Contents

Preface

Configuration management has had different definitions over the years. One of the older definitions tells us that the goal of configuration management is to "maximize productivity by minimizing mistakes" [Bab86]. Another definition is that configuration management "represents general techniques for controlling the creation and evolution of objects" [Dar00].

As its name applies configuration management manages configurations and a configuration can been seen as a photograph of an object at a certain point in time. As the amount of variations increases for the components of the object, so does the number of available configurations. Without sufficient ways of maintaining all these configurations we lose the ability to tell what components and dependencies that was put into a specific configuration and we lose history on versions of all components. Variations will become blurred and we would soon enough, simple put, end up with a mess.

This thesis is targeted for individuals with some expertise within the subject of software development, be it developers, project managers, configuration managers or undergraduate students in the field of computer science. We will attempt to keep our discussions at a level so that any person belonging to the aforementioned groups can make use of the information presented and discussed in this report.

Acknowledgments

First of all we would like to thank our supervisor, Christian Pendleton, and our examiner, Lars Bendix. Their invaluable assistance and great engagement in our work became fundamental in keeping us on our toes and spur our own engagement. We would also like to thank all employees of Softhouse in Malmö, the company where we carried out the work you are about to read, for their hospitality and the willingness to participate in our interviews and demonstrations.

Chapter 1

Introduction

As fine spirits mature and develop a more complex and interesting taste over the years, the same can, almost, be said about software development. Having a mature software development process may not produce any extraordinary taste sensations, but it does give you a comfortable and effective work process. The prerequisites are to put in effort to actually mature the software process.

In short, improving your software process aims to rid you of any manual, repetitive processes and possible downtimes due to events you have to wait for to complete. Unnecessary rework is considered an anti-pattern and a very bad practice. Automating this hard labor intends to give developers a break from tedious, manual tasks, but in theory quality and consistency would improve. This is because we have a lesser risk of random mistakes due to automation.

This master's thesis have studied the practice of *continuous delivery* as a means for software process improvement. Through this practice we expect less integration problems as software code is integrated in small portions several times per day. Through automated test executions upon integration, we receive iterative feedback which tells the status of our software.

1.1 Problem Statement

Softhouse is first and foremost a consulting firm but also conduct some in-house development. These development projects became our focus and it was initially perceived that one of our premises was that the in-house developers did not have a common development process.

Through interviews with developers and stakeholders, our take on the in-house environment was that the nature of the company required developers to be given assignments on other companies between projects. Their knowledge of processes is then temporarily lost for the in-house development. As there is no collective knowledge of processes, it is common with processes suitable only for a specific case, i.e. a lot of work made ad hoc at project start. In a way the context has a relation to open-source projects, where we have to expect that developers come and go and thus require a process that would make is simple to start contributing.

It was perceived that no value is seen in taking the required time to mature and baseline processes. Work hours are considered to be billable and the customer is most certainly not interested in paying good money for advancing the processes of Softhouse after they already accepted them as developers. The interest for Softhouse in developing more efficient processes is to place more time and focus on developing the software and thus increase productivity.

1.2 Purpose

One of the purposes of this thesis was to, through interviews and analysis, locate key areas in the development process that could benefit the most from improvement. Although being a smaller company, Softhouse had multiple projects running simultaneously with different number of developers in each one, ranging from one to six. The first target was only the projects that developed applications for the Android platform and try to extend the process by adding additional features and tools. The intention was to improve, not only to increase efficiency of the development, but also the communication with customers.

After a certain amount of weeks we changed our target to a newly started project that had three smaller parts. These parts consisted of an ASP.NET website, an Android and an iPhone application, respectively. The motivation of this change of target was to have the opportunity of accessing and monitoring a live project taking shape in real-time, and not just basing our work on theoretical or old projects. Also, the diversity aspect of this three-parted project interested us. The primary goal then became to analyze the process used for each part and together with the developers work with improving key areas during the course of the project. It was discovered that one of the smaller projects contained a deployment procedure that was performed several times a day to a testing environment, which we saw as a perfect opportunity to study how processes are carried out on Softhouse.

There was no unified or general approach for projects, which had led to that developers on each part of the studied project had performed implementations on an ad hoc basis. Besides analyzing the reasons and root causes of the chosen methods we also had a goal to develop a common process with the purpose of introducing automated procedures for as many of the manual activities as possible. This would further benefit our own work, but our initial hypothesis was that it would also help to increase knowledge of well defined practices and assist in increasing the level of software development maturity on the company, which we had as a third goal.

1.3 Contents

Below we describe the chapter outline and touch on what each chapter is intended to discuss.

Chapter 2 describes and discusses the methodology used during our work and how we can perform an analysis to determine the problem domain, a basis for further analysis.

Chapter 3 handles the main problem analysis that were conducted during the course of the project and define a set of problems in the studied process that are derived from the perceived problems described in chapter 2. The set of problems are used as basis to derive requirements on solutions.

Chapter 4 will describe what it means to have a mature development process and dig deeper into the practice of continuous delivery, which our solutions are mostly based upon.

Chapter 5 takes the requirements that are defined in chapter 3 and argues for possible solutions. It discusses the advantages and drawbacks of each derived solution and later argues why we choose a certain solution over others.

Chapter 6 contains a discussion of how the selected solutions was implemented into two project cases and measurements that we performed to see the result.

Chapter 7 evaluates and discusses the "validity" of the work together with related studies. We also derive a maturity model based on the domain and discuss potential areas were future work can be performed.

Chapter 8 is a conclusion of all the results of the thesis and potential thoughts for the future.

Chapter 2

Background

In this chapter we will begin with outlining the methodology that was used during the course of the project. The sections of this chapter will also discuss how to compare projects and analyze the current process. In the final section we discuss how these kinds of analysis will lead to a discovery of initial problems that are perceived while exploring a specific project case with relations to the problem domain.

2.1 Methodology

During the stages of our work we reviewed several methodologies that could be of use for defining the domain, discover the problems that the developers were faced with, analyze their cause and designing our solutions. Since our tasks included producing a template for a development process applicable for the company and improving key areas, our approaches had to reflect that. We wanted to derive requirements on a process improvement from a problem analysis and use appropriate methods to implements these into the current process.

We will begin by describing all of the methodologies that we reviewed during the course of our masters thesis.

Interview. An ordinary interview, individually or in group, where we ask questions and, hopefully, get answers.

Questionnaire. An approach where questionnaires (or forms) are sent to the interview subjects, be it in physical or digital form, with fixed and/or free-text answers.

Workshop. Doing a workshop with the developers to get a view of how they really work in their actual environment.

Demonstration. Present a solution for a group of developers to discuss the implementation and receive feedback.

Test environment. To see if a solution is practical it can first be implemented and tested on an environment which is not used by development teams.

Documentation. Every step is documented as well as discussions and results from tried solutions, which later put into the final report.

Literature study. The method of acquiring relevant literature for the problem domain and reading it.

We decided on conducting individual interviews with persons of interest. We felt comfortable with this approach and believed the interviewees would feel the same, as they would not have to do anything but sit back and speak their mind. The basis for this choice, other than previous stated beliefs, was that we could talk and discuss with developers directly and freely. Based on the answers from interviewees we could also manufacture new questions on the fly if we stumbled upon an interesting subject.

The questionnaire approach would not, in our perspective, have produced these spontaneous discussions, as the questions would have been static. A workshop-like interview is also believed to be unsatisfactory as developers could rely on their tools to speak for them. If we were aiming to improve a tool set, however, a workshop would be preferable. In a regular interview, the developers are extracted from their work environment and would have to explain the problems and their processes in their own words, making it easier for us to determine e.g. the level of competence based on their phrasing.

The downsides to the interview approach that we were aware of, or quickly became aware of, was that that it was difficult to steer the interview in the desired direction. When we gave the interview subjects too much freedom during the interview they quickly diverged from our original question and began discussing technical solutions, which seemed to be a more interesting topic. By doing the interviews as a pair, we could have one of us conducting and steering the interview, while the remaining one jotted down notes and answers from the interview.

Using a demonstration to present a plausible solution turned out to be very effective. We were able to get more feedback from a demo than any another method. It was also effective with a testing environment, since we could be free to try out much kind of solutions without hindering someone's work. It became very necessary for us to document every step we took during our those experiments, since the later stages of the test environment contained a lot of information and trials that would otherwise not had made much sense and their connection to our analysis would been hard to detect.

We also did some literature studies within the problem area, but relevant literature was scarce. Our intention was to improve a process that would assist in keeping products releasable and their quality maintained, we initially made some studies within the practice of continuous delivery—specifically Humble and Farley's book on the subject [HF10] and other sources [FF06, HRN06]. Amongst our literature we also studied previous work on similar conditions.

2.2 Domain

As was briefly mentioned in the introduction and in the previous section, we were presented with some initial problems from the organization. There had been some issues concerning the development environment and they had a wish for a more simplified, but still controlled, manner to handle integration and deployment. The problems had originated, as was explained to us, from a build server that was no longer maintained and no one on site had the knowledge to maintain it. Since the company is a consulting firm, the personnel may receive assignments and take their knowledge and competence with them.

Before attempting at analyzing a project, we would need to define our domain, to know our boundaries, as the analysis is dependent on the domain. The domain specifies what we need to look for when doing the analysis. With a definition of the context and a comparison between the three studied project types (case descriptions) we can start explaining the initial problems in chapter 1 in more detail.

2.2.1 Defining the Contexts

As was described in previous section, we chose to conduct interviews to gain a better understanding of the problem domain we were dealing with. The goal of our interviews was to root out what was and what was not part of our problem domain. Whatever approach we choose to conduct it is the initial problems the company has laid out that are in the focus, because they are the most visible symptoms, so to speak. Most of the answers to our questions during the interviews turned out to be focused on technical issues at the organization.

There were many suggestions for solutions, which involved implementing different types of tool support into the development process. The tools suggested ranged from more control of the build environments to cloud support. Given a suggested solution we tried to work out which problem this solution or tool would actually solve and if the underlying reason was that "it is nice or cool to have" we became skeptical. During our interviews, we tried to have the following question in the back of our minds: if this is the solution, what would be the problem?

We learned that asking which problems that would be solved and what area in the development process that would benefit from these solutions, we started to receive information about the context and the way of working. We also detected the existence of three contexts. We defined them as *company context*, *process context* and *project context*. With this division, we can narrow the scope on an interview depending on the position an employee holds. Our assumption was that a developer would be able to give the most accurate answers related to the project context and a finance manager to the company context, as it is part of their daily work.

Company context contains items with information like the status of the organization, its available resources in terms of economy and the average competence on its employees of configuration management or knowledge of technologies.

Process context is the context of what type of work model (agile, waterfall, Lean, etc.) that are used in the development and planning processes. It also explains if any well-known practices are followed, for example *test-driven development* or *planning poker*.

Project context describes a unique project in an organization in terms of the team size and its setup, as well as the different personalities in the team.

In our case, we were mostly able to collect information related to the project context. The reason for this was that the majority of the interviewed personnel were developers. An optimal approach would have been to put most focus on the process context, since it was the process at Softhouse we wanted to work with. But given that the company did not have any employees with the sole assignment to maintain the development process, or something similar to a configuration manager, on-site the interviews gave us accurate descriptions of how project and teams were set-up and some loosely information about the development process.

The properties we looked at when defining our contexts are as follows:

Company size. Softhouse has, at the writing of this report, just about 100 employees and have offices in four cities in Sweden. Most of the employees are on consulting assignments at other organizations.

Competence levels. There is no configuration manager in the in-house projects, but there still exist knowledge of configuration management and its processes. The employees with the highest experience and knowledge of configuration management are often those that are assigned to other companies.

Knowledge of tools. Through our interviews, we identified that tool support is very common in projects. Version control tools exist in every project and there has been some

experiments with tool support for integration, code review, task management and issue tracking purposes.

Process work model. In almost all of the projects and the one that we studied Scrum was used as the work model. It is not followed "by the book", but the planning process inherits much from the model.

Practices followed. What we were able to extract was that the only practice followed was continuous integration. The practice itself contains several concepts, that will be explained in chapter 4, but we only saw evidence that integration and merging was done few times per day.

Project type. The project was a client-server application for cell phones. It was split up into three smaller project, two client projects and one server project. See the case descriptions in the following section 2.2.2 for details.

Project size. The project started with eight members in total. Later it grew as more employers were called in to assist.

Team setup (activity distribution). The team consisted of a requirement engineer, a project coordinator and the rest of the members were developers. Testing was done at the developers own discretion and there was planned test phase in the final weeks before deadline.

With the contexts defined, it is now relevant to locate a connection back to the initial problem to lay the foundation of further analysis. Since the three contexts will be the basis for further analysis, making a list of perceived problems on the whole domain will help us achieve traceability when analyzing a specific project in the same way that the initial problems defined by the organization helped us when defining the contexts.

The issues the company been having with their experiments with build environments can be related to the fact that there was no employer designated as configuration manager and there had been no formal education on the subject. Another piece of evidence is the knowledge of both tools and practices are different from employer to employer. This is evident in the initial problem that when a certain individual with knowledge of a particular tool or practice takes charge of setting up an environment, there is no guarantee that the environment can be maintained if that employee receives an assignment. We did not find any evidence that there had been any prior attempts at a tool evaluation. All of these three factors has lead to that projects are set up ad hoc.

Summarizing the issues we found during our context definitions we have:

- No designated configuration manager
- Scarce knowledge of well-defined practices
- No evaluations on tools have been performed
- Integration and deployment problems are solved ad hoc

2.2.2 Case Descriptions

As we saw in section 2.2.1 the project we were tasked to target contained three different smaller projects. As we wanted to fully understand the domain, we also was required to understand these project cases. With understand we mean that we wished to know of any standard conventions, if any particular tools are required for development and how objects and artifacts are managed.

The reason for this is that it puts a limit on the amount of variant solutions we can have for each case.

Following is a list of our cases, which were the focus for all of the studies in the rest of the thesis.

ASP.NET website. An ASP.NET *Web Site Project* (WSP) using Microsoft's .NET framework. This case is commonly locked to Microsoft tools during development and thus complicate matters if developed on another platform than Windows. Compilation of source code is performed at runtime, which means that no binaries are produced after a build. In other words, the source code files are the primary artifacts and to verify the system we have to verify that they can be built on a specific platform.

Android client. Excluding the functionality for implementing graphical interfaces, it is quite similar to regular Java applications. The development can be done on any platform, with the development kit installed, but running the application requires either a simulated version of an Android phone or actual hardware. A build produces a single binary file, which is deployed and executes the application.

iPhone client. Produces binary artifacts similar to a regular application written in C. Developing these applications, however, requires an OS X platform. Running these kind of applications during development requires that the cell phone is explicitly granted access in the source code. As in the Android case, we are limited to execute our application either on a physical phone or with an emulator.

The next step is to perform a comparison against a general project type and other project types within the organization, in a software configuration management perspective. This is required to extract the differences among the cases to later establish a solution for each case. The projects were compared to each other, as well as a general case. The basics behind our general case are derived from the research made by Humble et al. in the book about continuous delivery [HF10].

General Project. So what does our definition of a general software project entail? Well, for one thing we have the production of binaries. A binary in this context is the output file or files produced via the process of compilation. While this binary may not qualify as a true binary file, it is no longer the source code.

Moving on, our general real-world software project employs some kind of *version control system* (VCS for short). With a VCS we have the shared codebase in our VCS repository. This repository can be accessed by the project team members, having been granted access to it. Developers can update themselves against the central repository, write their code locally, and then merge and commit (integrate) their work to the repository so that the rest of the team can update their local copy with the new version.

As for unit tests, they are most commonly a part of the project, often contained in a separate test package inside the source code folder. Unit test source files are version-controlled in the VCS repository together with the actual application code. It is, on the other hand, common to have acceptance tests in a separate, external project, as these kinds of tests are to perform black-box tests targeting the application project.

Summarizing the above we have the characteristics of the general software project:

- Source code is compiled into binaries, which become build artifacts.
- A version control system (VCS) is used for storing and version-controlling the source code in a shared repository.
- Unit tests are a part of the project.
- Acceptance tests are housed in a separate project and performs black-box testing against the application.

ASP.NET Project. The first project we studied was, as mentioned before, an ASP.NET website. Comparing this project type to the previously defined general project, we can notice one major difference: the lack of binaries. A WSP utilizes *runtime compilation*, meaning that the source code is compiled at runtime on the server, instead of compiling and deploying binaries beforehand.

If we refrain from diving into the technical aspects of how this runtime compilation is carried out and look at it from a software configuration management perspective, we have a set of artifacts contained in a specific file structure that we are to deploy. This file structure includes directories and many different file types as the web format allows for many different types of scripts (both server- and client-side) to interact with each other upon access via a web browser.

