Optimization of Configuration Management Processes

JOHAN KRISTENSSON
Abstract

Configuration management is a process for establishing and maintaining consistency of a product’s performance, as well as functional and physical attributes with regards to requirements, design and operational information throughout its lifecycle. The way configuration management is implemented in a project has a huge impact on the project’s chance of success. Configuration management is, however, notoriously difficult to implement in a good way, i.e. in such a way that it increases performance and decrease the risk of projects. What works well in one field may be difficult to implement or will not work in another. The aim of this thesis is to present a process for optimizing configuration management processes, using a telecom company as a case study. The telecom company is undergoing a major overhaul of their customer relationship management system, and they have serious issues with quality of the software that is produced and meeting deadlines, and therefore wants to optimize its existing CM processes in order to help with these problems. Data collected in preparation for the optimization revealed that configuration management tools were not used properly, tasks that could be automated were done manually, and existing processes were not built on sound configuration management principles. The recommended optimization strategy would have been to fully implement a version handling tool, and change the processes to take better advantage of a properly implemented version handling tool. This was deemed too big a change though, so instead a series of smaller changes with less impact were implemented, with the aim of improving quality control to minimize the number of bugs that reached production. The majority of the changes had the purpose of replicating the most basic functions of a version handling tool, as well as automating manual tasks that were error prone.

Keywords
Configuration Management, Process Optimization, Version Handling, Task Automatization
Abstract


Nyckelord
Konfigurationshantering, Processoptimering, Versionshantering, Automatisering
# Table of Contents

1 Introduction..............................................................................................................1 
   1.1 Background ....................................................................................................1
   1.2 Problem..........................................................................................................2
   1.3 Purpose...........................................................................................................3
   1.4 Goal................................................................................................................3
       1.4.1 Benefits, Ethics and Sustainability ..........................................................3
   1.5 Methods..........................................................................................................4
   1.6 Delimitations ...................................................................................................7
   1.7 Outline............................................................................................................7

2 Configuration Management and Configuration Management Processes..................9
   2.1 Software Development Methodology & Software Configuration Management .............................................12

3 Configuration Management Best Practices.............................................................15
   3.1 Version control software ..............................................................................15
   3.2 Requirements and documentation ...............................................................15
   3.3 Communication ............................................................................................16
   3.4 Shared data, multiple maintenance and the simultaneous update problem ...............16
   3.5 Baselines .......................................................................................................17
   3.6 Check-in and checkout ................................................................................18
   3.7 Versions.........................................................................................................18
   3.8 Branching and merging ................................................................................18
   3.9 Configuration Item Selection ......................................................................20
   3.10 Change and Configuration control ..............................................................20
   3.11 Configuration status accounting ................................................................21
   3.12 Configuration verification ...........................................................................22

4 Methods ..................................................................................................................23
   4.1 Data collection methods .............................................................................23
   4.2 Data analysis methods ................................................................................24
       Descriptions ....................................................................................................25
       Contexts ..........................................................................................................25
       Intentions .........................................................................................................25
       Process ............................................................................................................26
       Classification ................................................................................................26
   4.2.1 Qualitative analysis methods ................................................................26
       Content analysis ............................................................................................26
       Narrative analysis .........................................................................................26
       Grounded theory ...........................................................................................27
   4.3 Data analysis ................................................................................................27
   4.4 Results ..........................................................................................................27
       Point-wise improvements .............................................................................29
       Flow-wise improvements .............................................................................29
       Role restructuring and realignment of organization and process structure .........30
   4.4.1 Case Study Analysis ................................................................................32
   4.4.2 Implementation methods...........................................................................33
5 Optimization of Configuration Management Processes .................. 36

5.1 Overview of the CM processes .......................................................... 36
5.2 Continuous improvements .................................................................. 36
5.3 Version Control Tools ......................................................................... 37
5.4 Configuration items and configuration item selection .......................... 38
5.5 Optimization of baseline creation and maintenance ............................ 39
5.6 Check-in and checkout ....................................................................... 40
5.7 Branching and merging ................................................................. 42
5.8 Secure repository ............................................................................. 43
5.9 Optimization of the configuration control process ............................ 44
5.10 Optimization of configuration status accounting process ................. 44
5.11 Optimization of delivery, deployment & configuration verification .... 45
5.12 Labelling & comments .................................................................... 46
5.13 Version numbers ............................................................................ 47

6 Implementation of Optimizations ....................................................... 50

6.1 Patch list .......................................................................................... 50
6.2 Delivery & Quality controls ............................................................... 51
6.3 Project documentation ....................................................................... 51
6.4 Comparison ...................................................................................... 52

7 Discussion ............................................................................................ 53

7.1 Manual CM Processes ....................................................................... 54
7.2 Branching and Merging ..................................................................... 54
7.3 Security ............................................................................................. 54
7.4 Software Development Methods ........................................................ 55
7.5 Earlier Delivery to CM Team ............................................................ 55
7.6 Configuration item selection ............................................................. 55
7.7 Deployments ..................................................................................... 56
7.8 Version numbers ............................................................................... 56

8 Conclusions and future work ............................................................... 58

8.1 Discussion ........................................................................................ 58
8.2 Future work ....................................................................................... 59

References .................................................................................................. 61

Appendix A ................................................................................................ 1
1 Introduction

Configuration management (CM) has become increasingly important with the growth both in size and complexity of the applications that are being used and developed at most large corporations. The larger the system, the more expensive outages and problems will be [1].

These days it is common for a company to have several different vendors, partners or suppliers to be working on different versions of the same requirements or designs. Together with the growth of systems and the constant rise in the number of integrations, it has become ever more difficult to get a clear picture of the status of a system at any point in its life cycle [2].

As a result of changes to the software development process many companies work under inefficient or outdated configuration management processes, and could greatly benefit from optimizing these processes.

1.1 Background

Configuration management has been identified as having a major role in reducing costs, minimizing the risks, and avoiding delays in both development and maintenance of software projects [3]. Optimizing a CM process can therefore help prevent project failure and decrease the number of errors reaching production, while at the same time lowering the costs.

Many major corporations that were among the first to start implementing their own IT systems face a huge problem today since many of their systems are built using completely different technologies, and initially never were intended to communicate with each other, or at least not to the extent that is needed today. The result is a patchwork of integrated legacy systems that are increasingly difficult and costly to maintain, patch and upgrade. The resulting architecture is in many cases poorly suited to handle current demands [4].

Initially only two options were considered, maintaining and developing the legacy systems, with increasing costs and decreasing return on investment, or replace the systems entirely. Replacing a legacy system is often costly, and dangerous, since there is a risk of losing vital business knowledge that exists within the system. A more recent method that focuses on reengineering the system, i.e. improve and restructure it in such a way that it becomes easier to change and maintain, and better meets the requirements of today. This reengineering method falls somewhere between these other extremes, and is generally cheaper. These reengineering projects are often called transformation projects [4].

An important part of transformation projects may be to consolidate systems, and migrate similar products into a common system. CRM systems, which increasingly need to be able to present a complete picture of a customer, often becomes the central player in such transformation projects. Large CRM systems
often handles millions of customers and are invaluable tools both for customer support, as well as the cornerstone of any web based self-help services the company is providing its customers. It is therefore of utmost importance to minimize the number of errors that reach production in such a system, due to both poor publicity and high costs in case of downtime or severe errors in production.

Due to their complexity transformation projects often suffer serious delays, increased costs and poor quality, resulting in a large number of problems reaching production. The problem may be aggregated by the fact that many large projects are outsourced, due to a lack of expertise in house or a desire to let someone else take the risk. Outsourcing may in many cases limit the control the company has over the process.

Configuration management presents a unique opportunity to regain control over what could otherwise be a chaotic and costly affair. What makes CM such a good target for minimizing problems and reducing costs is that CM is the process that interfaces with software development, accepts changes and ultimately handles deployment. This means that even if the company is not in control of the software development process itself, anything that goes into production has to pass through the CM process, meaning that quality controls can be implemented at different stages to find problems as early as possible and ultimately prevent them from being deployed. A company may therefore as a first step analyze its own CM process with the goal of optimizing it to find a wider range of problems, as well as finding these problems earlier.

1.2 Problem

In many organizations key knowledge resides in a few individuals, and is rarely properly documented, leading to a low so called Truck factor [5]. The truck factor measure the concentration of information in individual team members, and refers to the number of people that can be unexpectedly lost from a project or system before it grinds to halt or collapses due to lack of information or people with the necessary competencies. Having proper processes as well as tools to track and manage change has become increasingly important in today's world where short contracts and job hopping is much more common. In addition a greater number of tasks are performed by consultants rather than by employees. With poor documentation, processes and a constant flow of people in and out of projects serious errors may pass through the CM process unnoticed, regardless of how good it is [6].

It is common for large corporations to have some sort of software configuration process, use CM software, such as version control tools, and for software development projects to include the role of configuration manager. The exact implementation varies from company to company. In some cases a software project will have dedicated resource working as configuration manager (CM), but often this role will be filled by for example a developer who works as a CM on the side. This can contribute to poor CM implementation, since CM will not
be that person’s primary concern. In cases where the development is outsourced to a vendor, a company may often have a CM function that receives deliveries from outsourced projects, but do not run the CM process in the development projects themselves. This too can create problems, since you have two different CM processes that has to interact with each other, rather than a single one [7].

Considering the long history of project failure within software development, it is clear that there is plenty more than can be done to improve the situation.

All these problems relate to how the configuration management process is set up, and these problem can thus be solved by optimization of the configuration management process. This thesis presents a method for solving this problem, using a real system as a case study.

1.3 Purpose

The purpose of the thesis is to present optimizations of configuration management processes. Optimization generally means to make something as functional or effective as possible, and in this context optimization of CM processes means making the CM process optimal in terms of finding as many different kinds of problems as possible, and discovering them as early as possible.

The earlier a defect can be found, the cheaper it generally is to correct it. By preventing defects from reaching production, it will also be possible to limit to impact on end users, i.e. customers. Depending on the severity of the bugs reaching production the consequences both in economic terms and in terms of good will could be catastrophic.

This thesis uses a real system as a case study to showcase how a CM process can be optimized, and what the results of such an optimization can be.

1.4 Goal

The goal of the thesis is to present a method for optimizing a CM processes, with the aim of finding problems earlier and minimizing the number of defects that reach production. The optimization of the CM process of a CRM system is used as a case study for the optimization process, with the goal of making it possible to do similar optimization of other processes at other systems.

1.4.1 Benefits, Ethics and Sustainability

An optimized CM process should help a company meet project deadlines, lower costs while at the same time reducing the number of problems in production. Fewer problems would ultimately benefit the end users, e.g. customer support for a CRM system, as well as customers using any services that has interfaces against the system for which the CM process has been optimized. Less problems
in production would also mean less downtime, due to shorter of fewer deployment windows for bug fixes.

By optimizing the CM process any development project is more likely to be successful, resulting in better utilization of resources. Successful systems with fewer bugs is also likely to have an extended life, significantly decreasing costs for new development over time. Therefore optimization of CM processes provides an economical benefit to the company, which in turn can provide a benefit to society in the form of better and cheaper products.

With increased reliance on digital systems, the importance of high uptime and correct behavior can only increase. Optimization of system support processes, like the CM process, will thus become more and more important and add more value over time.

With an increasing amount of our lives being digitalized, more and more sensitive data is being stored in different systems. Optimization of CM processes could help protect this data, by limiting bugs or problems that could either prevent access to the information, or reveal the information to malicious parties.

Configuration management processes has no real ethical impact, so the optimization of such processes will not lead to any ethical impact either.

Optimization of CM processes has no direct impact on sustainability, but in some cases optimization could extend the life of a system, resulting in a lower resource usage over time. The greatest benefit is arguably economic though.

1.5 Methods
Traditionally, a research study design can be either quantitative or qualitative [8].

Quantitative research quantifies a problem and involves the use of measurements, i.e. assigning numerical values to different aspects of the studied problem. This subsequently allows mathematical procedures for drawing conclusions, such as statistical averages, standard deviations, correlations etc [9].

Qualitative research is primarily exploratory and it focuses on an in-depth analysis of the object or problem of study. The purpose of it is to understand the problem and while in the quantitative study the analysis is statistical, here the analysis is interpretive [10].

However, according to Trochim & Donnelly [11] all research can be said to be quantitative, since everything can be counted, regardless if it is abstract, and at the same time all research can be qualitative, because even purely numerical
statements may conceal multiple meanings. They therefore conclude that the difference is not so much in method, as in approach.

If similar studies are rare, no previous research data exist, and there’s no definitive way of measuring outcomes the qualitative method, or approach, is better [11].

Most research within configuration management focuses on implementation of processes, or finding commonalities between different fields, i.e. best practices. Some of these best practices are contradictory, and most of them can be implemented on different levels or to different degrees. Some practices may work better in one field than another, and vice versa. In effect there is no previous research or research data, except in very broad terms, that is applicable in this case.

In general, it is hard to measure the effectiveness or outcome of a changed configuration management process, since it interfaces with other processes on many different levels, primarily the software development and maintenance process. In the same way there are many different categories of people who are ultimately affected by changes, e.g. users, developers, customers, maintenance.

The result of a changed configuration management process may be in form of changes in code quality, number of projects that meet deadline, changes in the cost of projects, change in the number of found bugs in production, change in goodwill with regards to customers or end users. To make the matter worse, most of these factors are most likely weighted differently for different companies, making measurements and comparisons on a numeric basis almost impossible.

Taken altogether it is clear that the qualitative method, or approach, is better for evaluating configuration management processes.

Throughout history, humans have sought knowledge in different ways. According to Ary et al [12] two major sources of knowledge that are more systematic than learning by individual experience or external authority and that can thus be used as a basis for building scientific inquiries are deductive and inductive reasoning.

Deductive reasoning is the process by which one proceeds from general to specific knowledge using logical arguments. The inductive reasoning process follows the opposite direction, i.e. it involves building a general conclusion on specific instances of observation. Both of these approaches can be used in a scientific study, depending on the specific research design chosen and the circumstances of the problem studied. Deductive reasoning is more associated with quantitative research designs whereas qualitative research designs more often use the inductive approach [13].
Considering the complexity of the problem, and the choice of a qualitative research method the inductive method is preferable.

According to Merriam [14] there are eight common types of qualitative studies: basic interpretive qualitative study, phenomenological study, grounded theory study, case studies, narrative analysis, critical qualitative research and postmodern research.

Out of these the basic interpretative qualitative study, the phenomenological study and the case study have qualities that make them suitable for performing a study of IT systems and processes within large companies. These are more suitable than the others because they try to see the system from the inside, rather than taking an outside perspective. These are also better for analyzing specific problems rather than generalities. They also focus more on observable data rather than making interpretations and inferring conclusions.

A basic interpretative qualitative study is a basic example of a study that uses the inductive approach. It consists of collecting a wide range of qualitative data, by the help of interviews, observations, documentation studies, focus groups, and drawing interpretations and conclusions based on the information obtained from them.

In a phenomenological study [15] a combination of methods is used in order to capture the individual perception that people involved in a certain problem or phenomenon have of it. Most often in-depth interviews are used on multiple individuals in order to look for commonalities of their experiences and views of the problem.

The essential question in case study research is “what are the characteristics of a specific single case (or multiple cases)?” A case here is the unit of analysis and can refer to any specific phenomenon or system. For Yin [16] a case is “a contemporary phenomenon within its real life context, especially when the boundaries between a phenomenon and context are not clear and the researcher has little control over the phenomenon and context”. Yin [16] further states that a case study is a particularly suitable method in situations where the main research questions are “how” and “why”, the researcher has little or no control over behavioural events and the focus of the study is a contemporary (as opposed to a historical) phenomenon.

Robert Stake [17] classifies case study research into three types: an intrinsic case study, which focuses on simply understanding a particular case, an instrumental case study (which includes an interest in understanding something more general than the case itself) and a collective case study, where multiple cases are studied and compared within a single study.

Since the goal of the study was to solve a unique problem within a specific strictly structured context, a case study was the most suitable approach. It was performed using an approach, which could be seen as a mixture of an intrinsic
and instrumental case study. While it focused on an individual instance of the problem (configuration management practices at a given point of time in a given company) it was also instrumental to the extent to which it actively aimed to solve the problems identified.

Hence, methodologically speaking, the study was a case study in line with the principles described above. In the first stage, the focus was on using scientifically and methodologically sound principles in order to get an insight in CM practices in general and in the specific context of a company. In the second stage, the study had an instrumental component, as it had served to not only describe and explain, but also solve the problems identified.

1.6 Delimitations

Since configuration management is tightly coupled with software development, the chosen software development method will have a huge impact on how the configuration management process can be optimized. In order to truly be able to optimize a configuration management process, the software development method has to be taken into account too, and most probably changed too. In many cases this will not be possible though, due to for example outsourcing, common development methods for all systems within a company etc. And even if possible, the gains from changing a software development method has to be weighed against possible drawbacks on a larger scale, e.g. using a different development method than other systems at the same company.

In the case study all development of the CRM system was outsourced to third party vendors, which the company has no control over, which caused the optimization of the CM processes to be limited to the part of the CM process that the CRM system has direct influence over, i.e. development projects starting with the delivery to acceptance test and ending with deployment, and the correction process and deployment of production patches. The case study was also limited to changes that could be done within a short period of time, since the need for changes was very urgent. Lastly only changes which were deemed to have minor impact on the development projects were allowed, since further delays due to process changes was unacceptable.

1.7 Outline

In chapter 2 configuration management and configuration management processes are presented, as well how configuration management interacts with software development. Chapter 3 contains a literature review of configuration management best practices, which is the basis of part of the analysis and optimization. Chapter 4 presents theories of methods for information gathering, analysis and implementation along with description of chosen methods. Chapter 5 contains the actual analysis of a CM processes in the form of a case study. Chapter 6 describes the implementation of the changes that were the result of the analysis. In chapter 7 a discussion of the study is presented.
along with suggested changes that were never implemented, and finally in chapter 8 the conclusions are presented.
2 Configuration Management and Configuration Management Processes

Configuration management is a process for establishing and maintaining consistency of a product’s performance, as well as functional and physical attributes with regards to requirements, design and operational information throughout its lifecycle.

The configuration management is usually divided into five disciplines or sub processes, as established in the MIL–HDBK–61A and ANSI/EIA-649 standards:

- CM Planning and Management
- Configuration Identification
- Configuration Control
- Configuration Status Accounting
- Configuration Audit and Verification

CM Planning and Management is the process for planning and documenting the other processes. It is also used for resource planning and allocation, naming conventions, training etc.

Configuration Identification is used for defining what is under configuration control, i.e. what should be version controlled. This provides the basis for creating and maintaining baselines which is the input to Configuration Status Accounting.

Configuration Control is the process for handling changes, from evaluating Change Requests (CRs) to approving and rejecting said CRs. Handles changes on all levels, not just software and hardware, but also changes to standards, documentation and firmware.

Configuration Status Accounting is a process for gathering and reporting the status of configuration items, as well as any deviations from the baseline during design and development.

Configuration Audit and Verification is essentially an independent review of configuration items and processes with the aim of determining if they are compliant with established functional requirements, required performance and chosen standards.

Previous work within the field of configuration management have found a large number of best practices for implementing configuration management, and the same best practices can be used to reimplement or optimize an existing processes.
David Bellagio and Tom J. Milligan identify the following SCM Best Practices in “Software Configuration Management Strategies and IBM Rational ClearCase” [18]:

- Identify and store artifacts in a secure repository
- Control and audit changes to artifacts
- Organize versioned artifacts into versioned components
- Create baselines at projects milestones
- Record and track requests for change
- Organize and integrate consistent sets of versions using activities
- Maintain stable and consistent workspaces
- Support concurrent changes to artifacts and components
- Integrate early and often
- Ensure reproducibility of software builds

Due to the nature of some of these best practices, fully implementing one best practice will impact the feasibility of implementing others, i.e. they are not independent of each other. For example integrate early and often become more difficult the heavier the implementation of control and audit changes to artifacts are. Therefore a successful implementation of configuration management very much involves finding a set of best practices with different levels of implementation that suits a particular company or industry. In the same way optimizing a configuration management process may involve both implementing new best practices and tweaking or shifting focus between already implemented best practices.

These best practices are very broad and a single best practice span many different kind of actions in a configuration management process. It is therefore sometimes better to look at specific actions and how best practices can be applied to a specific action or a specific problem within the field. This is perhaps especially true when it comes to optimizing a process, since the overall structures are in place, but some problems may be unsolved, only partially solved or actions may not be performed properly.

A division by action, area or problem may be constructed in many different ways, primarily by using different granularities. Drawing on the work of David Bellagio and Tom J. Milligan actions and problems can be grouped into the following fairly broad topics [18]:

- Requirements and documentation
- Communication
- Shared data, multiple maintenance and simultaneous update problem
- Baselines
- Check-in and checkout
- Versions
- Branching and merging
- Configuration item selection
- Change and configuration control
- Configuration status accounting
- Configuration verification

The three first topics, Requirements and documentation, Communication and Shared data, multiple maintenance and simultaneous update problem, all deal with general problems that any configuration management process needs to solve.

Requirements and documentation deals with the importance of proper documentation and to extend version control to documentation as well as source code. Communication looks at the general problem of communicating, which has become even more important with large teams that cross both geographical and cultural boundaries. Finally Shared data, multiple maintenance and simultaneous update problem looks at what is probably the most fundamental problem of configuration management, i.e. how to enable working with shared resources while maintaining consistency. This is the fundamental basis for the four next topics, baselines, check-ins and checkouts, versions, and branching and merging, which all corresponds to the basic actions of a configuration management process, normally implemented by help of some version control software.