The found characteristics of the ASP.NET project, differing against the general project:

- A WSP project uses runtime compilation, the source code and file structure of the project is the artifact to be deployed and are compiled and executed when accessed [Mit09].
- The production environment (operating system, runtime compiler) decides the output from the deployed source code.
- WSP's can alternatively be (pre)compiled [Mit09] for deployment or testing purposes.
- Necessarily not only one programming language, a website may use client-side scripts, markup files, style sheets, etc.
- Officially supported tools for ASP.NET, such as compilers or test frameworks, are only available for Windows.

Android Project. We also studied an Android application, which would serve as a client to the ASP.NET server application. Android applications are written in Java, and in many ways an Android project is similar to a regular Java project. However, there are some distinct differences which we will highlight here, together with the differences that Java development introduces compared to the generalized project.

Compared to the described ASP.NET WSP project where the source code files forms a set of artifacts, Android applications are compiled and packed into a single binary. This binary or executable can only run in an Android environment, meaning an emulator or an Android phone.

The unit testing strategy differs a bit from the general project in that it is suggested to have a separate unit test project [and]. This is because an Android application must be executed in an Android environment, i.e. an emulator or on a physical Android phone. Therefore the test project is also compiled and packed into a binary and deployed to an Android environment alongside the actual Android application, making it possible to test the Android application in its natural habitat. Worth noting is that tests not utilizing the Android API need not be tested in an Android environment.

The unique characteristics for the Android project compared to the general project are as follows:

- The source code is compiled and packed into a single binary file.
- An Android executable can only run within an Android environment, i.e. an emulator or Android phone.
- Android development is cross-platform.
- Unit tests dependent on the Android SDK must be contained in a separate project from the application.
- Testing that include function calls to the Android API is carried out by deploying the

compiled test binary alongside the application binary to an Android environment.

- No explicit restrictions on development environment or build tools, as long as the can, e.g. compile Java code.

2.3 Preliminary Analysis

Now that the domain and the three cases have been defined, analyzed and compared we can move forward with analyzing the process followed on a project on the company. In our case we had three smaller projects and we chose to treat them individually by analyzing the process for each case separately. As we will see in a while, there was distinct differences in the process followed in every case, which is precisely the reason for why we decided on this approach. To gain the information required to establish the current process we can again make use of interviews by the same reason that was discussed in section 2.1.

The information we are interested in here is mostly the kind that related to configuration management, e.g. the procedure to identify configuration items and how build artifacts are managed. It cannot hurt to find out what tools or development environments that are used and, more importantly, what problems they solve or how they support the process. This is because it might become evident that some tools are required in a particular case, which will become a required when discussing solutions.

- Project setup procedure
- Configuration identification
- Artifact management
- Tools used and to what end
- Test management
- Integration and deployment procedures

The first three bullets will give us information on how a project is managed in terms of general organization and configurations. *Tools used and to what end* was explain in the previous paragraph. By covering test management, we will know how the process handles the organization of tests related to a case description of the project type. The last bullet covers challenges with how different platforms are integrated, how environments work together and when integrations are performed.

For each case analyzed we will list any perceived problem that we identified as a potential issue. Any listed problem is linked to the preceding paragraph and is labeled unique for the purpose of traceability in later chapters. Due to limitation in time we had to make some priorities. Our top priority was to have at least one case fully analyzed and implemented. The second priority was to analyze another case for comparison. This lead to that we were unable to perform any further analysis on the client application project for iPhone.

ASP.NET Project. We started with the ASP.NET project because it was the only case with a deployment to another node through network channels. In that project the developers had chosen to work with Visual Studio that, together with the project type, required that unit test code be placed in a separate Visual Studio project. As there were different opinions on how and what to test they had decided to only write tests for their own code.

A.PP1. Tests are not part of the project; they are a project of their own.

A.PP2. The same person that wrote the code writes tests

They had made use of what should be a production-like environment. However, during our interviews we found out that the actual production environment was actually running on a different operating system than the production-like. Looking back to the case description of this project type, a part of verifying the system is to ensure that it can be built on a chosen environment.

A.PP3. Test environment much different from the production environment.

A deployment to the test environment is performed by copying the source code via a manually executed script. The files are not version-controlled on the test environment and are thus replaced at each deployment. When the system receives a connection after a change has been made it is rebuilt.

A.PP4. Builds are performed on development environment and then again on test/production environment.

A.PP5. Copies of source code exist on both development and test environment.

A.PP6. Tests on the client platforms that call the server can fail if a rebuild fails.

We were also able to discover that on certain occasions, changes to the system located on the test environment and the database it contained, were executed manually by directly accessing the node. The issue we saw with this procedure was that these changes never got recorded in any way and it had the risk of leading to a divergence between the two copies of the system.

A.PP7. Changes to the database and the system are not always recorded.

Android Project. For the Android application project the process was a very much common approach according to the documentation and case description of how an Android project is carried out. This project started out with only one developer and that developer had chosen to do all of his work in the Eclipse IDE. Eclipse creates certain configuration files for both the application and the local workspace. We found that these were present in the repository but not libraries that the application required to compile and build.

B.PP1. Workspace-specific files are checked into the repository.

B.PP2. Libraries, upon which the application is dependent, are not checked in to the repository.

According to the case description, unit tests are kept in a separate catalog. This was also the case with the project we analyzed, although the "test project" was also kept in a separate repository, which we did not find any evidence that it was a requirement. Actually it is not entirely correct to call the tests unit tests, since most of them called the API and making them more of acceptance tests or black-box tests. This also led to that execution of the tests required that the entire application be built, packaged and deployed to either a simulation of a cell phone or a physical one.

B.PP3. Tests are kept in a separate repository.

B.PP4. Production-like environments are required for all test executions.

During our interviews we become aware that it was an important that the graphical interfaces on both client platforms being identical. To ensure this, developers from each platform manually compared their implemented interfaces during development. These sessions were not documented and performed at their own discretion, i.e. not planned.

B.PP5. Verification of graphical interfaces are performed manually and not documented.

2.3.1 Perceived Problems

Looking back to the discussion of the problem domain as well as the initial problems we can see that an analysis of a specific case have revealed a link between the problems listed in the previous section with the problems for the domain. For the ASP.NET case there is evidently a lot of manual and ad hoc procedures, e.g. testing, deployment and changes to the database. The developers had different opinions of how to manage the project and the lack of a configuration manager might lead to the results that are repeated in the following list.

A.PP1. Tests are not part of the project; they are a project of their own.

A.PP2. The same person that wrote the code writes tests.

A.PP3. Test environment much different from the production environment.

A.PP4. Builds are performed on development environment and then again on test/production environment.

A.PP5. Copies of source code exist on both development and test environment.

A.PP6. Tests on the client platforms that call the server can fail if a rebuild fails.

A.PP7. Changes to the database and the system are not always recorded.

If we instead look at the Android project we had the case of a single person project. This lead to a majority of issues related to the structure and organization of the repositories and their artifacts. We chose not to delve into the standard convention laid out by the documentation for Android development. We were more interested in how the developer had chosen to manage his project based on the unique case.

B.PP1. Workspace-specific files are checked into the repository.

B.PP2. Libraries, upon which the application is dependent, are not checked in to the repository.

B.PP3. Tests are kept in a separate repository.

B.PP4. Production-like environments are required for all test executions.

B.PP5. Verification of graphical interfaces are performed manually and not documented.

Chapter 3

Locating the Gremlin

Originating from the air forces of the United Kingdom, a gremlin is a mythical creature which spends its days performing mischief. The airmen at that time used to blame faults with their aircrafts, for which they could find no cause to, on a gremlin responsible for sabotaging the craft. The main focus of this chapter is to discover the cause—the gremlin—to the perceived problems that we saw in the final two sections of chapter 2.

In the following section we will, for each case, repeat the perceived problems that were defined in section 2.3.1 and work to discover problems that are on a more abstract level. We will call these new problems "real problems". They are our basis for suitable solutions that will cover a broader scope than only the analyzed cases. Then we will perform an analysis to discover the causes that has spawned the perceived and real problems. In the final section we will use our real problems and root causes to derive requirements for solutions.

3.1 Discover the Real Problems

Before we start with detailing how we performed our analysis of the ASP.NET case and the Android case we want to explain the purpose of doing a root cause analysis. After gaining knowledge about the domain in chapter 2 and interviewing employees to discover the problems in sections 2.3 and 2.3.1 we could have begun to derive requirements and later solutions. However, what we do have to understand is that the perceived problems are only the visible consequences of one or more causes.

If we compare it to a disease: having a patient showing symptoms, the doctor is unable to treat the symptoms effectively until he or she actually knows the disease, i.e. the cause. By performing a root cause analysis this is exactly what we want to do: we want to treat the disease—the root cause—not its symptoms.

If we only solve the perceived problems then we will add the risk of them manifesting themselves again or perhaps another problem is introduced because we did not take into account the root cause. It is reasonable to assume that a cause has been solved if the symptoms have disappeared. Should symptoms still be present, then there is probably one or more causes left.

What is also important to remember is that a technical solution might not always be the best answer to a problem we have in a development environment. If we have a build process that seems to take forever or the lack of visual representations, a technical solution can assist us through other approaches to manage dependencies in order to speed up a build and construct graphical views based upon some calculation on e.g. the source code. These, however, are technical issues that are in turn solved by technical solutions.

We did not find a technical solution as an answer to our cases satisfactory, but instead as a means to support and implement solutions that are based on root causes. We wanted to break up the development process into smaller pieces and then focus on deriving requirements for solutions from these pieces—divide and conquer. The pieces are to be the real problems and how they were analyzed for our two cases is the topic of the following two subsections.

3.1.1 ASP.NET Case

In section 2.3.1 we discussed what we called perceived problems and the first perceived problem (A.PP1) was developers had chosen to have their tests in a separate project, which we viewed as an unusual approach when we discussed the general properties of a software project in section 2.2.2. More commonly a test package containing all unit tests is employed, but we found that because of the project type characteristics this was not a desired design. As all the source code files of the project were to be deployed to the test environment and compiled at runtime there when accessed, unit tests were suggested to be contained in a separate project which had a dependency to compile the actual application and performing its tests via this.

As for the perceived problem of having copies of the same source code on both development and test environments (A.PP5) we gathered through interviews with the developers that this was a wanted feature. Developers wanted to be able to make changes to the code directly in the test environment as a way to by-pass the deployment process. The rational behind this behavior was that including a deployment procedure was deemed as a two-step approach and thus would take more time.

A.RP1 By having two copies of the source code the double maintenance problem is introduced.

So here we arrive at Wayne Babich's famous problem of *double maintenance* [Bab86]. Since changes made in the test environment will diverge from the original code base, they will be overwritten at the next deployment if all changes are not remembered to be copied back to the development environment.

Moving on, we also had the perceived problem that tests are written by the same developer that wrote the code to be tested (A.PP2). Through our interviews with the developers of the ASP.NET project we found that one of the two developers only wrote unit tests for his own code, testing that e.g. his functions worked the way they were intended to.

The second developer had another approach where he only wrote API tests similar to the practice of black-box testing. With many years of experience this developer was convinced that these types of test had a tendency to capture more failures than unit testing. So there existed some differences in testing techniques among the two team members and they did not adopt the understanding of collective code ownership as they only cared about their own test project not failing.

For example, test runs were made at the developers own discretion, meaning that there was no process of pre-commit testing or testing after each commit to the repository. The issues we can have depends either on time or the amount of changes. If the first developer would check-in e.g. 15 commits in one day, it would not be an issue to locate an error. The other developer, however, would have to go through the history of 15 commits to locate the error. If we instead have two weeks on the 15 commits, then the first developer can have issues with remembering where the error could be located.

A.RP2 No certainty that tests are run after or before each commit/push.

Another possible problem we noticed when performing our preliminary analysis in section 2.3 was that the test environment differed from the production environment quite a bit (A.PP3). To clarify, the test environment is the server environment used as the deployment target during development, while the production environment is the actual server at the customer site.

E.g. the test environment was hosted on a Windows XP machine, while the server at the customer used a server operating system, namely Windows Server 2008. We tried to work out the root cause for this difference in setup, because if we had an identical configurations for both test and production environment we would have the same "parameters" for the runtime compilation.

However, from our interviews we gathered that the choice of using an old Windows XP machine boiled down to economical reasons. They already had this XP machine in place, and buying a Windows Server license just to assert that the application would, without doubt, work on the customer site was not considered valuable enough or economically feasible.

A.RP3 Proof in test environment that the application works does not prove that it works in production environment.

From this we arrive at yet another perceived problem, namely that builds are performed first in the development environment (i.e. via the Visual Studio IDE) and then again on the test/production environments at runtime (A.PP4). This is of course a common approach within script-like programming languages, but it raises some concern as the development environment may differ in configurations compared to test/production environments. Perhaps the code is dependent on some special version of the ASP.NET runtime compiler or framework, and when the application code is deployed, the lack of this prerequisite renders in failure.

A.RP4 Since a build is performed at the production environment after a deployment, there is no guarantee that it will work and failures are caught later rather than sooner.

We also observed the perceived problem that tests on the client platforms which call the server can fail if a "rebuild" of the server application fails (A.PP6). What we mean by a rebuild is that whenever source code files change—e.g. due to a deploy or manual changes on the server—the changed files are built again upon access. Should there be "faulty" code on the server or if the configuration (e.g. IIS version) does not comply with the code, the rebuild will fail and client applications are unable to run their tests.

This is not an uncommon issue with software; that an application no longer functions on, e.g. a new version of an operating system. Thus, we see this more of a compatibility issue than a real problem.

At last we noticed some issues for concern relating to database management. We did not really find the time to pursue this concern properly, but the perceived problem was that changes to the database are not always recorded (A.PP7). Since an arbitrarily chosen version of the application code is dependent on a specific version of the database, the application can not run if it can not find the desired database entries or tables, etc. So we have no rollback strategy since the database is not version-controlled in any specific manner, but only modified directly in the test environment.

A.RP5 The application cannot be rollbacked to an earlier version if no database version for that application version exists.

To conclude this analysis on the perceived problems of the studied ASP.NET project, we mainly found problems relating to the design with having source code files as the deployment artifacts. We discovered for example double maintenance and uncertainty of build status at another platform. What we intend later on is to display that our solutions will produce lower costs and higher benefits than using this process.

Real problems for the ASP.NET project:

A.RP1 By having two copies of the source code the double maintenance problem is introduced.

A.RP2 No certainty that tests are run after or before each commit/push.

A.RP3 Proof in test environment that the application works does not prove that it works in production environment.

A.RP4 Since a build is performed at the production environment after a deployment, there is no guarantee that it will work and failures are caught later rather than sooner.

A.RP5 The application cannot be rollbacked to an earlier version if no database version for that application version exists.

3.1.2 Android Case

The same type of analysis as we saw in previous section will here also be conducted for the Android case. Starting with the first perceived problem derived in section 2.3 we had discovered that the repository of the Android project contained configuration files that were specific to the workspace of the developer. These configuration files are automatically generated by Eclipse, which was the choice of development environment in the project.

By further interviewing the developer and performing a quick study of the Android documentation for managing projects [and] we learned that Eclipse had been chosen because of the fact that it contains all that the developer required in one package or bundle. It is also recommended in the Android documentation and the development kit can be directly integrated. Going back to the perceived problem, when trying to check-out the project ourselves failures were generated because of non-existing, dependent libraries in a location that corresponded to the file system of the developer (B.PP2).

What has happened is that there has been a mix-up between team-related and personal files. Having the workspace-specific files in the repository (B.PP1) and tracked in the version control system, we would run into collisions if the project contained more than one developer. If we, from the start, had identified and separated team-related and personal files it would be simpler for any additional developer.

B.RP1 No configuration identifications are performed.

We also had the condition with the dependent libraries (B.PP2). Having the project set up this way such that every additional developer are required to download the libraries from some location, make sure it is the same version and manually configure Eclipse so it knows where the libraries are installed do we not consider efficient. We would therefore like to add to the previous analysis that to reproduce the application is not a simple procedure, i.e. it would not possible to clone the project to a new workspace without any manual configuration.

B.RP2 The project is not easily reproducible.

The third problem that was perceived for the Android case in section 2.3 was that all of the unit tests were kept in a separate Eclipse project and also in a separate repository (B.PP3). According to the Android documentation it is a standard convention that all test code is separated from the production code. It is, however, not explicitly stated that test code should be in its own repository—in fact it is recommended to place all tests in a subdirectory under the main directory of the application.

The only explanation we were able to discover was that the developer was used to working this way, but it still has its consequences. Again, should additional developers be assigned to the project we risk that the "test repository" is not synchronized as often as it should which leads to an incorrect number of executed tests. By analyzing the log entries of the repository we could also detect that the test directory received very little attention, i.e. few tests. Our conclusion was

that the perceived problem originates from an ad hoc project setup and it gives us the problem that failures are not captured during the implementation of a feature.

B.RP3 Failures are captured later rather than sooner.

The fourth and fifth perceived problems both have a connection to a production environment. The former stated that such an environment is required for all test executions (B.PP4) and the latter showed us that verification of graphical interfaces are performed manually on unscheduled occasions (B.PP5). We performed some experiments with both simulated and actual hardware to gain a deeper understanding of the variables included in software testing for Android applications. We performed the experiments with the same software project that we analyzed so as to avoid having worrying about scale.

What was almost immediately discovered was that during test execution, the majority of total time was spent on deploying the application and the test code to the hardware. On a simulated device it took about twice the time for a deployment. It was also very demanding for the device we performed the experiments on, granted it was not the latest super computer, but it still showed a significant increase in hardware usage.

It is possible to prevent a deployment for certain test executions as long as they do not utilize any functionality restricted to Android devices and the Android API. Since this was not the case in the project we analyzed the cause for the fourth perceived problem is a choice made by the developer whom only has interest in testing functionality directly on the device. Through further interviews we were able to discover that the reason for manual verification of the graphical interface was because of no knowledge of a tool that could perform a similar visual verification as the human eye.

The fact that a production environment is always required and that new tests are scarce has led to a lack of management for older tests as well. We have to accept that it will take some time to perform packaging and deployment, since this is nothing that we can change. However, without any regression we have the risk that changes introduce undetectable defects.

B.RP4 A change can introduce undetected failures in production .environment

To sum up the analysis of the Android case we are dealing with a project that has most of its problems related to how it is set up, i.e. its structure of files and how tests are managed. All that we discovered might have a strong connection to the fact that only one developer was assigned to the project, together with no company standards. This would be a typical example of an ad hoc project.

Real problems for the Android project:

B.RP1 No configuration identifications are performed.

B.RP2 The project is not reproducible.

B.RP3 Failures are captured later rather than sooner.

B.RP4 A change can introduce undetected failures in production environment.

3.1.3 Commonalities

With the two cases analyzed we can try to abstract the root causes by first identifying the commonalities. These will assist in forming the real reasons for our discovered issues in the two previous sections. By focusing on the causes and not the symptoms we can derive solutions that each will cover more than one problem.

The first common factor we have identified is that a project is not set up and managed based on a template or a common process. They are instead set up ad hoc. We saw this through the problem with double maintenance in the first case and the issues with reproducibility and configuration identification in the second case.

Still on the topic of template processes we also saw an indication of issues regarding the management of production-like environments. Without any established goals for maintaining a test environment that mimics the production environment, developers resort to instead take what they see as most similar, but may in fact not be.

If we now focus on other activities that are part of development, except for implementation, our analysis revealed that testing the software is considered an inefficient usage of time and is instead pushed back to the final stages of a project. Granted, some testing is made during development as well, but in such small quantities that it cannot be deemed as part of the development process.

As a third commonality we have that a lot of the activities, which are performed on a regular basis during development, are performed manually. This gives us the issues we saw with the developers taking shortcuts, making changes to versions on other environments manually, and not performing any regression tests, because they take time to complete and decreases the performance on a workstation.

All of the discovered commonalities that are the root causes for the perceived and real problems from the analyzed cases:

RC1 No company standard or templates for a development process.

RC2 Feedback on implementation is only given at the final stages of a project.

RC3 Repetitive procedures are performed manually.

By constructing this abstract view of the real problems we have discovered the causes to our original symptoms. In figure 3.1 we can see the connection between our established real problems and the derived root causes. The purpose is not to display the traceability from a real problem to its cause, but instead to show that by focusing on a root cause instead of a problem, we ultimately solve a set of problems.

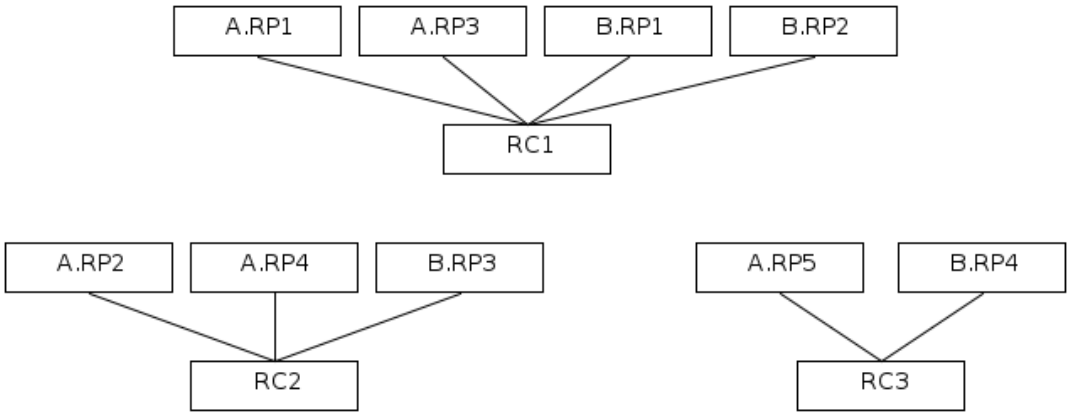

Figure 3.1: The amount of root causes compared to real problems.

Depending on which types of problems and goals we have in a company we are able to narrow our scope to a single root cause. For instance, let us assume that a company has placed a new goal which aims to increase the quality of their product. Once we have found problems that, if solved,

will assist in increasing the quality, we locate the cause of these problems and then we only require a solution for one cause and not one solution for each problem.

In both cases we can derive requirements for solutions that are not solely targeting a specific case or project context, but instead tries to target the whole domain. We realize that with further analysis of more cases within the company would have been preferable and perhaps required. We prioritized to experiment with solutions instead of a complete specification.

3.2 Define Requirements

Having discussed how we performed our in-depth analysis to discover the causes to the perceived problems we can now conclude its result into a set of cross-cutting requirements that we place on a solution. The requirements should to cover the real problems from the previous analysis by targeting the root causes and take the problem domain into consideration.

As a first requirement on our solutions we will have to accept the fact that having a too complex solution will lead to it not being used or maintained. This was the case with the previous build environment that had been set up as an experiment in the company. Only the ones responsible for that environment had the knowledge to configure it, which eventually resulted in no more maintenance, once they got assignments to work at other companies.

To define what is and what is not considered a complex solution is no small matter. It does, however, put a limit on the amount of available solutions we can have and that are seen as acceptable. We would settle with the definition that if the cost to implement and use the solution in practice outweighs the benefits the solution can be considered complex.

As an example of the previous motivation, we should not have to go through any extensive training or education before we are able to use a solution. This requirement will always exist in the background and thus has to be validated for every future addition to the solution.

R1 The cost for a solution must not outweigh its benefits.

Having a test or production-like environment (e.g. A.RP3) is not only limited to the two cases we studied, since almost every project within the company is targeting cell phone platforms. A suitable and abstract requirement for a common process would then be a requirement that covers support for multiple environments in a development project. A solution has be able to work with more than environment, thus be aware they exist.

R2 The solution needs to support working with multiple environments and be aware of their existence.

If we are to aid the need for a company standard for setting up and maintaining projects, as was discussed in section 3.1.3, we require a centralized environment on which applications can be built, tested, etc. In the cases we had the existence of two copies of the source code (A.RP1) and a root cause of a single developer spawning issues related to file structure (B.RP1). Without a standard we would have to decide on every project how and where to manage files such as build and deploy scripts, which goes against what we are trying to solve.

R3 We require a centralized node where we can check out or clone from a repository, perform builds, execute tests, i.e. perform the same activities as are done during development.

We have seen that the current process lacks feedback during development. In the ASP.NET project there was no certainty that the tests were executed before or after each integration (A.RP2) as well as no guarantee that the application would build (A.PR4). In the Android project we also had that feedback on implementation was given late in the development process (B.RP3).

If we are given feedback a long time after the functionality is actually written, we argue that finding errors will take more time as the developer has to go through his code again to actually work out what went wrong. To achieve feedback on the code committed to the repository, we need to require support so that builds, as well as tests or code metrics can be executed.

R4 Support has to exist for the implementation, execution and publication of results for tests and code metric analysis.

Still on the topic of feedback and again relating back to A.RP2 and B.RP3, we would like to extend the previous requirement by adding a second one detailing an acceptable time span between integrated changes and the generation and publication of feedback. Amongst our primary literature we were able to find that it is reasonable to at least accept 60 seconds [HF10] before we can start pulling our hair in despair. Like our first requirement, this is also considered as a background requirement.

R5 Feedback on checked in changes should not take more than 60 seconds

The final requirement we are going to place on our solutions is related to automation. As we discussed in section 3.1.3, a common behavior, in both cases, is to perform repetitive activities manually. An example of this was discovered as a test-related issue in the Android case (B.RP4). In the ASP.NET case the developers took shortcuts to change a database (A.RP5) on the test environment because it cost them less in time.

In the long run it will be considered inefficient and even to the point where it is tried to be avoided. This is partly because of the opinion that implementing is much more fun then, e.g. transferring files from one location to another.

R6 It has to exist support for automating procedures and activities.

Now we have defined a set of six somewhat abstract and general requirements whose connections to root causes can be viewed in figure 3.2 and which would help us find practices and design a solution with the focus of improving the software development process at the company. We want a non-complex, multi environment-oriented system for which the user (developer) does not need to adapt to very much. Quick feedback should be considered essential and therefor we need a centralized system which can clone a repository, perform a build, run tests, publish results from tests and code metric analysis—all within 60 seconds. Last, but not least, again automation is key and we need support in our solution for automating repetitive and manual processes.

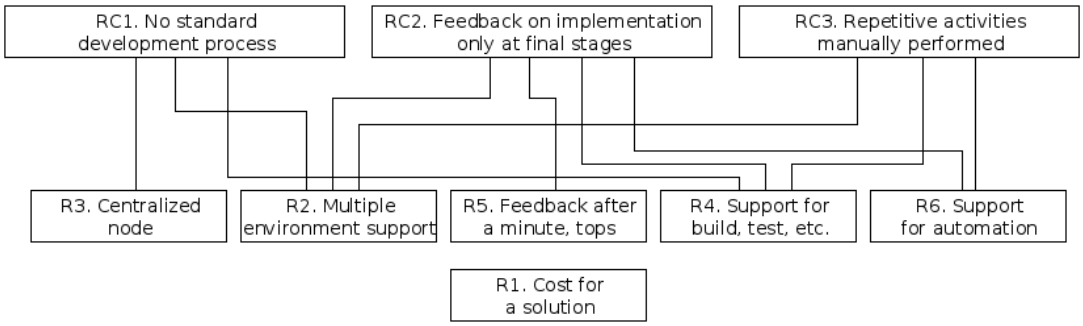

Figure 3.2: The connection between our root causes and the requirements.

In the next chapter we will use the discussed requirements to limit our scope when attempting to choose practices and techniques to use in our solutions. The requirements will be matched against the general concept of the reviewed practices. Should we find practices that suit our domain and takes care of our root causes we should be golden, and can then start designing a solution in chapter 5. We can then validate and evaluate against the stated requirements for assurance.

Chapter 4

The Cornerstone

Having gone through the process of identifying the problem domain followed by a root cause analysis on the perceived problems that were discovered during an initial analysis and interviews we can now begin to start thinking about solutions. We have our real problems and requirements for solutions from the previous chapter, but now we require methods and approaches that are theoretical in nature, which will then be the basis for the actual solutions, which are more technical.

This chapter will entail the approaches we found during our research and how they relate to the problem domain we were facing. We will define any requirements and constraints placed on a development team when improving the development process as well as argue the advantages and drawbacks for the approaches we have chosen. The chapter can be seen as an introduction to designing solutions for the real problems and root causes in the previous chapter.

4.1 Software Development Maturity

First we will look into an approach that serves as a mean to identify and improve a development process. Since we are aiming to develop a process for the company that they should be capable of improving themselves, we require a method that can model the current situation. Having goals for a software process improvement will assist in locating the key areas that our efforts should target.

Being able to perform something effectively means that we have the capability to successfully accomplish goals that has been set out for us. It might not be a efficient solution, which refers to the notion of making sure to minimize required time and effort during our process, but it is at least a complete solution. Questions raised are now related to how we can be certain that goals are accomplished, i.e. how can we validate our solution? The goals might also not always be laid out for us, so we would then be first required to define them and again how do we accomplish that?

During our research we came across a model called the *Capability Maturity Model* (CMM) [Bur03, PCCW93], which uses a concept called *software development maturity*. Now the whole idea behind software development maturity and the CMM is to assist in software process improvement by defining both goals but also procedures to validate how mature a process is, i.e. the level of maturity. For the interested reader there is a whole book on the CMM with much further detail [PWCC95].

In our case we will relate to our own context and problem domain when defining software development maturity. What we require is primarily a means to effectively measure the processes we have analyzed to determine any anomalies from standards and if a solution actually was an improvement. The purpose is to create awareness of how much or how little a process has developed.

With the analysis on the domain that was made in chapter 2 half our work is already done for us, now we just need to value our findings.

The cost of an implementation of a software process improvement can vary depending on what area of the development process that is targeted and so can the benefits. We can, for instance, have a solution that requires a week of training before it can be efficiently used by the developers. Since these variations exist it would make the lives of the software process improvement personnel easier, as well as the developers if areas in the development process which require the least effort to improve that also give high benefit could be identified. The same goes for the vice versa, which we will call bottlenecks.

The final piece to the puzzle of our definition of software development maturity is the ability to produce goals that will serve as the main purpose of reaching higher levels of maturity. With information gathered for the factors in the previous paragraphs, we now have the possibility to set the goals for process improvement. The main purpose of having maturity goals is to, again, validate our improvement against the changes that have been implemented.

We have discussed four different purposes of our definition of software development maturity that are outlined in the following list.

- Measure how far a process has deviated from standards
- Determine key areas that requires the least effort for improvement
- Identify bottlenecks
- Set goals for process improvement

The previously mentioned topics had one major thing in common and that was validation. The next topic we would like to discuss is therefore how we can effectively perform a validation on a software process improvement. But before we do that, a question had certainly been raised as to why all this matters and what a company can gain by improvement through software development maturity.

First of all we have the, potentially obvious, factor of increasing efficiency. In one of our cases we had a situation where a project was setup ad hoc with dependent libraries only on a local location. Now this was fine until the point came when another developer joined and required a clone of the repository on his own machine. Since he was not present from the start the first developer had to assist in setting it up which halted his work. This might be seen only as a common mistake, but the fact is that without any standards in project setup these kinds of problems will repeat themselves.

Standards and template procedures are therefore wanted in our context and we would argue that it is another benefit given by a maturity model. As we will see in the next chapter, designing solutions for software processes requires some level of abstraction and that is where templates come in. These standards are defined based upon the business goals of the company. In fact, having a maturity model specifically designed for the company domain will thus make sure that the company goals are, drum roll please, validated.

So far we have discussed how maturity models work to increase efficiency in development, validating company goals and turning ad hoc processes into standards or templates. There is one more point we think is worth mentioning and that is awareness. Now it is not about general awareness and to know when to look over your shoulder, but awareness of the development process and the tools it uses. Producing and maintaining templates and maturity goals force us to study and ponder on what before was taken for granted.

What is left is of course a practice or approach that will detail how to reach higher levels of maturity. As we see it, there are no general cases for this. Even though the CMM has been around for a while it cannot cover all types of company, process and project contexts that exists.

Before starting design for solutions, we first have to outline a methodology they will be based upon. Considering our domain, we have a requirement of continuously integrating and deploying.

4.2 Continuous Delivery

Before moving on, we should again reflect on the results from the previous chapter. What is shared between the two analyzed cases is that they both require a production-like environment during development to run specific tests. We have seen that both cases struggled with issues related to capturing failures as fast as possible and no regressions testing can lead to undetectable faults. The Android project also suffered some issues related to reproducibility and the ASP.NET project lacks guarantees that the system will successfully build.

Since our project context involves smaller development teams we do not need to worry about any constraints on the network load if more information should be passed after each code check-in. We do however need to take into consideration that we do not have an elite special force of testers and thus that burden is places on the developers. We are dealing with an agile development team, which means that it is feasible to assume that change requests from the customer is somewhat common.