Fundamentally a baseline is a snapshot in time of either all version handled items, or a subset of them. Baselines are generally used to mark specific milestones or criteria, e.g. stability, and number of defects, and this baseline can then be used for, for example, new development, and deployment. This leads to the concept of versions. Any change of an item will create a new version, so a baseline essentially points to a specific version of all items in the baseline.

The actions of check-in and checkout essentially corresponds to preparing to create a new version, and committing that version respectively. This in turn takes us to branching and merging. If more than one person is changing the same item, a conflict may arise with regards to which version or with set of versions to commit, which is solved by the act of merging. Branching is essentially the act of creating a separate copy of an item, or a set of items, in order for several people or even teams to be working on separate problems without interfering with each other.

The last four topics primarily deal with the processes around version control, how to set up version control, how to handle changes, how to handle verification etc.

Configuration management has its origins in engineering, and the first adaptations to software development were heavily influenced by the focus on detailed requirements and rigorous change processes, due to the high cost of late changes. Software configuration management and software development methodologies have evolved together, and as a result it is sometimes hard to say where the line between the two can be drawn. With roots in engineering and
manufacturing it is not surprising that the first software development methods and software configuration management has inherited much of the focus on requirements and change management. With time new software development methodologies have emerged, such as agile methodologies, which are perhaps better suited to the nature of software development. The core of software configuration management has not changed though, but with the rise of these new software development methodologies new best practices have emerged that are better suited for these methodologies. It is therefore of utmost importance to look at both the software development methodology that is used along with the configuration management process, when choosing which best practices to implement. Choosing best practices that are suitable for a certain software development methodology can therefore have a significant impact on the optimization of a software configuration management process.

2.1 Software Development Methodology & Software Configuration Management

Software development methodology is essentially a division of software development into distinct stages, in order to better be able to plan and execute these stages. There are a larger number of different methodologies such as waterfall, prototyping, spiral development and agile methodologies. Depending on the organization and the requirements of the system, different methodologies may be applied. A single process may often be unable to meet the specific demands of an organization, and therefore it is common for organizations to implement their own processes that are either a modification of an existing implementation of a methodology or a mix of several [19].

Some of the most common methodologies include waterfall [20], prototyping [20], spiral development and different agile methodologies [20]. Older and more traditional methods such as the waterfall method have distinct phases that are executed sequentially. On the other end of the spectrum we have newer agile methods, which is based on a process of iteration, where different phases can be executed simultaneously [20].

Studies [21] have shown that errors are most frequent during design and requirement activities, and the later in the process these errors are found, the more expensive it will be to fix them. Sequential methods have their roots in the manufacturing industry where design changes late in the development cycle are extremely expensive. These were the only type of methods that were available when people first tried to bring more structure to software development. Sequential methods have since then been criticized for being ineffective for software development [22].
Sequential methods try to solve the problem of having late design and requirements changes. Unfortunately, it has often proven hard to write good requirement documents and design documents that actually capture what the end users want. Agile methodologies are focusing on solving the problem of capturing and documenting requirements by having an iterative process with continuous deployment, so that feedback can be gathered from stakeholders from an early stage when it is still possible to do major changes, without increasing costs significantly [23].

Software development methodologies are sometimes divided into heavyweight and lightweight methodologies, were basis for classification is the number and complexity of the rules practices of the methodology. Heavyweight methods focuses on detailed documentation and inclusive planning where lightweight methods focuses on short iterative cycles and relies on the knowledge of the team rather than extensive documentation. Generally all traditional development methodologies can be classified as heavyweight, whereas most agile methodologies can be classified as lightweight [24].

The single biggest difference between these two types of methodologies is the acceptance of change. In traditional heavyweight methodologies the requirements are frozen at an early stage, and after that no further changes are allowed. In contrast agile methodologies allows constant change based on
feedback from previous iterations. The ability to effectively handle change can often determine the success or failure of a whole project, and one of the major drivers behind agile methodologies is the belief that change is inevitable [25].

Considering the big differences in how changes are treated, and how important configuration management is in handling change, it becomes clear that the configuration management process will benefit from adapting to the choice of software development method. Heavy weight methods with focus on full requirements from the start and few or no changes will benefit more from a process that focuses on ensuring that documentation is complete and reviewed properly, whereas light weight methods with lots of changes will be better focused on methods that put the focus on making sure that changes are supported and done in a controlled manner.

This in turn means that the choice of which configuration management best practices to adopt also depends on which software development methodology is used.

Since it is almost impossible to create a method for optimization of software configuration management processes that is independent of software development methodology, the choice was made to focus on optimization of software configuration processes that works in conjunction with heavy weight, waterfall like software development methodologies. The reason for this is the fact that legacy systems are generally in greater need of optimization, and generally use heavy-weight methodologies. Such an optimization method is therefore likely to see most use.
3 Configuration Management Best Practices

Due to the tight coupling between software development and configuration management, an analysis of configuration management best practices without taking software development and software development methodologies into account would be incomplete.

Software configuration management processes can either be implemented manually, or be fully or partially automated using some sort of version control software.

3.1 Version control software

Version control, also known as source control or revision control is essentially the management of changes to a set of items, most commonly source code and documents stored in a repository. Each change to an item creates a new version, which in turn is identified using a version number, commonly a numeric counter starting with 0 or 1, which is then incremented by one with each new change. Version numbers can either be local or global, i.e. the version number is associated only with the changed item or to all items in the repository. All changes are always associated with a user and a timestamp.

Most version control software track changes on a file level, which is intuitive, but may cause problems when splitting, merging or renaming files. Some newer version control system, such as Git, instead considers changes to the data as a whole, which is less intuitive, but makes operations such as merging, splitting and renaming files much easier.

Revisions or changes to an item can be thought of in terms of graph theory, and this also how many graphic version control tools choose to represent changes to an item.

3.2 Requirements and documentation

A major determinant for success in any software project is how well each requirement has been captured and documented, since these documents will be an important part of future baselines. For that reasons, requirement documents, projects plans, SCM plans etc. should all be version handled as well. It is just as important to update existing documentation as to write new documentation, otherwise the documentation will quickly cease to reflect reality. Keeping the information in the organization, rather than upholding proper documentation can mean significant risks, e.g. a low so called Truck or Bus factor [5]. This is especially important in today's world where short contracts and job hopping is much more common, and a greater number of tasks are performed by consultants. With poor documentation many problems will only be discovered later in the development process, and some serious
errors can pass unnoticed through the CM process regardless of how good it is [6].

3.3 Communication

Software development these days usually involves large teams, sometimes spread over different countries and time zones, developing different parts of the system. Different modules or parts could even be developed by different vendors or suppliers. When people who do not know each other, have different cultures, and the possibility of face-to-face talk, e.g. meetings and discussions, is limited due to distance, both physical and in time, there is an increased risk of miscommunication and communication breakdown. In addition, software systems are inherently complex and often abstract, which increases the risk of communication problems further [6].

SCM can help manage these kinds of problem by controlling and managing all changes. A proper CM plan together with a version control software makes it easy to communicate changes to all involved parties, thus lessening the risk of communication breakdown.

3.4 Shared data, multiple maintenance and the simultaneous update problem

As software development expands, programs get bigger and teams (rather than individuals) worked with the same resources, e.g. multiple people using and changing global variables for an application store in a database, one of the first problems that can arise is shared data. Shared data and resources can become a significant problem in any environment where several different programs, programmers or teams share a common resource, like a common function or a database table. If one or more parties are not aware of changes made by other parties, his or her program may end up not working at all or working incorrectly when using the shared resource next time. The fact that something may stop working without anything having visibly changed may also make debugging significantly harder. This problem is generally solved on many different levels in software development by letting programmers and teams have their own resources, i.e. multiple copies are created of the same resource. This does not solve all problems though, rather you have a different problem, a multiple maintenance problem [6].

The multiple maintenance problem refers to the problem of having to keep track of several different versions of a shared resource, e.g. how many copies exist, and what changes have been made to what copy. Ideally, all the copies of a resource should be identical. This can be solved by using a repository, i.e., a centralized network storage, but this in turn leads to the simultaneous update problem [6].

The heart of the simultaneous update problem is essentially that without any control of changes, i.e. updates to a resource, there is no way of to prevent two
parties from overwriting each other’s changes, thus potentially losing for example a bug fix [6].

Properly implemented SCM using almost any modern version control software will solve all these problems. The source code will be / can be stored in a repository, where access control can be implemented. In order to makes changes a change request has to be made, only after that has been approved the resource can be accessed. The change will be implemented, tested and approved before it is checked in to the repository. With controlled changes the shared data problem is eliminated, and with only one repository and only one person being able to check out a configuration item at a time, both the multiple maintenance problem and simultaneous update problem is solved [6].

3.5 Baselines
In order to give a better overview of branching and merging, there are some other basic SCM concepts that needs to be covered, starting with the fundamental concept of baselines.

A software development project produces the following types of items [6]:

- Programs (e.g. source code, executables etc.)
- Documentation (e.g. requirement documents, system design documents, high and low level design documents, test documents in the form of test plans and test scripts, release notes, installation guides, etc.)
- Data (Primarily test data and projected related data)

Such a set of items can collectively be called a software configuration, which is the IEEE Std-610-1990 defines as the functional and physical characteristics of the software as set forth in technical documentation or achieved in a product.

A baseline is essentially a software configuration at a specific point in time. Generally it will be a specification that has been formally reviewed and met some agreed upon criteria, e.g. passing system test and having no defects with high severity.

All new development projects in existing systems will be based on a baseline, a stable configuration to start out on, and this baseline will then mark the point of departure from the main or production baseline.

There are no definite rules as to when, or how often a baseline should be established, but rather this should be determined by the SCM team based on what the current needs are. As a minimum though a baseline should always be established whenever a copy of the baseline leaves the SCM teams control, e.g. creation of baseline for new development project by other party, or handing over a baseline to the test team [6].
The main reason a baseline should be created when source code leaves the control of the CM team is that without the baseline it will be almost impossible to recreate the conditions on which any new changes are built. This is essential for determining what has been changed between check out and check in. Without a baseline to compare against and with multiple parties doing updates it can also be difficult to determine at what point in time a bug was introduced, and who is responsible.

### 3.6 Check-in and checkout

The process of adding an item into the repository, either for the first time, or a new version of an already existing item, is called a check-in. In order to check-in an item to the repository it should be reviewed and approved. The process for approval may differ depending on need and item. In order to check-in source code, the process could, for example, require the changes to have been built, deployed and tested in a test environment. Once an item has been checked in, it will be version controlled and can no longer be changed without a change request. Once a change request has been approved a copy of the items will be made by the SCM team so that changes can be made. This process is called a checkout [6].

All modern version control software allows automatic check-in and checkout of version controlled items, and makes it possible for several people to work with the same item without physically checking it out.

### 3.7 Versions

Many people use version synonymously with baseline, but within configuration management a version usually refers to a configuration item. The initial check-in of an item created the first version, and any subsequent check-ins will create a new version. The different versions of an item make it possible to track its evolution. Each version of an item can be checked out, effectively making it possible to revert to an earlier version if necessary.

### 3.8 Branching and merging

In an ideal world no more than one person would need to work on the same resource at the same time, but in many circumstances it may be necessary to do so. SCM has two tools for handling these kinds of changes, i.e., branching and merging.

In modern software development linear development is rarely possible anymore, and branching is the solution to this problem. A branch is a deviation from the main development of a configuration item, and allows for more than one person to work on the same configuration item. One of the most common scenarios being new development coinciding with bug fixes for an older version. Branches can be made from any version of an item, and it is even possible to branch from the version on an already existing branch.
In most modern version handling tools branching is a fairly straightforward task. Exact implementation varies from tool to tool, but the general idea is always the same.

In order for version numbers to uniquely identify a version on a branch, these versions contain the branch number, i.e. the version or number of the branch made from the version being branched from, as well as the version of the change on the branch itself. Whenever a version is changed, regardless if it is on a branch or not, it is always the last digit that is updated. Exactly how version numbers are implemented or presented may vary from tool to tool, but Figure 1 shows an example of how branching could be represented graphically. Version 1.6 to 1.9 in the figure represents the main version, or trunk, which usually is the production version. Two branches have then been created from version 1.6, yielding the first version, 1.6.1.0 and 1.6.2.0 on the two new branches. Version 1.6.2.1 then has a further branch, often called a sub-branch. A sub branch could for example be used for a CR, or some critical or difficult development task that needs to be done in isolation, in order to not disturb other work.

![Diagram of version branching](image)

Figure 2 Two branches from main, with sub-branch from second branch

Branches cannot live on indefinitely. At some point the changes made in the branch needs to be incorporated into the main line for that configuration item. The act of incorporating changes from a branch into the main line, or another branch in case of sub-branch, is called a merge. If changes have been made to different parts of an item, e.g., different lines in the source code, the merge is trivial, and can be done automatically by most software. The problem comes if two people have done changes to the same place, then the merge is no longer trivial and has to be handled manually. Most SMC tools have options that allow you to compare versions side by side, and choose what to keep and what to
discard. In some cases the two changes may be incompatible, and a new version has to be created drawing from both merging versions.

3.9 Configuration Item Selection

A major part of setting up a SCM system is to select which items to version control, and decide on what granularity or level to use for configuration items. Different levels may be chosen for different types of objects. With too fine a granularity the SCM team may greatly increase the administrative burden, since the number of changes generally is proportional to the number of configuration items. Another possible drawback is that the number of files may make it harder to get a good overview.

Selecting too few files, e.g. establishing configuration item on a module level, might make it harder to find where to make changes, as well as causing more overhead in form of branching and merging, since more people will by necessity be working with the same configuration item at the same time [6].

3.10 Change and Configuration control

In IEEE Std-610-1990 configuration control is defined as an element of configuration management, consisting of the evaluation, coordination, approval or disapproval, and implementation of changes to configuration items after formal establishment of their configuration identification. More simply put configuration control is all the processes that supports all change processes involving configuration items. Configuration control is perhaps the single most important part of configuration management, and it is there processes that allow changes, and only approved changes, to be made in an orderly fashion.

The processes may differ greatly from one configuration item to the next, e.g. source codes require different types of checks and tests than documentation require.

Regardless of SCM implementation, configuration control is likely to play a major part, since it is at the heart of SCM, and it is perhaps here changes can have the greatest impact, since these processes are constantly being used over and over again.

The change process can be summarized in the following steps [6]:

1. Change initiation
2. Change classification
3. Change evaluation or change analysis
4. Change disposition
5. Change implementation
6. Change verification
7. Baseline change control
A change is generally initiated by the customer, and there can be many different causes, e.g. changed requirements, feedback from end users etc. Sometimes defects and bugs are handled separately from change requests, but generally the process is the same.

In order for the change control board (CCB) to better be able to process a CR it needs to be classified. Different methods for classification can be used depending on the needs of the organization, but some of the most common include severity, importance, impact and cost. Initial classification could be done either by whoever writes the CR, or by a member of the CM team, but the classification will generally be finalized by the CCB [6].

In order for the CCB to be able to make proper decisions all changes need to be evaluated in terms of impact, cost, change in scheduling etc. Different parts of the analysis can be performed by different people, e.g. a developer could make an estimate how long time a change would take to implement or a test manager could estimate how long time it would take to test etc. The analysis state can often involve some sort of pre-evaluation, where some sort of cost-benefit analysis is made, so that time is not spent on analysing change request that will never be implemented due to cost or timing restraints.

Once the analysis is complete the CCB can make a decision. The exact process for how accepted and rejected CRs are handled may vary, but generally rejected CRs are sent back to originator with a motivation, and the approved changes are sent to the development team for implementation.

Once a change has been approved the development team can implement the change, which will then have to be verified. The exact process and method of verification may differ depending on where in the development cycle the CR is created. Generally the later a change is made the more rigorous the test and verification methods are.

Provided the verification is successful the modified CIs can now be checked in and a new baseline be created. In order to minimize overhead changes are seldom implemented and tested alone, but rather a number of changes are implemented, tested and released together.

### 3.11 Configuration status accounting

Configuration status accounting (CSA) involves tracking and recording all information that is needed in order to effectively handle the other CM processes, like configuration control and the configuration items themselves. This can be anything from lists of approved changes and their status, environment plans, release notes, summaries of defect status etc. Documents created by CSA allows the CM team to report the status of ongoing development projects, or production if the CM team also handles maintenance process, to stakeholders [6].
3.12 Configuration verification

The purpose of a configuration verification is to establish that a software system, generally in the form of a development project delivery, matches its specifications. By doing so it is possible to establish trust in the new baseline, since this will be used for further development. Verification should cover not only functional specifications but also documentation, like design documents, release notes etc [6].

Ideally it should be verified that the product should be able to be reproduced, without further design work, based on the existing documentation, i.e. it fully and correctly described the design solution [6]. This kind of review is rarely done though, which is unfortunate since documentation rarely is as a complete as it ought to be.

Verification of functionality and requirements is seldom performed by the CM team, since this is generally covered in other process, such as system integration test and acceptance testing, which in most organizations are handled by a test team. Often these teams will work independently, and in most cases all parties would benefit from increased cooperation.
4 Methods

The following sections briefly describe methods used for data collection, analysis and implementation. The methods used were qualitative and based on inductive reasoning in line with the chosen case study approach.

4.1 Data collection methods

People are at the centre of activities such as developing and maintaining software, so in order to truly understand these fields it is of the utmost important to study people as they work in their environment. This essentially means conducting studies in field settings. Data collection methods using observation originate and have primarily been developed within fields such as psychology, sociology and anthropology, but are now becoming more common in software engineering settings [26].

Observational data collection methods have successfully been used to study software design, maintenance and to capture maintenance. In some cases these results have then been used to improve software engineering practices. Due to the similarities between software development and configuration management, and the close ties between the two fields, and similar goals, the observational data collection method is a good fit for the study [26].

In addition to data collection through observation, data was collected through literature review, in-depth interviews and documentation studies. The goal of a literature review is to gather knowledge that is already known in a certain field [10].

In-depth interviews are a qualitative research technique that involves conducting intensive individual interviews with a small number of respondents to explore their perspectives on a particular idea, program, or situation [27].

Documentation studies are a common part of case studies [28]. According to Mills et al [28], documentation refers to the variety of written, audio, and visual artifacts that exist within natural contexts before a research study is initiated.

In order to gather data for the analysis interviews and workshops were performed with key stakeholders, e.g. previous CM team participants, the line manager and the maintenance team. The CM function at the system was staffed by one employee in the role of a manager, and 2-3 consultants. Rotation of new consultants into the CM team always started with a 2 week knowledge transfer, so that new team members would have a good overview of the system and its processes before starting to work. We participated in such a knowledge transfer, in order to quickly be able to start our work. The knowledge transfer consisted of theory sessions, where all existing CM processes were gone through together with the whole CM team. In addition to the theory sessions, we would work alongside the team to get hands on knowledge of the processes.
Finally information was gathered by reading system documentation. Documentation was in the form of process descriptions, guidelines and flow charts.

4.2 Data analysis methods

Data analysis can broadly be divided into qualitative and quantitative analysis. Qualitative data has to do with meaning, whereas quantitative has to do with numbers. These fundamental differences have an impact on analysis, since meaning is analyzed through conceptualization whereas numbers are analyzed through statistics as well as other fields within mathematics [29].

Since the focus of the case study was qualitative, and essentially all data that was collected was qualitative, the focus will be on qualitative analysis methods.

In a study by Tesch [30] over 40 different types of qualitative research were identified. The width of the qualitative research field also means that there are a large number of different qualitative data analysis methods, which stem in the fact that different fields within the social sciences may look upon the same phenomena with a different perspective. Tesch [31] argues that these different perspectives can be reduced to the following three orientations:

- Language oriented
- Descriptive/Interpretive approaches
- Theory-building approaches

The language oriented approaches are those that are interested in the meaning of words, how people communicate with each other and make sense of these interactions. Descriptive/Interpretive approaches are in turn those that try to describe and/or interpret social phenomena. Lastly the theory-building approaches try to identify the connections between different social phenomena. This is only one of many possible categorizations though. Regardless of how one groups or categorizes different approaches or orientation, it will help in identifying a common core within the field [29].

Description is a basis for analysis, and the analysis then lays the foundation for further description. Analysis enables us to get a fresh perspective of the data. By breaking the data down into smaller pieces, and then seeing how these interact, it is possible to get a better understanding of the underlying system, rules or interactions, and with this new knowledge we can create a new description based on the reconceptualization of the data. Breaking down the data into smaller segments makes it possible to classify the data, and the concepts that are created in order to classify the data and the interactions between these concepts is what makes it possible to create this new description. Summed up the core of qualitative analysis can be said to lie in the processes of describing a phenomena, classifying it and seeing how the created concepts interact [29].
Descriptions
According to Geertz [32] and Denzin [33] the first step in any qualitative analysis has to be to develop a thorough and comprehensive description of the phenomenon under study. Such a description has become known as a "thick" description in contrast to a "thin" description, which only contains facts. Such a "thick" description would thus according to Dey [29] encompass the context of the action, the intentions of the actor and the process which the action takes place in.

Contexts
The need for describing contexts is a recurring theme in qualitative analysis. Contexts are especially important as a way of providing a broader view, both from a social and historical perspective, of actions. This may require a detailed description of the setting in which the actions takes place, e.g. [29]:

- The group, organization, institution, culture, society etc. in which the action occurs
- The time frame in which the action takes place
- The spatial context
- Networks or other social relationships

We have a tendency to take contexts as a given, which is a frequent cause of miscommunication, when the wrong context is assumed. Correct communication therefore often depend on knowing the right context. It is also worth noting that both actions and words can constitute communication.