What we would like to have on our recipe for designs is an approach that gives us the benefits outlined in the following list.

- Fast feedback on changes
- Always make sure that the code has passed tests and quality checks
- Larger scope of history, i.e. more traceability, then just changes to the source code
- Repetitive activities are automated

We require fast feedback so that we can ensure the second bullet that our application is working and quality is preserved. If not we cannot be sure that our product is releasable. With more traceability we can speed up the process of locating errors or defects. By making our repetitive activities automated, we can spend more time on implementation and increase our efficiency.

A concept that fulfills the preceding list, that we were able to find, is called *continuous delivery* [HF10]. This section will discuss how the different parts of continuous delivery will assist in designing solutions, which is performed in chapter 5.

4.2.1 Continuous Integration

Having already established that continuous delivery is a practice that may suite our domain, following it places a requirement on developers. If we want to have the possibility to capture failures early and increase our traceability we need the increments to be small or else it will take too much time and we lose our fast feedback. Relating back to the definition of our contexts (see section 2.2.1) we mentioned that integration to the shared repository was performed a few times per day. Thus in our case, we already had evidence of continuously integrating small changes to the repository with an up-to-date code base.

If we look to the real problems discovered in section 3.1 we found that for both the ASP.NET and Android projects there was no way of knowing if the project compiled and passed the unit tests as this was done when a developer felt the need to. This means that in some occasions a change was introduced into the central repository without any confirmation or feedback on its impact on the whole system. Should we instead apply for an approach that ensures that every increment is tested, we would rid the projects of stated problem and pave the way for introducing continuous delivery.

One such practice is *continuous integration* and was first introduced, or coined, through the birth of the agile software development methodology named *Extreme Programming* [Bec99] (commonly abbreviated as XP). It is one of the twelve practices followed in XP and is also the essential building block within the "parent practice" of continuous delivery. For the reader who is not familiar with continuous integration we will give a quick overview. Kent Beck's [Bec99] original two-sentence definition of continuous integration reads:

> New code is integrated with the current system after no more than a few hours. When integrating, the system is built from scratch and all tests must pass or the changes are discarded.

In an article by Fowler et al. [FF06] on the subject, the key practices within continuous integration are discussed. The purpose of continuous integration is to keep our code base up-to-date and the merges minimal. In short, the practice values having working and tested source code, reproducibility and automation of manual and repetitive tasks (i.e. building and testing).

4.2.2 Continuous Deployment

Should we look back to the preliminary analysis in section 2.3 we briefly mention that the studied ASP.NET project used a script for copying the source code files (for this project type the source files were equivalent to the build artifacts) to their test environment. This *deployment* script was executed manually by developers and we would like to ask—why?

Why can we not automate the deployment of our artifacts as an extension to the continuous integration practice? After having produced a successful and tested build, triggered by a commit/integration of code from a developer, why not simply deploy/copy the build artifacts as a post-build step? According to a white paper by UrbanCode [Urb12] deployment automation reduces script maintenance, introduces deployment error prevention and boosts efficiency in the long run by saving time on automating manual processes.

So deployment automation may be a key practice to employ, and this is exactly what the second part of continuous delivery, namely *continuous deployment*, attempts to accomplish. We realize that the similar names of the mentioned practices—that is continuous delivery, integration and deployment—might be a little confusing. Continuous delivery combines continuous integration and deployment.

At the risk of repeating ourselves: continuous integration is the practice of integrating code often, keeping a reproducible repository with working and tested code and automating builds and tests. Continuous deployment, on the other hand, handles the produced build artifacts by aiming to keep us performing a deployment to a test or production environment every time a new version has been approved.

4.2.3 The Deployment Pipeline

Up to this point we have discussed the importance of feedback for measuring a process improvement and during development. We have also discussed how we can use automation to our benefits to lower the cost of performing certain activities such as testing and deploying. What is left is to investigate how this can be achieved and successfully implemented in practice. In our case we are dealing with changes that are committed several times a day and though their sizes vary, they tend to be small, 10–30 lines of code (excluding file removals).

For every case there is a deployment process involved and tests that can and should only be performed on a dedicated test environment. We did not want our solutions to add an extra cost for developers by forcing them to switch to different workstations during development, insertion and verification of changes should be performed behind the curtains. Concerning the extra steps,

except implementation, during development and that further improvement might lead to more steps, we would require an approach that gives us feedback on every step of the way.

Since we did not have the luxury of time to reinvent the wheel we started by investigating any existing concept that would fill our requirements. The research of continuous delivery introduced an approach that is called *The Deployment Pipeline* [HF10, HRN06] and its purpose is to divide a release process into automated stages that are kicked off once a change has been introduced in the repository. Now, we are of course not dealing with release management per se but we still see a potential in this concept that we can exploit for our own context.

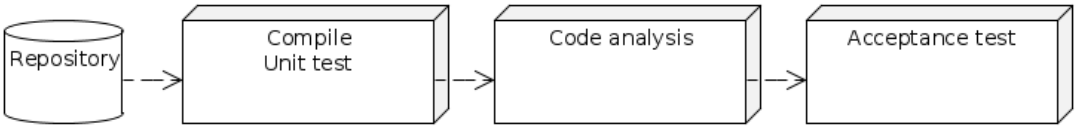

Figure 4.1: A simple pipeline.

With a build pipeline we will fulfill our requirement of fast feedback on an implemented change, that any failures gets caught sooner than later. If a failure would occur on the unit tests in the first stage of figure 4.1 it will immediately be reported and the procedure will stop. If we also implement it incrementally, one stage at a time, we can fall back to a previous version of the pipeline in case something goes wrong. For a more detailed description of a deployment pipeline we recommend the book *Continuous Delivery* by authors Jez Humble and David Farley [HF10].

Chapter 5

Designing the Solution

In chapter 3 we formulated a set of problems on our analyzed cases, that we called real problems. We then used these problems to derive cross-cutting requirements that we place on our solutions, with the ultimate goal to solve the set of problems. In chapter 4 we discussed practices and principles that we will base our solutions on. As was mentioned in that chapter this is only one approach, and part of our master's thesis was to analyze the possibilities of this approach in improving a development process.

We have now arrived at the point where we will begin designing solutions and we will do this by combining the requirements with the practices. This is, again, a part of our study. In this chapter we will explore the possibilities to design solutions with the aim of improving a development process and to reach higher levels of software development maturity. We should remember that the whole domain needs to be considered, since our goal is to improve the development process on the company, not just the two cases we studied.

The following sections will have a step-by-step approach. First we state the problem we wish to target. Next we sketch different solutions and discuss their advantages and drawbacks. The last step we take is to choose a particular solution amongst the alternatives, which we will argue fit the most. In the final section we will summarize the results of our designs.

5.1 Template Development Process

The first problem we are going to address is how to obtain both feedback and automation in a common process for projects. During our preliminary analysis and interviews we discovered that there did not exist any general or common process for projects.

A.RP1 By having two copies of the source the double maintenance problem is introduced

B.RP1 No configuration identifications are performed

RC1 No company standard or templates for a development process

R3 We require a centralized node were we can perform the same activities as are done during development

Developing and experimenting with a template process do we see as a fundamental building block to a software process improvement and what is required as a first step, if no such template exists. In our case it will deal with the two problems in the preceding list and the root cause, e.g. make two copies of source code complete unnecessary.

Workspace Model. If we begin with an alternative solution, which will ignore the previously mentioned requirement, and instead let each developer perform their development activities at their own convenience.

This will not force anyone to conform to a particular methodology, as they would be free to work in a way that they are used to and are comfortable with. Relating to our definition of software development maturity it would indicate that this solution might spawn processes on high levels of maturity, because of the experiences developers have.

One of the negative aspects of this solution is that we have a high risk of collision in interest and opinions, which will lead to a longer decision making process. In our interviews we found that part of the reason why projects had a slow start was because a great deal of time was spent on discussing and making decisions on what methodology that would be suited the most. These types of conversations would continue even during development and most of the time it eventually led to that the developers fell back to their own approaches.

Integration server. The opposite solution is to cover requirement R3 by having a unique workstation where we can perform all of the activities that are performed during development. It requires a connection to the repository and that every single activity performed by any developer can be executed on that station.

An expected cost with this solution would be increased maintenance now that we have a station that is always required to be up-to-date. By that we do not only mean that versions has to be made sure to be the latest but also any new activity has to be integrated as soon as it is performed in a project. With a template process in place is expected be a one-time cost at the start of a project and shorten down the project setup, because it is already decided what approach to use.

Among the benefits it will bring is the capability to perform time and hardware consuming activities while implementation can continue. As we saw in the Android project, we had to wait for a deployment to finish before we could do something else. A perfect time to go and grab a cup of coffee, one might say, but depending on the amount of deployments per day we might end up with a few too many coffee breaks or we develop a frustration.

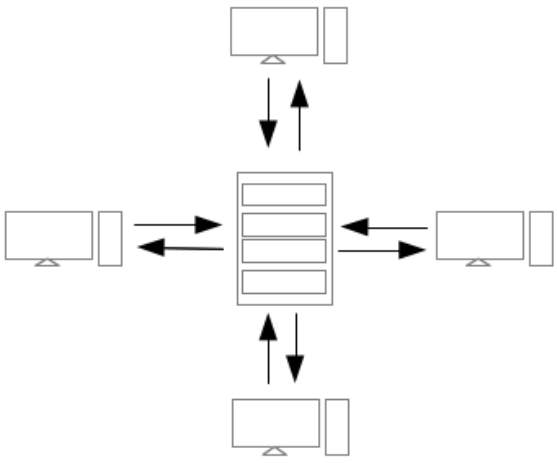

Figure 5.1: Integration environment.

The solution we favor the most is the latter with an integration server for which we have a visual representation of in figure 5.1. This is mostly because it covers are third requirement, which the former solution does not. It is also because this will assist in keeping the needed resources on a developer's workstation to a minimum by performing activities, such as acceptance test, which we found can consume a lot of resources.

5.1.1 Feedback

Now that we have designed a strategy for how we can obtain a template development process we can begin with how to integrate continuous feedback in a common process. With continuous feedback we mean that every integration into the repository or the mainline is tested and analyzed, by whatever means the development team have chosen, and that the result is published as feedback.

We can have different types of feedback generated by our integration server. We will not delve into which type of feedback is the most preferable, since it is outside the scope of our thesis. The focus of our solution is instead that, given any type of feedback, how do we best present it? Following is a list of typical activities we can expect feedback from.

- Unit, acceptance, security and performance tests.
- Static code analysis
- Analysis of code complexity
- Code coverage

In our interviews and analysis we discovered that feedback was not given in any particular great quantities until the final stages of development. A few tests were created during a project as well, but they were never any certainty that they were executed before or after integrations.

A.RP2 No certainty that tests are run after or before each check-in

B.RP3 Failures are captured later rather than sooner

RC2 Feedback on implementation is only given at the final stages of a project

R4 Support as to exist for building, testing and analyzing a software product as well as publication of their results.

The aim is to utilize the integration environment to ensure a capability of generating feedback. We do this by covering requirement R3 and implementing support for executing development activities. The solution would thus have to be simple enough to execute, since we might require executions several times a day.

Build Pipeline. In our discussions in the previous chapter and section 4.2.3, we discussed the approach of separating the different stages in a release process with a so called deployment pipeline. We can use this approach and apply it to our problem of generating feedback in a common process.

If we have any preference regarding a type of feedback that we expect to be delivered not long after our change has been integrated, the build pipeline will handle it for us. Think of requirement R5 in section 3.2 that said that feedback should be generated in under a minute. We would simply choose to execute those activities as a first stage in the pipeline.

The negative aspects of this approach is that we would have to be careful in both organizing the build system of a product and managing application binaries by making sure they are not part of the original version control [HRN06]. If we are not, we risk ending up with a processed build system with non visible values and breakages in the attempt to merge two different binaries.

Single Stage. As the opposite of the previously mentioned solution, an alternative would be to perform all of our activities in one single stage. As soon as it is done a single report, containing the results, would be generated and presented.

We could modify the previously mentioned solution so it, e.g. upholds the principle of "no news is good news", meaning that only failures will be present in the report, thus minimizing its potential size. The problem with this approach is again the lengthy activities. If we would only want to see the results of the unit tests, we would have to wait for any other activity to complete.

Pre-tested Commits. Instead of placing the generation of feedback on our centralized integration server, we could let each developer workstation also serve as a quality gate.

By executing tests and code analysis, that are deemed most important to maintain quality, before each check-in to the repository we will have the guarantee of always working code and preservation of quality. The negative aspect is that occasionally developers would "forget" to execute the tests, as was discovered during our interviews on the company.

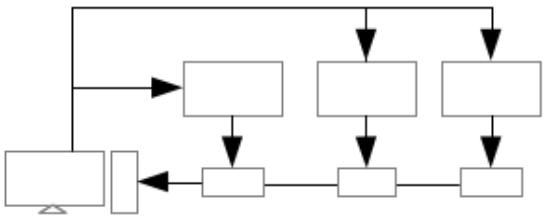

Figure 5.2: Build pipeline (without any automation).

Our most favorable solution is the first solution is this section, the build pipeline, as illustrated in figure 5.2. The reasons for this are the fact that we can prioritize activities and generate feedback for each type of activity. It is also very simple to automate, which will assist in covering our requirements for automation support.

5.1.2 Automation

With feedback in place together with a common process we will now study the possibilities of automating our build process and thus covering requirement R6 with support for automation. Performing the type of activities repeatedly can eventually become tedious and boring. In our interviews we found evidence that executing tests, performing a deployment and analyzing code complexity is considered less "fun" than implementing functionality into the product.

A.RP5 The application cannot be rollbacked to an earlier version if no database version for that application version exists

B.RP4 A change can introduce undetected defects in production environment

RC3 Repetitive procedures are performed manually

R5 Feedback on unit tests should not take more than 60 seconds

R6 It has to exist support for automating procedures and activities

We also need to consider the requirement that puts a limit on the time it should take to execute unit tests and present their results. Because most of the activities in out analyzed cases did not have much automation, the developers had resorted to develop their own, manual, processes that they found less costly, but gave us problems like the ones in the preceding list.

Automated Deployment. One of the most performed activities in our domain is deployment. This is because developing to cell phone requires that we occasionally transfer our application to a simulation or actual hardware, or else we cannot perform function calls to the phone's API. In other words, in our contexts, deployments are a highly repetitive activity.

If we are to automate the deployments we have to be careful in choosing our approach. One approach is to develop a script that transfers all necessary files from one location to another of our choosing but this will add another configuration item in our repository and would not serve our common process strategy.

We can do a much better job in regards to optimization by only transferring files that was recently changed since a previous deployment. A simple command and possible the push of a button, executing a procedure in our build script, would be much more preferable. This is exactly what we had in our Android case.

Automated Pipeline. Using the build pipeline from the previous section to present feedback we could enhance it by ensuring it is automated. Each unique stage will thus be configured to start a following stage, once it is completed.

With this solution we would have a complete automated procedure for generating feedback on a developers implementations. Any breakages in the build of failures of unit tests will be immediately reported and the functionally can be corrected. The more time consuming activities, which might have a lower priority, would also be automatically performed on our integration environment and no longer put constraints on a developer's workstation.

Push Notification. A very simple solution to automate the start of a process is to make the repository's location transmit a notification when a change has been received. If we would be using our integration server we would send a small information package telling it that now would be a perfect time to see if any changes has been integrated.

The primary benefit of this approach is that the execution of our activities would occur almost immediately after a check-in. Otherwise we would have to either start the execution ourselves or make use of other approaches, e.g. polling. With polling, we check for changes in the repository on a chosen time interval.

Since we have already established in the previous section that we have chosen a build pipeline as our solution to feedback, we also chose the solution to automate it. As we mentioned, its very concept makes it simple to automate. In figure 5.3 we show a diagram depicting the automated build pipeline with push notification.

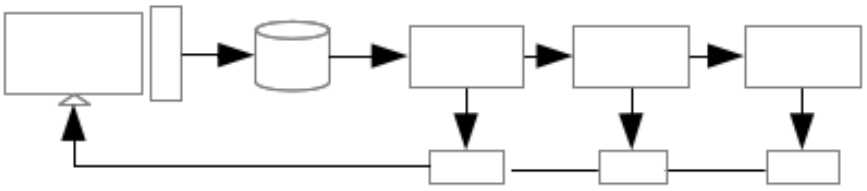

Figure 5.3: Automated build pipeline with push notification.

We will also make use of the last discussed solution, the notification. This is because of the fact that it is very simple to set up and manage and it will trigger a new execution of a build pipeline almost immediately after a change has been checked in to the repository.