Intentions
Qualitative analysis often put a lot of emphasis on describing the world according to how it is perceived by various observers, possibly due to the fact that the intentions and perceptions of actors makes it possible to understand the meaning of an action for a particular actor or observer [29].

Meaning may be ambivalent and deeply dependent on context though, and in these cases it is not possible to rely on the observers intentions as a guide to interpretation. All actors perceive and define situations, even their own intentions, according to their own understanding of their motivations and the context which they act. Neither of these are self-evident, so allowance have to be made for the usual mix of ignorance, self-deception and lies. Sometimes actors to behave completely rational where actions are governed by knowledge, and in some cases as social animals dictated by drives, impulses and appetites [29].

The conclusion is that we are not able to rely on observers and actors to give a rational account of their intentions, and neither is it possible to infer intentions from behavior. In short neither action nor intention can give us a good interpretation of behavior [29].
Process
Qualitative data may often be gathered through interactive methods, and the data collection can itself be seen as an interactive process, through which the researcher tries to make interpretations of social actions. Analysis may often commence in parallel with data collection, rather than at the end of the data collection. The result may very well be an interaction between the analysis and data collection, where analysis stimulates further data collection, which in turn modifies the analysis, and so on. This way the researcher becomes a participant in his or her own study, and thus their actions may become useful for subsequent analysis [29].

Classification
Qualitative data does not speak for itself, so it is up to the researcher to make interpretations. In order to be able to communicate about and compare different bits of data a conceptual framework needs to be developed, through which the actions or events can be understood. Interpreting something is to make it meaningful to someone else, though not necessarily in the same way the original actor or observer described the action or event. Without classifying the data there is no way of knowing what is being analyzed or to make any sort of comparisons. The classification is then the foundation on which the interpretation is built [29].

4.2.1 Qualitative analysis methods
There are a number of established qualitative analysis methods such as content analysis, narrative analysis and grounded theory, which can be used for analyzing qualitative data. Some methods may be more suitable than others depending on what kind of data or phenomena is being studied.

Content analysis
With the advent of mass communication content analysis has seen increasing usage, especially when analyzing and trying to understand media and media logic. Lasswell [34] defined the core of content analysis as the question: "Who says what, to whom, why, to what extent and with what effect?"

Narrative analysis
Narratives can be said to be transcribed experiences. Narrative analysis gathers these experiences from stories, autobiographies, journals, letters, conversation, interviews etc. The core of narrative analysis is essentially to reformulate stories presented by people in different contexts based on their experiences [35].
Grounded theory

Grounded theory is a methodology for developing theory, which is based on gathered and analyzed data. The theory evolves during research through interaction between analysis and data collection. Theory may either be generated from the initial data or from an existing theory, which is then modified as incoming data is compared against the theory [36].

Data may come from interviews as well as documents of different kinds, e.g. diaries, letters, autobiographies, newspapers and other media [36].

The major difference between grounded theory and other methodologies is the focus on theory development. The researcher can aim to formulate theories on different levels. Most studies focus on developing substantive theory, even if high-level general theory also is possible [36]

4.3 Data analysis

The main priority of the data collected from interviews and workshops were to get a picture of how the different actions and interactions of the configuration management processes were actually performed. By analysing this data it was possible to put together a process model that described how the process was actually used, i.e. an as-is model.

Some earlier attempts had been made to create processes map based on system documentation, however these processes maps were generally on too high a level, i.e. some actions and interactions in the process map actually ended up representing several actions. The information gathered from system documentation was then used to create a true representation of the process as described in the documentation.

The result was two process maps, one which described the actual working process and one that described the process such as it was documented. These two process maps could then be analysed to find discrepancies between the two process models. Any such discrepancies would then become the target for further analysis, since these were likely to provide information on problems with structure, role division etc.

In addition the as-is model was analysed from a configuration management best practice perspective, in order to see what optimizations could be done to better support quality control and early error detection.

4.4 Results

As time passes by organizations and their processes tend to become convoluted. What started out as something simple or clean becomes complex and messy. The world constantly change around us, and organizations and processes rarely
adapt automatically. Analysis of an organization and its processes is the first step towards improving it [37].

Two important concepts to consider when analysing processes is intent and mechanism, i.e. are the right things done (intent) or are things done right (mechanism)? A process may for example be highly efficient, but produce the wrong things, then the focus should be intent. On the other hand a process may be produce correct results (mechanism), but in a highly inefficient manner. In the worst case scenario things are neither done right nor effectively [37].

There are two different approaches for modelling an improved a process, which can be described as the to-be and as-is method. If a process clearly is broken, there is no real purpose in starting out with the old model. It may just as well be thrown out, and the new process can be modelled as it should be after improvement. This would be the to-be method. If on the other hand the process is working, or there are external limiting factors preventing major changes, then starting out with a model of the process as it is, and then analyse it for possible improvements may be the best way to proceed. In some cases there may not even be a documented process in place to begin with, and the first step may be to just compile an as-is model to have something to work with [37].

It is best to not assume that the way things are done are automatically bad, often there is a good reason why something is designed the way it is. So it is important to clearly understand a process before reworking it, or more harm than good may result [37].

In order to prepare a model for analysis it is necessary to make sure that the model is an actual representation of reality, with all its flaws. This can then be compared with a model of the desired outcome, if such exist. This will make problems and faults more obvious, than working from an as-is model, or an abstract representation of the process. In case quantitative analysis is to be performed, actions and interactions needs to be measured, i.e. time taken and resources used etc [37].

Another important part of process analysis is to talk to actual users of the process, to hear how they use, or don’t use it, if any work arounds are necessary, if steps are logged as done, but never performed etc. Generally users of a process will be aware of inefficiencies that will not be apparent from just studying a model. In some cases it can be good to exclude managers in order to create an environment where it feels to air actual problems with the process that will only be apparent to people actually working with it [37].

According to M.A. Ould [37] processes improvements can be narrowed down to one of the following four different types:

- point-wise improvements to individual actions or interactions
- flow-wise improvements
- role restructuring
- realignment of organization structure and process structure
The three first types of changes can be used for incremental improvements, whereas the last two can be used for more radical changes.

*Point-wise improvements*

Point-wise improvements are the most fine-grained changes, such as a single work step in a production process, or an individual action performed by a person. At this level the goal is to increase effectiveness, e.g. time spent, quality of the work or resources spent. Which actions and interactions to look at depends on what kind of improvements we are looking for, i.e. is the goal to minimize the time, maximize quality or lower resource usage etc [37].

There are many different approaches to doing point-wise improvements, but some possible approaches include [37]:

- Measure how expensive individual actions are and look at the expensive ones, e.g. those that consume a lot of time or resources, since those are most likely to yield big savings.
- Errors are expensive, and become more expensive the later they are discovered. Look at where errors are discovered in the process, and trace them back to their origin, and see if it is possible to introduce quality checks in that step or later in the process so that errors can be detected as early as possible. The more steps something goes through before an error is discovered the more work is wasted and has to be redone.
- Duplication means waste. Is the same task performed by several different roles? Overlap in responsibilities?
- Look for potential error sources, and remove them.
- Look at input from third parties. Poor quality of input means higher costs. See what can be done to improve input quality.

*Flow-wise improvements*

Flow-wise improvements looks at the whole process, and tries to find improvements that decrease processing time, increases capacity or throughput, or tries to minimize the total number of resources used. This can be done by looking at the order in which individual actions or interactions are performed, and see how changing that order impacts the whole [37].

Another option for flow-wise improvements is to look at the process, and try to see where concurrency and/or parallelism could be introduced. Moving from sequential to parallel action should greatly reduce the time required to complete a path of cycle in a process, but requires the actions being parallelized to be independent [37].

Each item or case that is being processed doesn’t necessarily have to go through the same process, e.g. perhaps only purchases over a certain value needs to be pre-approved by management. If limitations can be implemented, so that each case or item doesn’t have to go through the full process, then the average
process time for a case or item can be decreased dramatically. This type of changes can also be coupled with specialization of personnel, e.g. difficult cases can be filtered out early and sent to expert users who has specialized knowledge for dealing with these kind of cases. This could also decrease both processing time and resource costs, since more generalized, and possibly cheaper resources can be used for processing the simpler cases [37].

Planning for success is another technique for speeding up processes. If B is only necessary if A is successful, then B could be started at the same time as A. This could save a significant amount of time, and for example cut the time to market, which in some cases is critical for overall success. The downside is of course that if A fails or is abandoned, then the resources spent on B is almost always wasted [37].

Processes often tend to include workarounds, who are only there to deal with problems in the formal process. By labelling actions and interactions with inputs and outputs it is possible to ask and get answers the following types of questions [37]:

- Is anything generated but never used?
- Are all inputs used to generate output?
- Is everything that is needed in the process actually generated?
- Is everything generated before it is needed?

In many cases the answers to these type of question will give plenty of opportunities for process improvements.

The final flow-wise improvement method involves error catching. Many processes contain iterative segments where something is repeated until it is correct or has reached a certain quality. The number of iterations can often be reduced by inserting error or quality checks earlier in the process, or require some sort of preparatory work to be done before something can be submitted to the next step of a process [37].

**Role restructuring and realignment of organization and process structure**

As a process is mapped a structure is revealed that in many cases may be the result of maybe decades of changes in job descriptions, organizational changes, technology changes etc. Due to these kind of changes many of the roles in a process, especially where a role consist of a group of people, may no longer be entirely rational. It might be possible to improve the process by reducing the number of interactions that are required to make the process work. Doing this will often mean having to restructure roles and what a particular role does [37].

In order to better understand the relationship between a process and the organization that used the process, both a concrete and an abstract process model is needed. The as-is model is a good concrete model. This model can then be used to create an abstract model, which focuses on the intent of the process. A comparison between the two process models should then reveal where the
division of the real world into departments and other organizational structures has introduced new interactions that break the natural division of responsibilities, and don’t add anything of value [37].

The restructuring process can be distilled into four steps:

- Draw up an as-is model with concrete roles
- Find the underlying abstract roles in the process
- Identify ways of reallocating actions and decisions of the abstract roles to the concrete roles
- Define the new concrete roles

Depending on what criteria is used for reallocating actions and decisions we may end up with the same concrete roles, but with changed responsibilities, or there might be entirely new roles that combine responsibilities more efficiently [37].

This type of modelling is also good for understanding the approval and authorization mechanisms of a process. According to W. Oncken [38] five different types of relationships, i.e. interactions, exist between a manager and the managed:

- Wait until you are told.
- Ask what to do.
- Recommend what should be done.
- Act but advise at once.
- Act on your own, reporting routinely.

By analysing the model we may be able to determine if any of the interaction are overly strict or too permissive. In some cases changing the type of interaction may improve the process, e.g. speeding it up by loosening overly strict interactions or improve quality by tightening those that are too permissive [37].

In some cases some roles possess very few actions or only participate as third parties in other roles interactions. These sort of roles may often be redundant, and the process could benefit from removing the role, and reassigning actions and decisions to other roles [37].

The interactions in a process is generally there because different real world organizational units need to collaborate. By analysing interactions between different roles, it is possible to determine if the organization is getting in the way of the process [37].

Finally there some general questions that can be asked, that can help determine if the right interactions and roles exist [37]:

- Are there pairs of roles with a large amount of interactions? This could lead to confusion and mean a poor division of responsibilities.
• Are there roles with the same interactions as other roles? This could be hiding a function that might be best dealt with by a separate role.

• Does the interaction have a buffer of some kind? Buffers generally slows processes down, and the presence of one might indicate an interaction that in reality is a sub-process of some kind that should be modelled as such.

• Does checking, authorization, request for comments etc help achieve the goals of the process?

4.4.1 Case Study Analysis
The first step of the analysis was to determine the scope of the CM process both as a function of time and distance. In order to be able to plan and coordinate large integrated releases the company was using a system where each year would be divided into 4 release cycles, all roughly 3 months in length. From a process point of view each release cycle was identical, so the release cycle became the natural unit of analysis from a time perspective.

All projects that required integration with other systems would then have to adhere to these release cycles. A normal release, called a General Release (GR), could consist of deliveries from 30-40 different systems, even though the projects involving the system generally only had integrations with 10+ other systems. The main focus during our study was a large transformation project, henceforth called Project E, which continuously delivered new installments in each cycle. In addition to this a number of smaller projects were running, which were usually implemented over one or two release cycles. The majority of these projects were integrations projects, which required software development in several different systems at the same time. In addition the maintenance process was constantly running in parallel delivering bug fixes to existing problems in production.

Considering the limitations on what parts of the process could be changed, coupled with management’s desire to do as little as possible that would impact the workings of the transformation project, the as-is approach was the only viable path, even though a complete redesign probably would have been preferable.

For the same reason changes were limited to point-wise and flow-wise improvements, since role restructuring and realignment of organization and process structure were considered too disruptive.

Since all development was outsourced to third parties the CM process interfaced with the maintenance process and software development process rather than being an integrated part. Due to the limitations regarding where changes could be applied, the analysis was focused on the parts of the process that were under the company’s control, i.e. from delivery to deployment.
Once the limitations had been established an as-is model was created, including all roles, both users and stakeholders. Once all actions and interactions were mapped, the as-is model was analyzed for point-wise improvements. The main focus was finding errors early, and looking at deliveries from third parties.

The next step involved looking at flow-wise improvements. In order to be able to better perform this analysis each action and interaction were labeled with input and output, the individual actions were analyzed to find what prerequisites had to be fulfilled in order to perform the action. This made it possible to better see if any step could be split, removed or moved earlier or later in the chain of actions. The primary goal was to move actions that was related to the verification of the quality of a delivery or action as early in the process as possible, to minimize the amount of wasted time, in case errors made it past the initial quality controls.

In addition to this each action was labeled according to how it corresponded to standard CM actions, e.g. check-in, checkout, create baseline, label etc. These actions would then be analyzed and compared to established best practices for respective action. If deviations were found further analysis would be made to determine if the deviation was necessary due to specific circumstances at the system. If no such circumstances existed the step would be redesign so that it better matched best practices. Finally an action list would be produced detailing the necessary actions to take in order to change the process. These changes would be discussed in a weekly meeting, and if approved the changes would move on to implementation.

Even though changes couldn’t be made to third party processes, these were analysed where possible too, in order to get a better understanding of the inputs to the processes that could be controlled. This made it possible to correct for errors in third party processes to some degree.

4.4.2 Implementation methods
There are a larger number of different software process improvement models, such as CMM, CMMI, IOS/IEC, Bootstrap and Trillium, which could be used for improving CM processes [6].

CMM
CMM stands for Capability Maturity Model and describes the principles underlying software process maturity, and the model can be used to improve software processes from unorganized or ad hoc principles to something disciplined and mature. CMM is organized into five maturity level: initial, repeatable, defined, managed and optimizing. Each level describes a well-defined plateau in the evolution towards mature processes, and indicates what areas an organization should focus on in order to reach a higher maturity level [6].
According to CMM the CM process should achieve the following goals [6]:
- Software configuration management activities are planned.
- Configuration items are identified, controlled and available.
- Changes to configuration items are controlled.
- Affected parties are informed of the status and content of software baselines.

In order to achieve these goals CMM requires certain commitments from the organization as well as resources in the form of funding, tools, training etc. Finally the model provides various tools for among other things measuring, analysing and auditing.

**CMMI**

CMMI was established to deal with the fact that software development concerns so much more than just software. CMMI draws on the following three models [6]:

- The Capability Maturity Model (CMM)
- The Systems Engineering Capability Model (SECM)
- The Integrated Product Development Capability Maturity Model (IPD-CMM)

This framework is primarily intended for usage on an enterprise wide level, and exists in two variants: staged and continuous. The staged approach work similar to CMM, and uses a predefined set of process areas to define a path to improvement. The continuous approach allows selecting a single or a subset of process and improve just those, which allows functions or groups within a larger organization to move independently up the maturity levels (Alexis, L., 2005).

CMMI specifies the following specific goals for configuration management [6]:
- Establish baselines
- Track and control changes
- Establish integrity

**ISO/IEC 15504**

ISO/IEC 15504 was developed by the International Organization for Standardization (ISO) together with International Electrotechnical Commission (IEC). The project went under the name SPICE (Software Process Improvement and Capability dEtermination), and the method is sometimes referred to by that name.

ISO/IEC 15504 divides process into five different categories: organizational processes, management processes, customer-supplier processes, engineering processes and support processes. In this model configuration management is classified as a support processes, and has the following goals:
Identifying, defining, and baselining all relevant items generated by the processes or project.

Controlling modifications and releases.

Recording and reporting the status of configuration items and change requests

Ensuring the completeness and consistency of configuration items.

Controlling storage, handling and delivery of configuration items.

4.4.3 Case Study Implementation

The majority of process improvement methods usually require a huge commitment from the organization, and it is generally necessary to be in complete control of the process, which was not the case here, due to part of the process being outsourced. With the development already outsourced and started, it was more or less impossible to impose any changes beyond those of already agreed upon Service Level Agreements (SLAs).

In addition the existing CM process already was fairly mature, limiting the usefulness or quick gains that could have been made from applying any of these standardized methods. The main flaw of the system’s CM process was that it was almost entirely manual, and not scalable to the demands required by the transformation project, nor used to find and handle the huge amount of bugs produced by the project.

Finally management would only allow smaller changes that would not greatly impact, and thus risk further delaying the transformation project. It might have been possible to implement one of these methods if only or two of these limitations existed, but taken together they made it more or less impossible to use any of these large standardized processes.

The system’s CM process was only interfacing with parts of the software development process, since the majority of the process was outsourced. The vendor would handle everything from requirements to implementation, and even part of the verification, in the form of system testing (ST) and system integration testing (SIT). Only with the delivery to acceptance testing (AT) did the system’s CM process take over and interface with the software development process. The separate CM process for handling error correction process in production likewise only interfaced with parts of the maintenance part of the software development process.

This set up with many external stakeholders and severe restrictions concerning changes called for a more agile approach. User participation is an important part of agile processes, and this was an important part of minimizing the effects of any changes. By involving both the development teams as well as the maintenance team early on, it was possible to get feedback on proposed changes, and perform proof of concept without causing too much disruption.
5 Optimization of Configuration Management Processes

The process of optimization is the process of obtaining the "best", if it is possible to measure and change what is "good" or "bad". Generally though what is sought, is a maximum, (e.g. salary) or the minimum (e.g. expenses). Therefore the word optimum can be said to mean maximum or minimum, depending on the circumstances [39].

To optimize a process would then mean to change a process in such a way that it produces a maximum or minimum based on some specific criteria.

5.1 Overview of the CM processes

In almost all cases development projects were wholly outsourced to vendors, which meant they had their own CM teams and CM processes, which jacked in to System A’s CM process at several points. The first contact would be a request for a baseline, which all development projects needed to have in order to start development. System A’s CM team would then keep track of each baseline, and notify projects when changes were made in production to CIs that were part of their baseline, in order for them to be aware of the need for upcoming merges.

The next point of interaction was delivery to acceptance test (AT), which was done to the System A’s CM team. Installation would only be performed after an initial review had been made by the team. The change and control processes for AT was handled by the CM team, e.g., all changes had to be approved by the CM team, and all check-ins and check outs were performed by the CM team.

After completed AT the CM team would make an additional review, and prepare documentation and make recommendations for the deployment process. Several different development projects would generally run in parallel, and all coordination between these projects were handled through System A’s CM team.

In addition to this there was a maintenance process that always ran in parallel with any development. The purpose of the maintenance process was only to deal with defects found in production. The CM team handled the whole process, which mostly served the maintenance team.

5.2 Continuous improvements

The CM function at the system was staffed both by a mix of consultants and employees. Special focus was put on a proactive approach when hiring consultants, and the importance of constantly being in control in the CM process, since any errors potentially could affect hundreds of thousands of customers. This showed an awareness of how important the CM process is, and
how important it is to anticipate problems and/or adjust the process to changing circumstances.

Unfortunately this goal was not always backed by the power or desire to see necessary changes implemented. Generally it was fairly easy to implement changes in the internal process, since this did not greatly impact anyone else intersecting with the CM team. However some major flaws were continuously overlooked or solutions to the problem dismissed, usually only with the explanation that it would be too risky to make too large changes to the process. Unfortunately neglecting a problem often only makes it necessary to do larger changes at a later date instead.

One example that illustrates this was the suggestion to change tool for version handling. System A had been using Virtual Source Safe for many years, and even though it was not used or implemented properly, forcing the CM team to do much of the actual version handling manually, the suggestion to look at changing and properly implementing a modern version handling system was dismissed, even though this could have solved many of the problems the current processes suffered from.

5.3 Version Control Tools

System A used Microsoft Visual SourceSafe, VSS, as a SCM tool. Unfortunately only a small part of the tools functionality was ever used. Two repositories were set up, a production repository and a development repository. The production repository mostly served as a centralized storage, and access was highly regulated. Only the CM team had write access, and very few other users even had read access. This could have been a contributing factor to some of the problems discussed later with regards to baselines and merging.

Since the development project had their own CM processes up until delivery to AT, and access was restricted it was more or less impossible to support parallel development through branching, using the SCM tool. Therefore support for parallel development had to be handled manually, with a lot of manual work as a result.

It is unclear for what purpose the development repository was originally setup, but maybe the idea was to set up baselines there and then let the development projects use it in their own CM processes. This never happened though, since all projects seemed to handle version control externally.