5.2 Development Environments

In chapter 2 we discussed our preliminary analysis where it was discovered that certain project types had some platform dependencies. Now that we also have our common process in place (see previous section) we can move on to discussing the pros and cons of different solutions within this area of development environments, as having only a single development environment proved impossible, and we will argue for why that is.

A.RP1 By having two copies of the source the double maintenance problem is introduced

A.RP3 Proof in test environment that the application works does not prove that it works in production environment

A.RP4 Since a build is performed at the production environment after a deployment, there is no guarantee that it will work and failures are caught later rather than sooner

B.RP4 A change can introduce undetected failures in production environment

RC1 No company standard or templates for a development process.

RC2 Feedback on implementation is only given at the final stages of a project.

RC3 Repetitive procedures are performed manually.

R2 The solution needs to support working with multiple environments and be aware of their existence.

R3 We require a centralized node where we can check out or clone from a repository, perform builds, execute tests, i.e. perform the same activities as are done during development.

The main goal here is to cover the requirement about multiple platform support and sort out what our set limitations are. For example, if we have .NET development, which we have, we need to have a Windows platform since there only exists official support for this platform. Such a limitation is non-negotiable, but we are still free to experiment with the setup, e.g. if we see value in having a Linux master which controls a Windows slave. The solutions discussed below are to take the above stated problems, root causes and requirements into consideration.

Optimal Environment Platform. The operating systems or platforms that are best suited for the given project cases. Here, the availability or the quality of developer tools such as compilers and test frameworks mostly dictate what is the optimal environment platform as not all of these tools are cross-platform.

From the development carried out at the company we saw both cross-platform (Android) and platform-specific development (ASP.NET, iPhone). As cross-platform development literally means that developer tools are made available for at least the major operating systems this did not set any restrictions, but of course the ASP.NET and iPhone development did. Developer tools for ASP.NET are only available for Windows, and iPhone tools are only available for the OS X operating system.

So to support these two development types we needed to have a Windows and an OS X platform. We looked into having a third Linux machine, but found that since Android development was supposed to work well under Windows or OS X the addition of a Linux machine would add little but more maintenance. A Linux server would not cost anything in terms of license, but our intention was to keep our solution as minimized and contained as possible, i.e. holding down the complexity and size of the solution.

Master/Slave. The master/slave approach suggests to have one of the server platforms as the master. This would be the centralized node that developers interact with and store their configurations on. Should there be need of another development platform this would be hooked up as a slave to the master so that the slave server can be controlled via the master. As we can see in figure 5.4, developers workstations only communicates with the repository and the master. The slave then extracts the software from the repository and sends results from tests back to the developers by going through the master.

This topic relates to the just previously discussed topic of the optimal environment platform. We discussed the need of two platforms if we would support both ASP.NET and iPhone development. As we have such a need the master/slave idea can be discussed as a way to support two or more platform environments with less hassle as we only directly interact with the master, and the master interacts with potential slaves.

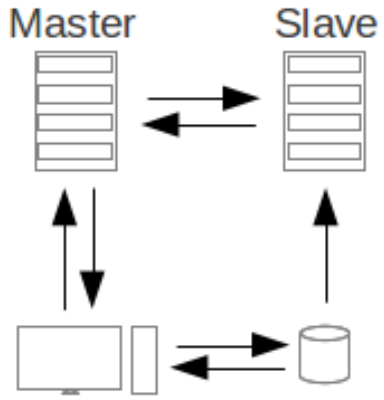

Figure 5.4: Master/slave solution.

Of course, there are obvious downsides to the idea of master/slave. For example, we have more platforms to manage and increased storage requirements. Should we, as in our case, already be dependent on having multiple platforms we have already paid the costs and we can enjoy the benefits of having one centralized node, rather than having a number of equal servers.

Virtual Environments. A virtual machine is a "fictive" computer, emulating the architecture of a physical computer through software. A virtual environment would be simulating a network topology consisting of one or more virtual machines on the same physical computer [HF10, chap. 11].

The advantages we found with using a virtual environment for our solution would be that maintenance would be easier via a so called virtual machine monitor than several physical machines hooked up to their own monitors. Another advantage of using a virtual environment is that we can create and use baselines of our environment, i.e. save the states of our virtual machines and version-control these binary files in some manner.

Among the disadvantages of employing a virtual environment we have the economical aspect. When is it worth it to use a virtual environment? If the need is to have one server perhaps buying a large and expensive machine, capable of running several instances of virtual machines is not very feasible. Also, we have the resource allocation problem: what would be sufficient in terms of CPU, RAM and drive space in order to have a working solution?

For our solution we decided to employ a Windows platform as this would give us the opportunity of implementing support for both ASP.NET and Android projects which were our primary targets. We also wanted to leave our options open as to use the master/slave approach with the Windows platform as the master and an OS X server as slave if future support for iPhone development would become realistic. We liked the idea of the master/slave approach as this supported our requirement R3 for a centralized node, but for only ASP.NET and Android we were not in need of another platform environment than Windows.

Finally, we chose to use a virtual environment and having the Windows platform as a virtual machine within this environment. Actually, this choice was partly because the company already had a virtual environment in place where they kept a lot of their servers and we were provided with easy access through remote desktop. Weighing in on this choice was also that we could create and store images of our environment setup for future use.

5.3 Branching Strategies

This section will discuss more than one solution with branching. To have a mature and improved process where we can be certain that are product is tested and releasable we can make use of different branching strategies. We will discuss the couple of strategies that we experimented with and worked to identify their strengths and weaknesses.

Stable Branch. This branching strategy, learned during an event held by company Praqma in Copenhagen, works by having one development main branch (the integration branch) towards which developers carry out and integrate their daily work. Quality in this branch can vary, e.g. unit test coverage and code metrics. Then we have stable branch where quality is of the essence and only "stable" code, i.e. only code of a certain quality is allowed into this branch. Developers are not suggested to have commit access to this branch, as this should be managed via the environment when quality has reached the desired level in the integration/unstable branch and then pull the changes from there. The stable branch is essentially a copy of the development branch with all the lesser quality revisions removed—a "picky" copy.

The goal is to be certain that anything in the stable branch is to be considered as releasable, which will in turn shorten the release process and minimize the risk of having a broken application. As an added benefit, this branching strategy integrates well with the practice of continuous integration. See figure 5.5 for a visual representation of the stable branch strategy.

Among the disadvantages of this strategy is of course the extended maintenance of branches as we with this approach have two branches, and depending on the implementation of the version control system, the size of the repository can become much larger. This strategy is also supposed to enforce quality improvement, which in our opinion should be viewed as an advantage, but more time would be spent on for example tests. We can also see the risk of rogue developers by-passing the strategy if they feel it limits them.

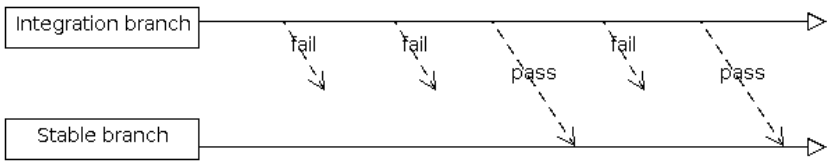

Figure 5.5: The stable branch strategy.

Personal Branch. The name of this branching strategy is self-explanatory as developers can only commit to their own personal branch, but can update themselves against other developers or the mainline should they want to. After having checked-in to the personal branch the environment should be able to try to integrate the personal changes into the mainline should the code pass compilation, tests and code metrics.

This strategy is similar to the stable branch strategy in that we have a known working mainline branch. The addition here is that developers have the possibility to receive feedback only their own working copy, since they are in their own branches. The only alterations in how the developers normally work are that they are now free to experiment and exchange untested changes without affecting the working system.

5.4 Distributed Development

As part of solutions on higher levels of maturity, we chose to study the possibilities of supporting distributed development. In ASP.NET we had developers arriving from another office to assist in the project, so this section will only focus on designing a solution that would have made it possible for the developers to stay in their offices and still being able to work with a common process.

First of all we will assume that we have some sort of network connection between offices and that the common process is somewhat known throughout the company. We do also need to expect that we have decentralized repository, meaning that we have to be prepared for one main repository in one office containing the whole application and smaller repositories at the other offices with parts of the application.

Distributed Build Server. If we would combine our solution with an integration server together with our design for a master/slave approach we can have one build server in one office acting as the central or master and install build servers on the other offices as slaves. The master is where we configure the activities and the slaves perform them. In figure 5.6 one developer communicates to the main site via a build server, which orders a local build server to perform whatever activities it is configured to execute.

The advantages of this solution is that we have assurance that company standards are followed on every site and that knowledge is not limited to only one office. We would also not put any constraints in the time it takes to receive feedback, because activities are performed on the same site as the developers through the slaves. Drawbacks are the cost in educating more personnel in configuring build servers and the fact that we will have more servers that require maintenance.

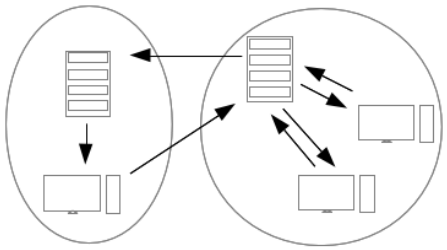

Figure 5.6: The distributed build server approach.

Multiple Build Servers. An alternative the previous suggested solution would be to let each site have its own implementation of the common process. They would each have an integration environment, which only contains the activities that are relevant and performed on only that site.

We would keep our cost in more maintenance but possibly removing the extra required education because each site would use a type of build server they are familiar with. Ultimately, our dream of a common company process could shatter since this strategy contains the risk of introducing divergence in the development processes between sites.

Since our solutions are based on that support for distributed development is on higher levels of maturity, the costs for integrating it into the company would not be very large. This is because we have an efficient and polished common process that is abstract enough so that it is simple to apply a solution that would make it possible for development teams not to be in the same office.

5.5 Survival of the Fittest

Because of the fact that we prioritized to study two project cases and perform a complete analysis in at least one of the cases we could not aim to implement all of our designs. We also have to consider the opportunity presented by the domain. In our domain we found no evidence of a common process and automation was rather scare, which leads to the conclusion that we first need to focus on a suited development process.

We will therefore implement solutions, in the next chapter, that aim to establish a common development process with automated activities. They will be performed on an integration server together with an automated build pipeline. These solutions suit our domain since they are simple, configurable by developers and require little maintenance. We will also be using push notifications to trigger pipelines.

Since we will be implementing an integration server we also need to consider solutions on development environments, i.e. where it shall be located. We will use one Windows platform, because it is required by one of the cases. As we have mentioned, the company already has a virtual machine monitor installed and because of that our integration server can be placed on a virtual environment without any extra costs in hardware. It also means we can baseline our implementations.

The designed solutions that we will implement into the two cases in the next chapter.

- Integration server
- Automated build pipeline
- Push notification
- Windows platform on a virtual environment

Chapter 6

Implementation and Measurements

With the designs from the previous chapter we will move forward with a discussion on how and why they were implemented. This chapter will not put too much detail in how we implemented our solution nor on variations to a general case. This is because we assume that anyone capable of reading and understanding this report will also be capable of performing their own implementation based on our specification and designs.

Instead we will focus on difficulties and challenges that we encountered during our implementation. We consider this much more important to place focus on because such factors would increase the cost for a company, should they implement our solutions or alternatives. With knowledge of obstacles beforehand, we can avoid or overcome these challenges.

The implementations for each case will first state the solutions that we intend to implement. Next we briefly motivate our choice of implementation and state any methods we took during implementation. Any challenges we encountered are presented during the discussion. Finally we present a summary of the implementation that compares what we have now with what we had before.

The final section of this chapter will summarize our implementations and present measurements that we performed on the results of the implementation in the ASP.NET case. We will discuss a few initial conclusions of our results as well as the values from the measurements.

6.1 Implementations

As our goal was to improve the development process in the company and not only the two cases we analyzed, we will begin to discuss an overall solution. This solution is based on our designs from the previous chapter and even more so on the domain and contexts detailed in chapter 2. From now on, this architecture shall be considered a view of the maturity level we wished the development process to reach and its purpose is to display how developers interact with the improved process and the connections between different parts of the new system.

In chapter 4 we discussed that one of the basic principles of continuous delivery is to always have a releasable product. We want to apply this principle to the development process of the company as a means to preserve quality. As can be seen in figure 6.1 the developers communicate with the new architecture through the repository, which is the same procedure as they use today.

Since we are dealing with team sizes between two to five we will not have much traffic between our integration environment and the repository, thus limiting the load on the network. Notifications from the location of the repository will also assist in that regard. If our number of developers

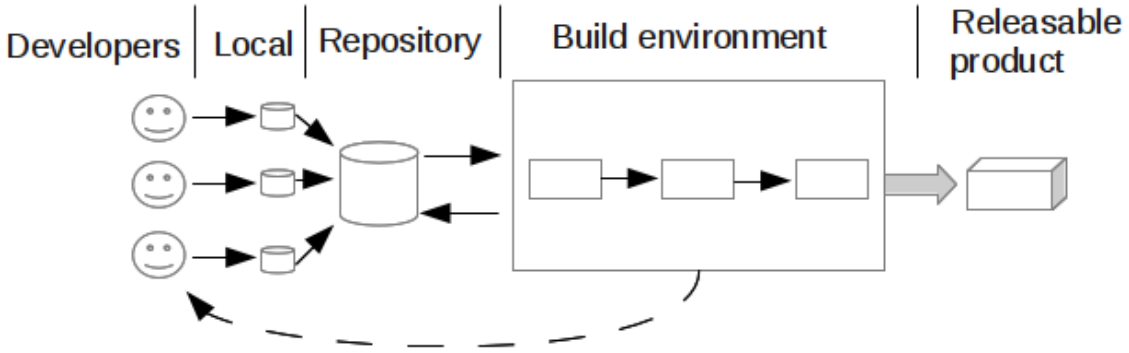

Figure 6.1: Sketch of the overall architecture for the new development process.

would be much larger, e.g. one hundred, we would have to come up with a solution that could split the team into smaller groups and apply the solution on each part.

We also need to consider the available resources on the company, as well as the type of conducted projects. The solution presented in figure 6.1 is very low in cost. The only addition will be an environment capable of executing the development activities and the purchase of new software, but only if a commercial solution is chosen. The additional costs in time would be in increased maintenance, which would be negligible in regards to the benefits we aim to produce.

If we look at the types of projects developed at the company they are mostly, as of now, various applications to cell phones. What is most important to notice is that they are not large or highly complex systems as e.g. a trading system. This means that we can ensure that the integration environment will be able to build and test the software as well as produce results for feedback in an acceptable timespan.

Before moving on with our implementations in the two studied cases, we would like to discuss some of the technical solutions that we chose. As software for the integration environment we chose Jenkins, which has a wide variety of plug-ins that would suite our needs and we also had some familiarity with the tool. The content of our toolbox that was common in both cases are presented in the following list.

Git Plugin. One of the plug-ins in Jenkins. It is currently not installed by default and thus it requires a manual installation. It allows Jenkins to communicate with the Git installation on the same machine and we needed this tool because the company used Git for version control.

Build Pipeline Plugin. Jenkins has a plug-in which allows a user to crate a view which becomes a graphically implementation of a build pipeline with the same principles as we have discussed in chapters 4 and 5.

Post-receive hook. To enable the transmission of a notification once a change has been integrated into the repository we utilized a script which is commonly referred to as a Git hook. They are placed in a specific folder in the repository and will, in this case, triggers after a change has been received. In our script, we made it send out a notification to the Jenkins installation and telling it to update from the repository.

VMware Server. The integration and build environment we implemented was placed on a virtual machine server as a virtual environment. The software was already installed and used on the company and we saw no reasons to introduce another.

The following two subsections will present a more detailed description of an implementation, which is highly specific to that particular case. They will follow the template that was briefly mentioned

in the introduction to this chapter and, again, place the most focus on detailing any encountered obstacles. Now would be a perfect time to take another look at our architecture in figure 3.2 and commit it to memory.

6.1.1 ASP.NET Website Project

We will start by detailing the ASP.NET implementation, with focus on the problems we faced performing this implementation. It should be clear that the implementation discussed is to be considered specific for the project case and domain we had, but should still be of value for the reader when wanting to see how a process can be improved based on the designed solutions and requirements.

As we discovered in our preliminary analysis back in chapter 2, the developers in this targeted case performed their activities on their own workstations. They had different kinds of tests, which were often only executed by the author. The first step we took was to implement an integration server, the first solution back in section 5.5.