Originally the installation process for System A was manual, but a few years ago it was partially automated by help of a tool called PPM. PPM could be linked to a repository, and as a part of an installation setup you could point out specific versions for all deliverables, and PPM would then run the installation scripts for these deliverables in the specified order. Since the AT environment was to be production like, installation was done using PPM, and thus the development repository came to be used to for deliverables that were to be installed in AT.
Unfortunately the development projects did not use the development repository correctly. Instead of creating a new version for each change, a new CI was created for each new version, containing the version number in the name. This made it impossible to use the tool to compare different versions, in order to find changes between different versions. This had to be done manually or through a program like diff.

Only the CM team was allowed to make checkouts from the production repository, which meant that the tool couldn’t be used to see who had checked out what objects. However the actual checkout process did not even use the VSS checkout command. Even though VSS supports branching it was not used to create baselines for development projects. Finally since the CIs were kept tared and zipped in the repository, the merge command could not be used, even though most files in a CI were plain text, like PL/SQL scripts, which would have facilitated easy merging.

In summary the version control software was not used for anything other than storing files, with the ability to retrieve earlier versions. The manual processes for check-in, checkout, branching, merging and baselines are discussed further under respective section.

5.4 Configuration items and configuration item selection

The configuration items at System A was of two types, code modules and documentation. The code modules were further divided into 3 types, called XXE, XXIO and XXC, representing customization, integrations and conversion objects respectively. Each object was related to a functional area, and usually contained a number of different files, e.g. PL/SQL scripts, Forms, templates, Java EAR-files etc. Each object had its own installation script and could be installed separately.

Since the code packages were stored pre-packaged, i.e. tared and zipped in the repository, it was not possible to check out on any lower level. So in many cases an object would be locked for check out even though the needed changes would be in totally different areas of the object. So essentially the configuration item selection had been made on a module level.

As mentioned earlier the majority of all installations were done through a tool called PPM, which would retrieve an indicated version from VSS and run its install script. This meant that the objects had to be in the packaged form in VSS in order for this work. However nothing would have prevented the source code and the installation package from being version handled separately. Each time a file in a package was changed, the package could just have been repackaged. This would not have caused any additional work, since repacking was already part of the change process. This would have made it possible to check out files on a lower level, while still using PPM for installations.
System A was using the Oracle Application Implementation Methodology (AIM) for documents, and was mainly producing documents of the type MD50, MD70 and MD120. MD50 is type of requirement documents, whereas MD70 is a technical specification, usually written by the developer, and finally the MD120 is an installation guide. With some rare exceptions these were the only documents that were version handled, which meant that almost all other types of documents produced by a development project, like progress reports, test reports, etc. were never version handled at all. This could partly be due to the fact that the majority of the development was outsourced, but it should still have been possible to receive and version handle these documents, provided they were actually produced.

Test related objects, like test cases for regression tests and acceptance test were at least stored in HP Quality Center (QC), which is perhaps better suited for this kind of task, than a repository.

Many project documents that were produced were generally only stored by the individual that produced them, and possibly sent out through mail to interested parties. This could make it difficult to find old reports from previous releases, and could have meant that they could very well have been lost in case of an accident, or a person quitting their job. It was surprising that this had gone on for so long, considering the risks, but the fact that many people in key positions had been working with the system in the same role for many years, probably meant that there had never been any incident that would have fostered awareness of the problem.

5.5 Optimization of baseline creation and maintenance

Since development projects had their own CM process up until delivery to AT, milestone baselines were only created for delivery to AT and later for delivery to Production. Baselines for development projects on the other hand were taken out regularly. A baseline would consist of production versions of those objects initial analysis had shown would need changing. Since projects did not have access to the repository, a physical copy had to be made and delivered to the project CM. It would often happen that later analysis would show that more CI needed changing, and these would then be added after hand. The CM team would notify all project CMs whenever there were production changes for CIs in their baseline, so that they could perform the necessary merges. Having merged all production patches, was also one of the criteria for delivery to AT.

In addition to requesting baselines for existing CIs a project could also request to deliver new CIs. The CM team maintained a list of all code CIs, i.e. a list of all XXE, XXIO and XXC objects. A new CI would get a name consisting of the object type, a number, simply the next number from an iterator for that particular object type, and a descriptive name, e.g. XXIO_56_Business_integrations.
Unfortunately many projects that stretched over more than one general release, especially Project E, consistently used versions from previous releases as a baseline instead, which sometimes resulted in incorrect version numbers and loss of function due to missed merges.

5.6 Check-in and checkout

System A had two different processes for check-in and checkout, one for production and one for AT.

Two types of corrections could be made in production, warranty corrections and maintenance corrections. All errors found within a year of deployment of a software project, had to be corrected by that project, a so called warranty correction. After that time period any new defects that were found would have to be handled by the maintenance team.

The correction process generally started with maintenance finding and error in production, and thus logging a defect for the error. After initial analysis to determine the cause of the defect, it would be assigned to either maintenance or a warranty team. Both maintenance and warranty could at any time request a copy of the affected CI. Both warranty and maintenance had their own environments where corrections could be tested. In order to be able to check out a CI, the interested party first had to provide a test protocol indicating that they had a tested solution for the defect.

This process probably evolved due to the fact that it quite often took both warranty and maintenance very long time to find the cause of the problem and come up with a solution. Due to CIs being on a module level, each CI contained a lot of functionality, which meant that a wide range of problems could end up needing modifications in the same CI. At times 4 or 5 defects could be in line to be corrected in the same CI. So in order to get as many defects fixed as possible in a given patch window, check out was only allowed once you could prove that you had a solution. Otherwise a check out for a correction that would take long time, would potentially prevent other fixes from being made in the meantime.

The no check out without a tested solution did not solve all problems though. Quite often the situation arose where a check out had been made, when a new defect with higher priority was found. This would generally be solved either by reverting the checkout and reassigning the CI to the higher priority defect, or allowing several defects to be corrected in the same version. Neither of these solutions were perfect, and would often result in problems, especially if one defect belonged to warranty and the other to maintenance.

Reverting the checkout, would require the defect to be re-implemented on a new baseline. If the defects were in related areas, the whole solution would at times have to be thrown out, causing a lot of overhead.
Allowing several defect corrections in the same version lessened overhead, but could on the other hand cause difficult merges with very tight deadlines, significantly increasing the risk of errors. In addition if changes were made by both warranty and maintenance, there was always the risk of a blame game in case something went wrong. Having several corrections in the same version, also meant that all changes had to be reverted in case one of the fixes did not pass the final testing before deployment, a sure source of conflict if more than one party had delivered fixes.

The correction process for production never actually used the checkout functionality in VSS, so instead of checking out the CI in the tool, all changes were instead tracked in a custom document, called the Patch list, which was used to track all changes in production during a release. The Patch list would contain information regarding all installed patches, ongoing patches and their test status, current checkouts and a manually compiled baseline for the release.

The only plausible explanation as to how this document evolved, is that it originated as some sort of manual version control and was used for Configuration Status Accounting (CSA) before System A decided to start using VSS for version control. It is clear though that System A never implemented or made use of almost any of the features of VSS, which is probably why the patch list was still used to do the same thing VSS could have done automatically several years after the initial adoption.

As a result VSS could not be used to check what CIs were checked out, but rather the patch list had to be used. This would sometimes lead to double check outs due to human error, e.g. one person in CM team forgets to update the patch list with the status of a CI, and then another check out request comes for the same object from another party.

Normally it is good practice to not check in code that has not been properly tested, but since AKFU, the last production like test instance before deployment, used PPM for installation, the code was checked in to the production repository anyway.

Unfortunately a lot of corrections did fail in AKFU, which caused a lot of versions to be reverted in the repository, since management only wanted versions that worked in the repository, which was a bit contradictory, since there were versions in production with existing defects.

The fact that code was checked in before it was actually in production, would sometimes cause problems when creating baselines. If not careful the CM team could end up giving a baseline containing a version that never actually went into production. Since no annotation was made in VSS regarding this the patch list had to be consulted every time a baseline was created, just to ensure that no such mistakes were made. But of course, humans are error prone, and sometimes dead versions would be part of a baseline, which could cause significant headache later.
In the AT correction process the CM team would coordinate all corrections and handle checkouts. During the AT period the only prerequisite for a checkout was that you had registered a defect for the CI you wanted to checkout. If more than one defect was registered in the same CI, several corrections were generally allowed in the same version. In case different projects had defects in the same CI, checkout would be provided based on the severity of the defects.

Sometimes the development projects would deliver patches without actually having made a checkout, which would sometimes result in two parties delivering a correction on the same baseline, which would then require unnecessary merges.

Since CIs delivered to AT in the initial delivery or later patches were never really checked in anywhere, but rather just stored on a shared network drive, checkout primarily consisted in finding the delivered version, or latest patch if one existed, and then provide a copy to the project who wanted a checkout.

As mentioned earlier installations in AT were made using PPM, which required the CIs to be present in a repository. Both the CM team and the development projects had access to the development repository, so generally the development project would upload the new version to the repository as a part of the check-in process.

The check-ins in the repository did not follow convention though, but rather the development projects had the habit of creating a new CI for each version, including the version in the name of the CI. This made it harder to see and compare different version of a CI, since you could not use any of VSS built in tools for comparison. This was further complicated by the fact that a lot of times the development projects would make check-ins with incorrect version numbers.

5.7 Branching and merging

Branching allows parallel development or changes, at the expense of increased complexity in case of non-trivial merges. All modern version control software support branching and merging in some form. The CM process at System A allowed parallel development, but provided no support for it at all. All branching was done manually outside the repository, and without using branching in VSS there was no way to get support for merging either, so that too had to be done manually.

A prerequisite for implementing a proper branching and merging strategy would have required the modules to have been checked-in in a non-packaged format, i.e. not tared and zipped, which would essentially change the level of the configuration item selection from modular level to a file level. The same structure could still have been retained in VSS, and by keeping the packages PPM installations would still be supported.
As mentioned earlier when discussing check-ins and checkout procedures, parallel changes only rarely were admitted in production. Considering how often changes had to be made in the same CIs, a software support for branching and merging would have been extremely helpful. Just changing the CI level from modular to file based would have eliminated the need for most branching and merging though, since defects in the same CI rarely required changes in the same files. Actually using the branch and merge functions in VSS would further have reduced the number of manual merges, since a lot of changes even in the same files lead to trivial merges that can be handled by the software.

5.8 Secure repository

There was no process in place for handling the repository. No one in the CM team nor IT operations knew who owned the server the repository was located on, which made it difficult to get information or authorize changes. It was unsure if the database was part of any backup routines, or even how these would have been retrieved in case of a disk failure. As far as anyone knew no maintenance work was done on the database, and the software had not as far as we could determine been patched since installed. From time to time the CM team would run into errors that would make it impossible to make a new check-ins to the VSS repository. This could usually be solved by restarting the server which the DB was on. It is possible this error was caused by some sort of backup routine locking the files, preventing writing. No routine existed for handling this type of problems.