Since the goal of this server was to perform all development activities we installed Jenkins, Git and Visual Studio (which included the .NET framework) within a Windows environment. What we never became quite clear on was how much of the .NET framework that was required to successfully support the project. The developers has the latest installation of Visual Studio and we chose to install the exact same version so that we could be sure that nothing would be missing, i.e. we wanted our environment to mimic a developers workstation.

At the time of our implementation, Jenkins had no out-of-the-box support (e.g. through plug-ins) for ASP.NET development and deployment. This required us to to make some research on which tools that were used by Visual Studio when compiling, testing, etc. The focus on deployment within the ASP.NET project was fairly easy to take care of within Jenkins as there existed some general plugins for this purpose. These tools or applications we used are as follows:

ASP.NET Compilation Tool (`Aspnet_compiler.exe`). To be able to compile the application in order to see if it actually could be compiled we needed to use an ASP.NET compilation tool bundled with the .NET framework. This was a command-line tool and was suggested to use in an article [Mit09] at the official ASP.NET website.

MSBuild. The suggested stand-alone test tool for ASP.NET WSP projects.

Artifact Deployer. As deployment was an important part of this project we found that this plug-in for Jenkins helped us a lot so that we did not need to do any scripting for the deployment process; only for the compilation and testing.

We are especially content with our ASP.NET implementation of compilation and testing as the literature on this in combination with Jenkins was scarce. The tools for the framework is commonly delivered together with the IDE Visual Studio, but they can still be used as stand-alone command-line applications. As Jenkins supported writing Windows or batch scripts it was fairly easy to use them via the guidance given at [Mit09]. Feeding the project file to the compiler via command-line returned a pass or a fail within Jenkins.

The next step we took was to make the three activities (compile, test and deployment) automated. The primary focus was still to implement a common process, but since Jenkins has almost a straightforward approach to automation, we saw no reason not to try. By using the solutions of a build pipeline together with a post-receive hook, which we discussed earlier, the architecture became very similar to our overall solution.

We called for a demonstration meeting together with the developers of the ASP project together with our supervisor. At this demonstration we made a somewhat improvised and open walkthrough on Jenkins, the first build job, the concept of a build pipeline and how we could implement

automatic deployment. The developers seemed to like our approach with the demo as it was very hands-on and they could ask questions whenever, which we wrote down as feedback.

When we felt confident enough that our experiment solution did what it was supposed to or we had enough "know-how" to set it up properly we inquired about getting a virtual machine on the company-wide VMware server. We quickly noticed a significant drop in performance on this new environment. The installation process took its time and when even Jenkins was under-performing we had to ask for more resources.

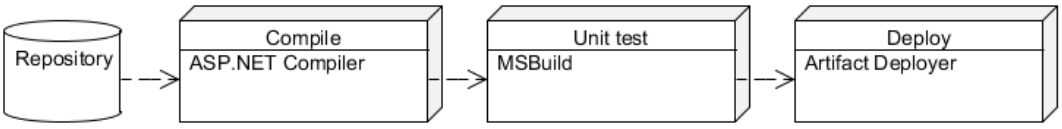

Figure 6.2: A visual representation of our ASP.NET implementation using the deployment pipeline concept.

During the implementation of the automatic deployment to the production-like environment we ran into some unforeseen problems due to network security. The issue originated from the user on our integration environment that was running the Jenkins process. The user did not have sufficient write access to change files on the production-like environment. This is something that has to be considered if our environment are on a secured network domain. See figure 6.2 for a diagram depicting our implemented solution utilizing the build pipeline.

6.1.2 Android Project

In the same manner as with the previous section on the ASP.NET implementation, we will here document our Android implementation. Again, this will not be the thoroughest of documentations, because our intention is to present and argue for our set of solutions and to "raise some awareness" for what we found difficult with the Android implementation rather than give the full tutorial.

Since we already had our build environment in place we chose to stick with it (Jenkins) in this instance too. Since Android development is cross-platform the chosen Windows platform was not supposed to limit our implementation.

We began our implementation by verifying that the Android project was complete and reproducible. If this was not the case, we would have us a problem since we had defined the requirement R3 that we would need a centralized server which could perform a check-out and build the project from scratch. The Android project repository was far from complete and this was due to that the only developer of this application had dependencies to local libraries in his project, together with other workspace-specific files.

We used the Android Eclipse bundle (which included the Android SDK) when we tried to get the application to build, but in order to build the Android project we needed to have the dependent library binary which was only available as source code. To produce this binary and then make the necessary configurations needed for the project proved to be time-consuming.

We had the Android developer show us how he set up the project from scratch so that we could grasp the configuration and compare with the Android support pages. For example, for Android development it was suggested to have three separate projects for application, tests and libraries, respectively. This choice by the developer seemed strange at first, but as it was the suggested template changing this would maybe have had some unforeseen consequences. However, the fact that the third project, the libraries project, was not available in the Git repository was a real

break of the R3 requirement and in extension something we would have to take care of should Jenkins be able to perform check-out and build the project.

So in our copy of the repository we created a third library project, bringing reproducibility into the application. We also decided to merge the two existent repositories (application and test) and have all three projects in one repository to ease the complexity of the check-out performed by Jenkins.

As the build script for Android we used Ant, as this was the officially recommended software for Android builds. Jenkins supports Ant natively and after having configured Jenkins to have access to the Android API we could set up a pipeline in a similar manner to the ASP.NET case with commit, unit test and deploy stages.

One aspect of the Android development that raised some concern for us, but that we were not able to pursue because our priority was how to efficiently handle deployments. As Android applications can only execute in an Android setting (including unit tests dependent on the Android API) the available choices were to deploy to an emulated environment or to real Android hardware, we were afraid that the long deploy times could limit the quick feedback requirement (R5) that we set in section 3.2, which says that feedback should be generated in under a minute.

As we had to prioritize what we would have time to do in our limited time scope, a basic version of this Android implementation was never released, although it was more or less a complete, basic version. Therefor we have no measurements to present in section 6.2.2 as we will have for the ASP.NET project.

6.2 Summary

To lay the foundation of the next chapter, where we reflect our work, we will in this section first of all briefly summarizes the results from the implementations in our two specific cases. After that we will present and argue our measurements we performed on the ASP.NET case and discuss their outcome and value.

6.2.1 Initial Conclusions

The primary difference in our implementations was that we required specific plug-ins to build and test each case. This is, of course, to be expected. What we see as important is that the basic principles from the overall solution are successfully applied or else it could not be a common development process and we have only developed a new type of ad hoc process. The matter of validating our work is the topic of the next chapter.

We can still draw some initial conclusions in this stage. In both cases a developer could check in a change to the repository, which would trigger a notification to a build environment and kick off a process that would start by updating from the repository and last a trip through the stages in a build pipeline. All of the regular activities that the developers performed during a work day was contained in a single environment and performed in an automated fashion.

If our overall solution would be applied to the iPhone case we hypothesize that the greatest challenge would be in the communication between two environments. If we recall our previous discussions of this case, it requires a certain platform for performing activities such as building or testing, similar to the ASP.NET case. Our design of a master/slave solution in section 5.2 might be helpful in this situation.

6.2.2 Measurements

We we will now analyze three types of measurements that we performed on the ASP.NET case, before and after our implementation. As we have mentioned, we did not perform any measurements on the Android case because we prioritized a complete study on at least one project case. Thanks to the implemented feedback and the logs from the version control tool, we were able to perform most of the measurements ourselves, while a few required interviews with the developers. Our aim with the measurements was to answer three questions.

The first question we sought an answer to was if there had been any changes in the time it takes to perform certain activities. We wanted to explore any potential gained benefits or additional costs of our implementation. In other words, our first question was: is the implementation efficient?

We chose to measure all of the activities that the developers performed daily, excluding testing because the same tool was used to execute the tests. The results can be seen it table 6.1. Effort is the time spent by a developer and duration is the effort plus the time it takes for any tools to perform the activity.

	Before		After	
Description	Duration	Effort	Duration	Effort
Deployment	2	1	1/6	0
File copy	No change			
Build	No change			
From push to feedback	10	5	1	0

Table 6.1: Time measurements in minutes.

As we can see in table 6.1 we have either gains or no changes at all and thus no additional costs. The gains may not be much to look at, but if we imagine that e.g. we perform five check-ins each day, the total gains during a whole project would be a greater number. By automating activities in this particular case we increase our efficiency by 10%. The "before" measurements were collected through interviews with the developers, while the "after" measurements were calculated by taking a rough mean value of a few of the time measurements recorded in the implemented build server.

For the second type of measurements we wanted an answer to what the effects our implementation had on the quality of the development process. These are measurements that have a connection to the activities that are performed in the development process, but cannot be measured in any particular unit. As we have discussed earlier, to reach higher levels of maturity in our domain we require a process that can present the quality of our work.

If we take our unit tests as an example, they warn us of errors we have inserted into the software, but that requires that they are executed preferably after any change. Should a failure be caught four or five days after it was integrated we would have to remember what we did four or five days ago. If it is not our own error we might have to go through four or five days of history to discover when it was inserted.

What we have in table 6.2 is thus a focus on how are implementation has effected testing in the development process. We have also chosen to include two other measurements. The first one is if our implementation introduced a more or less complex process to configure, which is mostly related to our integration environment and its software. The second additional measurement tells us if constant feedback has had any impact on the success of builds.

As was expected we can see in table 6.2 that the intensity of test execution has increased, since they are performed at each integration and thus failures are also caught no later than after they were integrated. According to the developers themselves, they did not find our implementation

Description	Before (Level)	After (Level)
Test execution intensity	Low	High
Failures caught	Unscheduled occasions	At each integration
Configuration complexity	Low	Minor increase
Successful build rate	High	High

Table 6.2: Measurements on development impact.

complex and they were able to make changes to it. It is still a bit more complex than it was before because of the nature of our build server, Jenkins.

Our third and final question was related to any changes in the division of labor. Besides implementing a process with the aims of reaching higher levels of maturity we also wanted a more efficient solution. We wished to study if more time could be spent on development, thus increasing the efficiency in the team, and also to analyze if the implemented solutions gave any change in patterns we detected during our preliminary analysis in section 2.3.

Description	Before (Amount)	After (Amount)
Merge conflicts	Few	Fewer
Reworks per day	Few	Few
Commit intensity	Normal	Higher
Number of timeouts	Few	Few
Script management	Deploy script	None

Table 6.3: Measurements on labor division.

What we can derive from table 6.3 is that now that the activities are automatically performed on a unique environment, the developers do no longer see a check-in to the repository as an obstacle. Now that the tedious activities no longer have to be performed manually, the developers feel more free to integrate their changes more often then before our implementation and thus the increments are smaller.

Chapter 7

Evaluation

In this chapter we will first validate the implementations discussed in chapter 6. By implementing the solutions discussed and fitted in chapter 5, did we make a positive impact on the software development maturity with regards to the requirements presented in section 3.2?

In the section following the validation we will combine our implemented solution with the notion of software development maturity. We will shortly arrive at our suggested *Software Development Maturity Model* comprised of four maturity levels. We will compare it to the well-known *Capability Maturity Model* [PCCW93, PWCC95] (CMM) and a maturity model entirely based on continuous delivery created by the Danish company Praqma [Pra13].

Last, but not least, we will make some comments on what parts of our work where future work could be placed. This include areas we did not prioritize or was considered out of scope for our thesis.

7.1 Validating Implementations

The purpose of our validation is to ensure that we have met the goals we had on our work and if we have covered requirements that we placed on solutions in chapter 3. The primary goal and purpose with our master's thesis, discussed all the way back in chapter 1, was to improve key areas in the development process. It was supposed to result in a general solution that could be used for the entire company.

7.1.1 Goals

The key areas in both cases turned out to be how to handle deployment and to capture failures as early as possible. In both implementations a build pipeline is used to perform development activities, including testing and deployment, in an automated fashion. An integration environment is notified after a check-in, which triggers a new trip through the pipeline and its every stage has the capability to report the results.

Every case would of course have a slightly different implementation, especially if we are developing for different platforms. What is important is that the concept of the build pipeline, by separating development activities into unique stages, together with an integration environment, gives us the possibility to apply our solution to many different project cases. We have seen that it allows us to prioritize activities and we can configure our environment to present feedback on our development, which will assist in keeping our product releasable and thus shorten a release process.

The third and final goal, discussed in chapter 1, of our thesis was that our improved development process should raise the development maturity. Thus the ultimate goal of our solutions was to create a general, company-wide solution which reaches for a higher level of maturity by improving key areas in development.

- Develop a general development process
- Reach a new maturity level
- Improve key areas of development.

If we look more thoroughly into the solutions we implemented they were based upon five designs from chapter 5. We had chosen an integration environment that was aiming to utilize a build pipeline to perform development activities. A new pipeline was to be triggered by a notification from the location of the repository. The environment itself should be a virtual instance. In figure 7.1 we can see the connection between the goals of our thesis and the solutions we implemented.

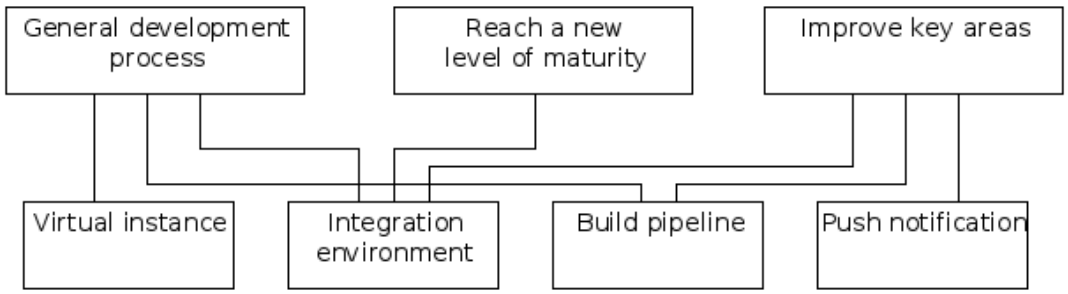

Figure 7.1: The goals each solution fulfills.

With the virtual instance we have the capability of saving and storing a state of the environment it is running, which can later be reused or copied on to another server. The integration environment supports all of our goals because it performs all development activities, provides feedback and is an end result of a focused attention for improving a software process.

The concept of the build pipeline is, in itself, an abstract solution to manage and visualize development activities. It will assist in improving our key areas by performing the prioritized activities as initial stages. Finally our push notification will lower the cost in time by making an automated process begin almost immediately after a check-in.

We have now gone through a validation against the goals we placed on our work. Next we will validate our implementations against the cross-cutting requirements we derived in chapter 3.

7.1.2 Requirements

If we start with ASP.NET project we did have a centralized node capable of building, testing and deploying on an average under 60 seconds, it was automated and triggered by a change in the repository and it communicated with both development and test environments. We had to do a follow-up interview with the two developers to validate the requirement about keeping a low complexity.

They found it simple to change the technical solutions we had created and they recognized their own process in its new form. They had been very frustrated with how deployments were carried out in their project and were very happy to have it automated.

A validation of the implementation of the Android case reveals that all but two requirements are covered. Both of them were the two background requirements. The first was that time it took to get feedback far exceeded 60 seconds, because each run deployed to either actual hardware or a simulation—the latter taking even longer.

What is required in this case could be to split the deployments in two parts. The purpose of the first, minor, deployment would be to only perform the highest prioritized tests to lower the cost in time of a deployment. The second type, which would be larger and take more time, would perform a complete test procedure of the software.

The second requirement that could not be validated against in the Android case was the requirement we validated in the ASP.NET case through interviews. It is requirement R1 from chapter 3, that states that we cannot have a too complex solution. We did not have any interviews with developers in the Android case because we prioritized to have a complete study on at least one project case.

With an interview we could have had the opportunity to receive feedback on our implementation, as we got from the ASP.NET developers. The foremost important factor has been to discover if the developer felt that the implemented solution was too complex to work with and would take them more time to adjust to the change than the benefits from the new process. This does not mean that our implementation was a complete waste of time, because we were still able to experiment with our designed solutions in more than one case.

The integration environment was the most important solution because it serves as centralized node capable of performing our development activities and can communicate with multiple environments. In figure 7.2 we have the complete connection between our solutions and four of our requirements. Actually the integration environment also has support for automation, but not without any additional solution, such as our build pipeline.

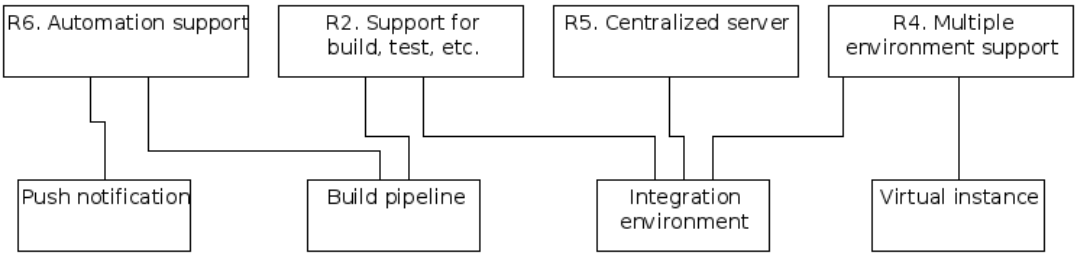

Figure 7.2: The coverage of the requirements by the solutions.

In the previous section we also saw that the solution of an integration environment covered all goals of the thesis. That solution will thus solve many of the problems that we discovered during our analysis. A build pipeline will extend the integration environment by adding support for automating our activities and presenting visual representation of our development process. The solutions of push notification and virtual instance can be considered as lower prioritized solutions.

7.1.3 In Hindsight

As we discussed in both implementations in chapter 6, we ran into some unforeseen difficulties. In the ASP.NET case we had problems with installing some of the required tools and issues with network access. The obvious question is now if our problems could have been avoided or if we could have managed them with a better approach.

The issues with access rights might have been discovered with a more thorough interview with the network administrator. If we had explained our plans to set up a new environment that required access to other environments within the network, we might have received information that it requires a configuration by the administrator.

When it comes to the issues with the tools for the ASP.NET case, the required knowledge we received was through our research. It might have been possible that someone in the company possessed this knowledge and could have assisted in our initial setup. If we had reached out to every developer, we might have been able to prevent the issues we had. When improving the development process on an entire company, we should remember to make use of every bit of competence we have in our disposal.

We did not have the same amount of issues in the Android case. The documentation on the Android website provided sufficient information on the standard conventions of an Android project. Since our issues were mostly tool-related, we again could have shorten our time for a setup by asking more developers on the company and not only those in the project we analyzed. The research we instead chose to conduct should not be considered a waste of time, because it gave us a deeper understanding of the management of the two project cases.

7.2 Putting the Pieces Together

With a complete validation of our implementations performed in the previous section we will now evaluate all of the discoveries made in our analysis, designs and implementations. What we signify as a discovery in this context is a single fact that poses a restriction or a condition that has to be met before something else can be put into place. We will use these facts and conditions to derive maturity goals for our type of domain.

The maturity goals are our guidelines to reach higher levels of maturity, but these also require that we have defined maturity levels. The following four sections will each define a unique level of maturity by stating specific problems we got from our analysis in chapter 3. They will then repeat all findings that were discovered for that particular problem in a discussion that will accept or reject them with motivations on domain, requirements and cost against benefit.

During our discussions we will also compare our findings and results against two slightly different maturity models mentioned in the introduction for this chapter. The first one is the *Capability Maturity Model* (CMM) and the second is called *Continuous Delivery Maturity Model* (CDMM). The structure of the levels in the former model is similar to our own, while the latter has novice, beginner, advanced and expert levels on four types of areas: build, test and QA, source code management and, finally, visibility.

The two final maturity levels will have a lack of findings because we did not have the opportunity to experiment with higher levels of maturity. The basis for the discussions will instead be from what we gathered in our literature and the experience we have after all our work. Further studies are required to gain more scientific facts on those two maturity levels, but that is the topic of section 7.3.

7.2.1 Common Process

In our early stages of our preliminary analysis in chapter 2 we determined that our project contexts did not contain any common or company standard development process. We have seen the existence of a tendency to introduce a double maintenance problem, the resulting organization of a repository when only a single developer is assigned to a project and irregular testing of software.

In the first level of maturity we will look into will address the absence of a common process. We have seen the benefits of automating activities, but before we focus on this we require an established development process. We would want to erase the unwanted extra cost when a developer is taking over or joins other developer's projects, as was the case in the Android project.

What we should remember is that on this level we place our primary focus on implementing a common process. It does not mean that, e.g. automation is not allowed. It is in fact encouraged to experiment with automation and quality control already at this level, but it is not the focus. However, if we in the future work to reach a higher level of maturity, a common process suited for the domain has to be implemented or else we risk falling back to ad hoc procedures. This is what occurred on the company with the previous build environment.

In the beginner stage of the CDMM we have an integration server that is capable of performing development activities, is triggered by a check-in to the repository and results of a build is available to stakeholders. We have seen in both of our implementations that by keeping increments small, we can receive fast feedback through an integration server. That, however was not entirely capable before testing became automated.

Because of the fact that increments only got smaller once we had established a common process, together with an automated integration environment, it is still our primary focus on this maturity level. We do not consider making our feedback available to stakeholders as walking along the wrong path, but it is not something we place any priority on at this moment.

In the CDMM we also have code metrics as part of the beginner stage and the CMM has quality assurance on its first level. We see this as part of quality control and not something that should be a target before a common development process is available. Again, there is nothing wrong with experimenting, but that should only be to further test the capabilities of an ongoing process improvement. We do not want the risk that every project has its own specific process.

We have to remember that this maturity level is the first stage after ad hoc behavior to deciding on approaches in a software project. It means that we should not expect that every single developer is an expert in working with well improved software processes. Thus placing too much focus on quality, testing or automation leads to, what we discovered in our analysis, evaluating and deciding on "the best tool for the job".

If we instead first solve our root causes and implement those solutions into a new and common process we will also gain a deeper understanding of the capabilities in the company, the performance of different approaches and the pitfalls of having an ad hoc process. Once we have reached this level of maturity we focus on increasing the efficiency of our common process and the quality of our products.

On the topic of configured tools or scripts, we did discover that storing a configuration is required to keep a common process in a specific project. This also requires that the script or tool is configurable for more than one case, e.g. the build script in the Android case. If not, then we cannot ensure that additional members in a team will be able to use them.

By having a dedicated environment which performed all of the development activities and as a central headquarters, so to speak, for the common process forced the structure and organization of a repository to allow reproducibility for the application. It became almost identical to always have another member in a project that should have the capability to clone the repository to a workspace and build, test, etc. without having to e.g. download dependent libraries.

The first maturity level will thus target the activities performed in a development process and we work to establish a common process. To reach this maturity level we have to ensure that our integrations in the repository are always made through small increments, be in possession of an environment that is capable of performing our development activities and share any configurations we have in our tools or scripts amongst team members.

The maturity goals for the first level of maturity.

- Keep increments small
- Dedicating an environment for development activities
- Store configurations for repetitive processes

7.2.2 Automation

In this section subsection we will focus on the problem of automation. As we have discussed in previous chapters, a common problem was that repetitive procedures were performed manually. This was the reason our requirements from chapter 3 included support for automation in a solution and why we chose to experiment with automating a process in our cases in chapter 6, once we had established a common process.

Automation is an important part when improving a software development process, namely because it eliminates repetitive behavior which costs money in the long run, it reduces manual errors and we do not think anyone will argue that developers like to push a number of buttons in a specific order just for fun.

Knowing these benefits it is perfectly reasonable to wonder why automation is not the first step to process improvement. Our basis for this order is partly based upon our own experience. Take for example the Android project case we studied: could we have implemented e.g. build automation for this project as the first step in improving the software development process?

The answer is no and the reason for this was the lack of a proper file structure and that dependencies were not available in the repository. These reasons of reproducibility relate to the absence of a common process so we believe, with some experience behind our belief, that automation should be the second focus.

After having implemented automation for building, testing and deploying in the build environment for the ASP.NET project case when doing the demonstration for the developers we got feedback on the automation aspect. They were amazed by the speed, or rather the short span of time, from doing a commit to actually having a pre-tested and working deploy running at the test server environment.

Specifically the deployment automation was viewed as beneficial since deploys became more regular compared with before when deploys were more of a rare event. The fact that every deploy after our implementation saved a couple of minutes in time was also appreciated. We see automation of these three activities (build, test and deploy) as the first of the maturity goals at this level.

Furthermore we would like to argue that the publication of results from just previously mentioned activities renders in interest from the developers in keeping the application in a working and tested state within the build server environment. We saw for example that whenever the compile or test automation job failed and produced an error report developers were eager to fix the application within minutes so as to keep the application in a green and passed state.

Finally, we have the maturity goal that automation was easily implemented within a CI environment. This was something that we found during our implementation and a CI environment is not necessarily the way to go when wanting to improve, but we found that such an environment supports automation very well.

So to summarize, at this level the focus lies on automating the activities that make for fast feedback, publication of results for these activities and that a CI build environment such as the one we utilized in our implementation has support for automation. Following this section, we will

discuss the next maturity level, targeting quality control automation for which we will carefully distinguish in relation to the automation described here.

Maturity goals for the second level of maturity:

- Begin automating build, test and deployment activities
- Beneficial to publish results from each activity
- Automation can be implemented easily with a CI environment

7.2.3 Quality Control

This section will target what we call "quality control". In chapter 3 we reviewed the two cases and found that there existed little focus on the monitoring code quality through tools like code metric analysis, the test environment was not exactly deemed production-like and other than some scarce unit testing there were only some manual acceptance tests. These are all topics that relate to quality control, meaning that they all check the quality of the application in some form—be it code, assurance that the application works in the desired environment or test coverage.

Due to our priorities we did not have time to pursue quality control through our experiments during our implementations in chapter 6. The motivation for that quality control would make the third level is therefor based solely on literature. We refer to Humble et al. [HF10, chapter 5] where it is stated that only unit tests seldom are sufficient to know that the application runs as intended. Separate from the commit stage we should have quality control comprising e.g. code metric analysis and acceptance tests in a production-like environment.

Creating an analogy in order to motivate the order, an electrocardiography (EKG) machine continuously monitors the heartbeats of a patient at a hospital and sets off an alarm if the heartbeats stop or become irregular, i.e. checks if the patient is alive. This is what the automation level in the previous section attempts to do, by continuously, at each integration, checking if the application is alive. When the doctor performs a proper health checkup it is a more time-expensive process and is only worthwhile if the patient is alive, but it gives us a status on the health of the patient. This can be related to quality control in that we get a detailed view on the health of the application, but it is only purposeful to do this more time-consuming health check if we know that the application is actually working.

If we are to derive the maturity goals at this level we first want to argue for having code quality measurements to run automatically on integration. Since the focus here is quality control making this check automated is a good way to always know the code quality of the application. Comparing with CDMM [Pra13] they had their code quality control at levels 1–2 and not at level 4 which would equal this level. Our rationale for employing these type of measurements in such a high level was that we did not see value in measuring code quality if we did not know that the application was fine and working first. Automatic publication of code metrics is of course dependent on what we have just discussed and is a complement to the previous goal.

Humble et al. [HF10, chapter 5] proposes to have a production-like environment within their acceptance test gate. Of course, such an environment is more of an assurance than a quality check—we already know that the application is working, but will it work in the desired environment? Praqma's CDMM does not mention the need of a production-like environment in their model matrix and nor does the CMM [PCCW93].

Last, but not least, we will argue for the maturity goal to introduce an acceptance test phase at this level. The reason for why we want to have this is also what the research in [HF10, chapter 5] has shown. However, CDMM proposes that automated acceptance tests belongs to the top level,

i.e. the most experienced level. As we already have an automated process in place from previous maturity levels we see acceptance tests as an extension to the automation focus.

The maturity goals we outlined for this level are as follows:

- Implement code quality measurements to run automatically on integration
- Publish code metrics result automatically
- Use a production-like test environment
- Introduce an acceptance test phase

7.2.4 Innovation

Up to this point we have a development process that is applicable for all projects in a company, it is quality-controlled and we have automated all of the tedious, repetitive activities through an integration environment. We have ordered the procedures that test and analyze our software so we can receive fast feedback from our most prioritized activities. The more performance-heavy tests are no longer executed on our own workstation and we can continue with our development while they run behind the curtains.

We do not, however, feel that this is the end of the road and that a process cannot be improved further. The CMM places focus on implementing measurable process improvements at its highest level. This is what we have aimed to implement on all of the preceding maturity levels. This is because we want to ensure that, once a certain level has been reached we should not fall back to a previous and lower level.

At this stage in a process improvement we can consider our development process as very mature and we would instead like to focus on experimental procedures that changes how the developers collaborate. They would not be seen as a risk, because any errors would be reported by our implementation from the previous maturity levels.

In our designs in chapter 5 we discussed branching strategies and possibilities to support distributed development. In the CDMM different types of branching is managed in the earlier maturity levels. We would say it depends on the type of scale on a project. Since our maturity levels and goals are not made for large-scale development, we see no reason as to focus on branching strategies before this maturity level.

It also depends on what type of strategies we are considering. On this level of maturity we are relating to strategies that developers would experiment with to further improve an already well established development process. According to Humble and Farley, we should avoid branching on maintenance and performance tuning, because it leads to irregular merges into the mainline [HF10].

We could also experiment with our management of application binaries, keeping them separated from the rest of our configuration items. If we relate this to our project cases, we had a single packaged file generated for an Android application. This file was deployed to an Android platform were it could be executed. A new build replaced this file and we could therefor not simply retrieve earlier versions of the executable.

What we also have to consider is that the technology to maintain a development process is ever changing. On this level, it is encouraged to experiment with applying new technology to support our development process. One approach would be to implement cloud technology so that the common process and a specific developer's configurations can be accessed on any workstation at the company.

Typical maturity goals that could be found on this maturity level.

- Experiment with branching strategies
- Implementing cloud technology
- Support distributed development
- Version control of artifacts in separate repositories

7.2.5 Software Development Maturity Model

Having discussed our four maturity levels in the preceding sections we will here arrive at our proposed maturity model, which we simply call the *Software Development Maturity Model* (SDMM). Our model has some similarities with CMM [PCCW93, PWCC95] in its design, partly because we based our model on the same principles of maturity goals which can be followed in order to reach a higher maturity level.

Summarizing the maturity levels discussed we have the following focuses:

Level nulla. Ad hoc. This level was not discussed in previous sections as we see it as the most primitive level where development is carried out completely ad hoc, literally translated to "for this" from Latin. What we mean is that everything is done without looking back to previous experiences or templates, i.e. at this level we are reinventing the wheel over and over again.

Level I. Common Process. Through our research we found that having an established common development process was the first step to improving the maturity at a company with a similar domain to the one we experienced. Maturity goals for reaching this level were:

- Keep increments small
- Dedicating an environment for development activities
- Store configurations for repetitive processes

Level II. Automation. The focus for the second maturity level was automation, and even though automation is absolutely not forbidden without an established common process we found through experience that having this common process minimized many of the potential problems during the automation process of e.g. builds, tests and deploys. Maturity goals:

- Begin automating build, test and deployment activities
- Beneficial to publish results from each activity
- Automation can be implemented easily with a CI environment

Level III. Quality control. Due to our priorities this and the next level are only theoretically derived through literature and require some practical testing. We saw quality control as a natural next step when having automated the basic activities to continuously know the overall state of the application (building and unit testing). Maturity goals:

- Implement code quality measurements to run automatically on integration
- Publish code metrics result automatically

- Use a production-like test environment
- Introduce an acceptance test phase

Level IV. Innovation. When arriving at this, the last, maturity level the company should be mature and confident enough to experiment or innovate. From the previous level we have ways of measuring quality and through this we can quickly see if some experiment for the process takes a turn for the worse. Maturity goals:

- Experiment with branching strategies
- Implementing cloud technology
- Support distributed development
- Version control of artifacts in separate repositories

The model, which we can see in figure 7.3, is not the end product of our master's thesis work, but a mere summary of our gathered experiences and knowledge throughout our work. It is based upon the domain we had at the company and with a different domain definition we may have arrived at a different model. For example, if the team sizes are considered large, changing the common process may be very expensive as the work is halted for 100+ developers.

The purpose of the model is to aid companies wanting to improve their software development process. It can be used when planning improvements for the development process, and the goals that are presented can be used to validate against when having implemented a change into the company development process. Although it might not seem like it, it is most definitely an iterative model.

While having touched on this during preceding discussions of our maturity level focuses, we would like to emphasize that even though we have specific focuses for each maturity level we do not in any way forbid for example performing some automation before having established a common development process. It is just that through our experiences and literature we believe the order of maturity level focuses we have presented and argued for will make for a more efficient and less troublesome improvement process.

In the previously discussed summary of the model we also mentioned the *nulla* level, meaning level 0 in Roman numerals. To reiterate we see this as the absolute starting point when discussing software development maturity. That is why there are no maturity goals for this level. At this level we can only improve our maturity.

Parallel to our work Praqma developed their CDMM model [Pra13] which we have reviewed and used in a comparative purpose. A main difference was that their model was split into four categories of activities (build, test and QA, SCM, visibility) and for each category there were five maturity levels ranging from novice to expert. We felt more comfortable with our approach of having a special focus for each level as this approach did not demand a new activity for each of the predefined focuses as in Praqma's model.

One observation we made was that some maturity goals in the CDMM may have been placed at a certain maturity level because of their belonging to a specified category and the fact that there was an empty maturity level box. Compared to our model consisting of 4 maturity levels, Praqma had 5^4 combinations as it was possible to be at one level within one category and another within another category, etc. This was of course an interesting approach detailing what category of activities are in need of most improvement depending on the current level, but we have to look at the bigger picture.

As an example, we can have a situation where we are expert builders but only novice testers. Since most of the categories involves upgrading our technical areas we would have a development

environment with highly efficient tools for building our software, but none of them cannot or are not used to improve our testing of the software. Improving our testing capabilities would then require us to either experiment with the tools at hand or force us to find additional ones. According to our work and experiences, we would instead argue that this is a typical flaw of the development process and focusing on activities instead of iteratively improving the whole process can lead to more rework and higher costs.

Of course we acknowledge the fact that Praqma has got a lot of experience within the field of implementing continuous delivery to various companies. Maybe the differences between our models could be explained by differences in domain: we had one specific domain that we did analyze, while Praqma probably had more hands-on experience with implementing continuous delivery solutions for businesses with differing domains. In short, Praqma's model is more of a practical and experience-based nature, while our model is theoretical.

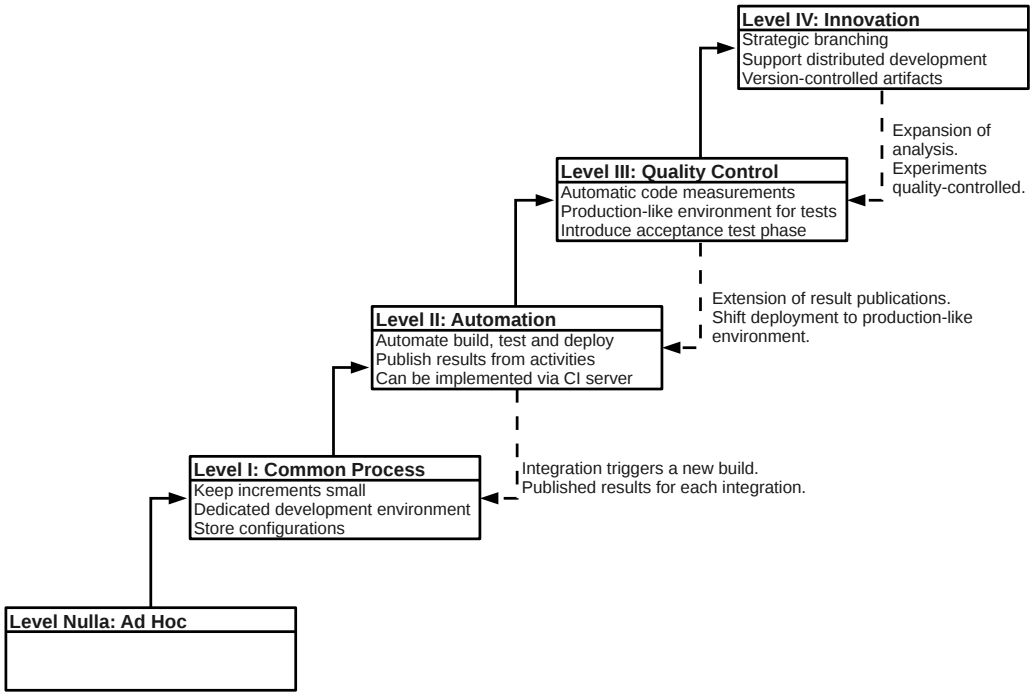

Figure 7.3: Our *Software Development Maturity Model.*

7.3 Further Studies

During our work we had to make some prioritizes since we had limitations, in both time and scope. We hope that someone will continue where we left off or advance on other topics that we have bordered on in this report. In this section we will therefor discuss what areas we have noticed could require future work.

In discussions within section 7.2 we compared the results of our SDMM with Praqma's CDMM [Pra13]. However, there are a number of differences for where in the maturity levels we put certain activities compared with Praqma's model. We made our ordering based on experience and later maturity levels on literature. Because of the major differences in the two models mentioned it might prove

interesting to conduct further studies, both theoretical and practical, for which maturity model that is the most beneficial to use under for example certain conditions (i.e. with a certain domain). As we have stated, the two latter maturity levels of our SDMM model are strictly theoretical and will indeed require some practical evaluation before they can be deemed fit for use.

One topic we did not prioritize while doing the ASP.NET project case analysis was the database management in our domain. We arrived at the problem somewhat late and due to prioritizes we had to put this potential problem to the side. The problem (A.RP5, see section 3.1.1) was that the database for the ASP.NET case was altered live in the test environment and therefor not version-controlled together with the rest of the application, posing restrictions on if we could rollback the application to an earlier version if we had no corresponding database version. How to solve this perhaps "fatal" behavior in relation to our attempted implementation would maybe be something that can be pursued in future work.

One thing we noticed while researching principles of configuration management for the ASP.NET project was that there was very little literature on web SCM. The literature we found was written some time ago, around the millennium shift, but we were unable to find any "modern" literature within this field.

Chapter 8

Conclusions

The work that has been conducted during this thesis gave us a lot of ideas, some of them which we were able to fulfill with implementations. The majority of results gave positive impressions to continuous delivery. No one can deny that reaching high levels of software development maturity requires effort from the majority of the organization, but there is no particular reason as to why this should not be one of the most prioritized goals.

The most promising result from this master's thesis has been that even minor changes, which requires very little effort, have significant effects and benefits on a software process. Although, implementing a solution for one of the parts of the studied project proved only to be a challenge for reasons related to installations and first-time setup. Since neither of us have had much experience in this type of work, this resulted in unexpected problems.

We saw that the changes we introduced and have discussed will be very beneficial for a software company and the benefits are discovered almost immediately after implementing solutions. The results have also shown that a developer will increase his or hers interest in their own work, because it is their own work that a process improvement is targeting. In the long run, a changed process will also become a natural way of working, which eventually leads to further improvement and higher levels of maturity.

There are still some issues that needs to be tackled linked to the view of concepts, such as continuous delivery and the tools developers can exploit to simplify activities. One of the most common issues we ran into during the project was a misguided view that tools are something of a wizard or a magic wand. The larger application, for example a continuous integration server application, is supposed to magically solve all your problems while you, the developer, only have to install the application and tell (configure) it what to do. The magic wand consists of tools like your version control system that you somehow wave over your software code and every fault is corrected.

The above mentioned issues are what we found in our context, taking another context into consideration might have a lot more issues or perhaps fewer. Our work showed a general solution can only be abstracted for a specific domain, which is different from company to company. To reduce the friction in a software process improvement and remove ad hoc procedures a maturity model, like the one we developed, will narrow the scope for key areas and assist in creating maturity goals to reach higher levels of development maturity.

It proved very well to combine the principles of continuous delivery together with that of software development maturity. The end result became a model that is incremental by nature and more related to development activities and therefor more easily interpretable for developers. Although faster releases is a given benefit for continuous delivery this thesis has shown that the concept is much more than that. The core feature, the build pipeline, has shown to be a well-designed approach when implementing control and automation of your software.

As a final note it is worth taking the future into consideration. The solutions presented here might require modifications depending on changes in technology. However we can, with confidence, say that the problems related to software development that are the main focus for a process improvement will forever be present. The impact they have on development might differ, but the root causes will always require maintenance.

Bibliography

[and] Managing Projects. `http://developer.android.com/tools/projects/index.html`. Retrieved June 26, 2013.

[Bab86] W.A. Babich. *Software Configuration Management: Coordination for Team Productivity*. Addison-Wesley, 1986.

[Bec99] Kent Beck. Embracing Change with Extreme Programming. *Computer*, 32(10):70–77, 1999.

[Bur03] Ilene Burnstein. *Practical Software Testing: A Process-Oriented Approach*. Springer-Verlag New York Incorporated, 2003.

[Dar00] S. Dart. *Configuration Management: The Missing Link in Web Engineering*, chapter 3. Computing Library. Artech House, 2000.

[FF06] M. Fowler and M. Foemmel. Continuous Integration. *ThoughtWorks*, 2006.

[HF10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[HRN06] Jez Humble, Chris Read, and Dan North. The Deployment Production Line. In *Proceedings of the Conference on AGILE 2006*, AGILE '06, pages 113–118, Washington, DC, USA, 2006. IEEE Computer Society.

[Mit09] Scott Mitchell. Precompiling Your Website (C#). `http://www.asp.net/web-forms/tutorials/deployment/deploying-web-site-projects/precompiling-your-website-cs`, 2009. Retrieved June 25, 2013.

[PCCW93] Mark C Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V Weber. Capability Maturity Model, Version 1.1. *Software, IEEE*, 10(4):18–27, 1993.

[Pra13] Praqma. Continuous delivery maturity model. `http://www.praqma.com/papers/cdmaturity`, 2013. Retrieved June 18, 2013.

[PWCC95] Mark C Paulk, Charles V Weber, Bill Curtis, and MB (Ed.) CHRISSIS. *The capability maturity model: Guidelines for improving the software process*, volume 441. Addison-wesley Reading, 1995.

[Urb12] UrbanCode. ROI: The Value of Deployment Automation. `http://www.urbancode.com/html/resources/white-papers/The_Value_of_Deployment_Automation/`, UrbanCode, Inc., 1228 Euclid Ave., Suite 855, Cleveland, OH 44115, 2012.

Appendix A

Appendix

A.1 Title Page Figure

The figure on the title page is a modified version of a figure published under the *GNU Free Documentation License* (GFDL), version 1.2 or later. This means that the figure on the title page is also published under the same license. This means that permission is granted to copy, distribute and/or modify under the terms of that license. A copy of the license can be found in the following URL: `http://www.gnu.org/licenses/fdl.html`.

A.2 Division of Work

Chapter 1. Introduction. Both.

Chapter 2. Background. Fredrik: 2.2.1. Viktor: 2.1, 2.2.2. Both: 2, 2.2, 2.3, 2.3.1.

Chapter 3. Locating the Gremlin. Fredrik: 3, 3.1, 3.1.2, 3.1.3. Viktor: 3.1.1. Both: 3.2.

Chapter 4. The Cornerstone. Fredrik: 4.1, 4.2.3. Viktor: 4.2.1, 4.2.2. Both: 4, 4.2.

Chapter 5. Designing the Solution. Fredrik: 5, 5.1, 5.1.1, 5.1.2, 5.4, 5.5. Viktor: 5.2, 5.3.

Chapter 6. Implementation and Measurements. Fredrik: 6, 6.1, 6.2, 6.2.1, 6.2.2. Viktor: 6.1.1, 6.1.2.

Chapter 7. Evaluation. Fredrik: 7, 7.1, 7.1.1, 7.1.2, 7.1.3, 7.2, 7.2.1, 7.2.4. Viktor: 7.2.2, 7.2.3, 7.2.5, 7.3.

Chapter 8. Conclusions. Both.

Appendix B

Popular Science Article

Reaching Software Development Maturity with Continuous Delivery

Viktor Ekholm
Fredrik Stål

Faculty of Engineering, Lund University

Having a mature software development process signifies that it has undergone certain improvements to maximize productivity while the users of the process find it simple to work with.

In this master's thesis we have studied an approach to increase the level of software development maturity in a company by analyzing key areas in software projects on the company.

We were able to find that the most important first step in a process improvement is to develop a template for a common process.

As fine spirits mature and develop a more complex and interesting taste over the years, the same can, almost, be said about software development. Having a mature software development process may not produce any extraordinary taste sensations, but it does give a comfortable and effective work process. The prerequisites are to put in effort to actually mature the software process.

In short, improving your software process aims to rid you of any manual, repetitive processes and possible downtimes due to events you have to wait for to complete. Unnecessary rework is considered an anti-pattern and thus a very bad practice. Automating this hard labor intends to give developers a break from tedious, manual tasks, but in theory quality and consistency would also improve. This is because we have a smaller risk of random mistakes due to automation.

This master's thesis have studied a practice, where we aim to always have a releasable product, as a means for software process improvement. Through this practice we expect less integration problems as software code is integrated in small increments several times per day. Through automated test executions upon integration, we receive iterative feedback which gives the status of our software.

The company is first and foremost a consulting firm, but also conduct some in-house development. These development projects became our focus and it was initially perceived that one of our premises was that the in-house developers did not have a common development process.

Through interviews with developers and stakeholders, our take on the in-house environment was that the nature of the company required developers to be given assignments on other companies between projects. Their knowledge of processes is then temporarily lost for the in-house development when they leave. As there is no collective knowledge of processes, there is a lot of work only suitable for the moment at project start. This is usually called *ad hoc*. In a way the context has a relation to open-source projects, where we have to expect that developers come and go and thus require a process that would make it simple to start contributing.

It was perceived that no value is seen in taking the required time to mature and baseline processes. Work hours are considered as billable and the customer is most certainly not interested in paying good money for advancing the processes at the company after they already have accepted them as developers. The interest for the company in developing more efficient processes is to place more time and focus on developing the software and thus increase productivity.

One of the purposes of this thesis was to, through interviews and analysis, locate key areas in the development process that could benefit the most from improvement. Although being a smaller company, they had multiple projects running simultaneously with different number of developers in each one, ranging from one to six. The first target was only the projects that developed applications for the Android platform and try to extend the process by adding additional features and tools. The intention was to improve, not only to increase efficiency of the development, but also the communication with customers.

After a certain amount of weeks we changed our target to a newly started project that had three smaller parts. These parts consisted of an ASP.NET website, an Android and an iPhone application, respectively. The motivation of this change of target was to have the opportunity of accessing and monitoring a live project taking shape in real time, and not just basing our work on theoretical or old projects. Also, the diversity aspect

of this three-parted project interested us.

The primary goal then became to analyze the process used for each part and together with the developers work with improving key areas during the course of the project. It was discovered that one of the smaller projects contained a deployment procedure that was performed several times a day to a testing environment, which we saw as a perfect opportunity to study how processes are carried out at the company.

There was no unified or general approach for projects, which had led to that developers on each part of the studied project had performed implementations on an ad hoc basis. Besides analyzing the reasons and root causes of the chosen methods we also had a goal to develop a common process with the purpose of introducing automated procedures for as many of the manual activities as possible. This would further benefit our own work, but our initial hypothesis was that it would also help to increase knowledge of well-defined practices and assist in increasing the level of software development maturity on the company, which we had as a third goal.

Based on our analysis and studies we were able to determine the most simple solution for the problems we had found. The key areas of the development process turned out to be development activities such as building, testing and deploying software. By ensuring that these activities could be automated in an abstract process applicable to every project, we had developed a common process with the aims of fulfilling our goals.

In the prototype solution, development activities are separated into sequential and automated stages. Each stage has the capability of reporting its results as feedback to developers and once a stage (i.e. an activity) passes, the next stage will be triggered. For every new iteration, any additional changes to the software are collected to ensure that the latest version of the application is built and tested.

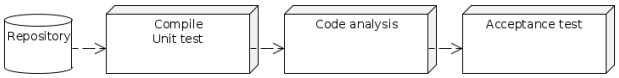

Fig. 1. An example of the prototype solution.

After we had made a prototype we held a demonstration for the developers in the project, in which we showed them how to set up the solution from scratch. Our setup handled builds, unit tests and deployments of the application automatically once a new version of the software was created. This demo with the developers generated a lot of valuable discussion and it was decided that the developers would try our implemented solution during the remaining couple of months work on the project.

The work that has been conducted during this thesis gave us a lot of ideas, some of them which we were able to fulfill with implementations. No one can deny that reaching high levels of software development maturity requires effort from the majority of the organization, but there is no particular reason as to why this should not be one of the most prioritized goals. A strong argument is that our implemented solution was able to increase efficiency by almost 10%.

We saw that the changes we introduced and have discussed will be very beneficial for a software company and the benefits are discovered almost immediately after implementing solutions. The results have also shown that the commitment of developers will increase because they no longer are forced to endure tedious, manual tasks followed by a new version of the software. In the long run, an improved process will also become a natural way of working, which eventually leads to further improvement and higher levels of maturity.

The above mentioned issues are what we found in our context, taking another context into consideration might have a lot more issues, or perhaps fewer. Our work showed that a general solution can only be abstracted for a specific domain, which is different from company to company. By first defining what goals our improvement aims to fulfill we can reduce the friction in a software process improvement, remove ad hoc procedures and be one step closer to ultimately reach higher levels of software development maturity.