Access to the production repository was heavily restricted, with the CM team being the only ones with write access, and only a handful of other people even had read access. If the purpose of limiting even read access to the source code, this did not really work, since both source code and documentation was frequently give out to create baselines and in order to so design studies and reviews early in the development phase. In case of Project E a lot of source code and documentation was even stored externally.

If the original purpose only was to prevent unauthorized changes a lot of people outside the CM team could have greatly benefitted from having read access. This would also lessened the CM teams workload, since a significant part of the workload consisted of finding and providing copies of documentation for interested parties.

Since only the CM team had access to the Production repository, you could not use the CM tool to see who had made a change, since all check-ins have been made by the CM-team. A version history was usually maintained in those files that made it possible, e.g. documents, PL/SQL files etc. This could potentially make it harder to track who had made the changes in a specific version of a file. This was however a minor concern.
5.9 **Optimization of the configuration control process**

A lot of the change process was invisible to the CM team, since all changes up to delivery to AT was handed internally in the various development projects. So the only two change processes, was the error correction process in production and AT.

The last two weeks before deployment was generally dedicated to regression tests, and no further changes were allowed to be made neither in production nor the development projects, in order to avoid late changes that could jeopardize stability. In case of urgent or severe problems during the freeze a CCB would be put together, who would then decide to do. Outside of the freeze there generally were no CCB meetings. Only scope changes generally required a CCB. The different development projects had their own defect meetings, the CM team were seldom invited to these. So the only rule outside the freeze was that no corrections were allowed without a defect first having been logged. Exceptions could be made if authorized by either System A's Release Manager, or the Release Manager for the whole General Release.

For changes in production the CM team would hold meetings with participants from maintenance and all projects that had outstanding warranty corrections. These meetings would function as an informal CCB, where priority of defects and coordination of changes were discussed. The CM team would often act as a negotiation between maintenance and the warranty team, since these often had conflicting agendas driven by their SLAs.

5.10 **Optimization of configuration status accounting process**

The two most important documents that were generated by the CM function were the Production patch list and AT patch list. One set of these two documents would be generated for each general release. Both these documents would contain a list of all patches created for each correction process and their status, as well as the current baselines for the associated environments. These documents were also used to keep track of checkouts. Essentially the majority of the CM teams work was made in the document and recorded there.

The fact that all process associated with this document was manual, was probably one of the greatest risks with the whole CM process. The document itself was not even under version control, until after the end of the general release it was associated with, since it was so big that it would have been almost impossible to work with in VSS. The document probably saw 10 -20 updates per day, which would have generated over a thousand versions in just one release if each update would have generated a new version.

The CM function did not really generate any other standardized reports or documents. The only other recurring document was a recommendation regarding installation of a release in production based on a review made by the CM team.
A lot of other reports were generated for a release, like a SIT test report, an AT test report, individual project status reports, a release status report, etc. All these documents were generated by other functions though, and were never brought under version control.

5.11 Optimization of delivery, deployment & configuration verification

After receiving the delivery to AT the CM team would make a review of the delivery before preparing installation in AT. Any problems that were found, e.g. missing merges, wrong version numbers, missing documentation etc, would be reported back to the delivering project, so that they could fix these problems before installation.

The delivery consisted of code packages, changed or new documentation, as well as a custom document called a MD120 Master, which essentially was an installation guide for the whole release. The MD120 Master’s most important function was to map out dependencies between different modules, which then determined installation order.

Fairly often this document would be of poor quality, mostly due to the fact the projects generally never did complete installations of the whole release in their own environments. Often bug fixes were performed by hacking the already installed version of the code, rather than creating a new package and reinstalling the whole module. This meant that often new dependencies were introduced that the developers were not always aware of.

Some of these errors could be found by inspection of the MD120 Master and the individual MD120s. However some problems could not be found until the release was actually installed in AT. This was unfortunate since the installation in AT to some degree was meant to be the first of two installation tests before the deployment in production. So if lots of problems were found, this would create more uncertainty about the production deployment. Some of the most serious problems that were found, were actual cyclic dependencies.

Some sort of internal review by the projects before delivery to AT, would probably have found a lot of these errors, which would probably have saved a lot of times.

For the problem with the MD120 Master, a test installation in a production clone would have revealed these problems at a much earlier stage. Unfortunately the development projects, especially Project E, never seemed to make realistic estimates when it came to what scope they would be able to deliver in a release, which lead to constant delays, and quality control was probably one of the first things to go in an effort to meet deadlines, since their contract only had clauses regarding delivering on time, not the quality of the initial delivery to AT.
The delivery to production, essentially looked the same as delivery to AT. The same sort of checks were run again, e.g. checking for missing merges, and specifically checking that all problems found during installation in AT had actually been corrected.

As a last dress rehearsal before installation in production, the production environment would be cloned to AKFU, maintenance’s primary test environment, and the delivery would be installed there. This dress rehearsal would be a major determinant in deciding whether to deploy the release or not.

The delivery was not actually packaged or put together physically. The installation in production used PPM, which meant that in order to prepare the installation only the right version of the packages had to be pointed out, and then the installation could be started automatically. Here the process took a bit of a weird turn, and this was probably a change to the initial process that had to be made when they introduced PPM. Since the code packages were not checked in to the production repository yet, the installation both in AKFU and production actually used the versions in the development repository.

This had some major drawbacks and risks. The fact that the installations were made from a repository which a large number of people had write access to, would potentially have made it possible for a large number of people to introduce code into production simply by replacing a code package with a modified package with the same version number. It is highly unlikely that anyone would ever have noticed this until after the installation, if even then. The other problem was that all the packages in the development repository had 4-digit version numbers for AT, which meant that the version numbers would not actually match the version number for the objects that were later checked-in to the production repository.

The fact that no packing actually occurred above object level, meant that an installation could take 8-10 hours to perform. A complete reinstallation of the system in case of backup failure, had never been tested, and even if successful would probably have taken weeks, since a large part of any delivery was manual setup.

5.12 Labelling & comments

For each patch that was created in production two labels would be created, one on a module level, and one on a system level. The label on object level would contain patch number and version number for the object, and the system level label would only contain a system version.

Each patch would only contain one object, but sometimes objects would need to be deployed together in order to solve a defect. However both patches would still generate a system version. Even non-functional updates, e.g. a patch containing only document updates would generate a new system version. This meant that the system version never really had a use, since it did not describe a
state in production. Instead state was always described using object versions instead.

It is generally recommended that check-ins comments should contain information about the changes that had been made. Unfortunately this was never the case. This information would instead have to be found in the patch notes, which were not under version control. Thus you could not really use VSS to learn anything about what changes had been made, without actually going into the code and checking the annotations made by the developer.

5.13 Version numbers

System A was using a non-standardized method for assigning version numbers to configuration items, which is shown in Figure 4. The first part of the version number contained information about which general release the configuration item had last seen development in, which meant that this number could vary for items in production, while it always was the same for all items currently being developed or changed in a release. The remaining numbers were used to keep track of the version of the configuration item itself. For production only one digit was used, and for ongoing development a digit was added for every release ahead of it in the line, i.e. one digit for production for the next general release, and production the next general release for development intended for the second next release cycle.

Version numbers in:

Production (Release X) \( X.0.Y \) Release RX +1 \( X+1.0.Z.Y \)

Release RX +2 \( X+2.0.W.Z.Y \)

Figure 3 Overview of the System A version number strategy

The fact that the version number was changed, essentially a digit was dropped, during delivery to a higher instance, was problematic, since you were essentially losing information that could be useful when verifying the correctness of a delivery. The version number should therefore only have been changed after
delivery, or a mapping table could have been provided to make sure the information was not lost.

Since only production versions of an object was actually stored in VSS, you were unable to see the incremental changes to an object throughout the development phase, without going back and looking at versions in the development repository. Unfortunately the development repository was not used to version handle code. Each new version of the code was checked in as a separate object, which along with the fact that vendors did not always use standardized version numbering, frequently checked in code in the development repository with the wrong version number, and dropped a digit when delivering to a higher instance, meant that it could be extremely hard to follow the history of an object.

Essentially this meant you had no way of knowing if a version delivered in a release was actually correct, just by checking in the production VSS. The version numbers themselves did not give any such information either.

This could also be problematic with parallel development, since each deployment of a release in production, should trigger an update of version numbers for all related objects in ongoing projects.

![Diagram](image_url)

**Figure 4** Change of version number due to rebase, and dropping of digit referring to the previous, and now deployed version.

Most of the time it would have been easy to see if this had not been done, since the version digit pointing to the production version would be incorrect. However you could run into problems, since you sometimes ended up having the same internal version numbers in two subsequent releases, meaning you could be using an old version, but the version number would still appear to be correct.
Figure 5 Risk of missed merge for upcoming releases when current release has same last digit as previous production version.
6 Implementation of Optimizations

As described earlier an agile iterative method was used for optimizing the CM process. Due to the limitations described before we were essentially limited to either small changes, or changes that only affected the CM team, which restricted both possible implementations as well as implementation methods.

All suggested changes were discussed in a weekly meeting with management. Management would rank changes in accordance with our analysis of cost, implementation time and impact on other stakeholders outside the CM team. Small changes could generally be implemented right away, whereas larger changes would require a proof of concept implementation before it would be incorporated into CM process.

Changes would be implemented, then evaluated to see if the outcome matched our expectations, and we would ask impacted stakeholders to give us feedback on the changes. This would repeat either until the process ran smoothly, or it was deemed unfeasible to reach the current goal with the suggested process change. We would then revert the change, and go back to the drawing-board.

The analysis found the check-in and checkout processes for both AT and production, as well as the review of delivery to AT and production to be the greatest sources of errors, and changes there were most likely to yield the best result for the least effort or change.

6.1 Patch list

Since the Patch list already contained all essential information, although compiled manually, it seemed natural to try to automate part of the job, in order to reduce workload and reduce problems caused by human error. Automatization was implemented by help of VB-scripts. Check-in and checkout functions were added, which would contain some of the functions of a version control tool, plus custom made checks specific to System A. Information would be collected from the user through dialog boxes.

When checking out, the user would be asked to specify a CI name, a check would first be made to see that it was a valid CI. If the CI did not exist, the user could first choose to have it recorded in the document, and then a check would be made to see if the object was marked as checked out or not. In case it was checked out, a warning would be given, and the checkout request would be added to a queue.

When check-in was made, the dialog box would ask for CI and version, the scripts would then check that the version number was correct, based on the current baseline. If everything matched a patch would be recorded in the document and a patch mail would be created. All the user needed to do was to verify the information in the patch mail and then press send.
In addition a script for updating the baseline was created. The script would essentially go through all patches that had been recorded in the document, check if they had been installed in production, and if so update the CI in the baseline with the new version number. The patch would then be marked as having been added to the baseline.

All these changes to the patch list made it possible to create a large number of patches in short time, which was often important both in production and AT since all parties generally were very late with check-ins, and you wanted to get as many patches into an installation window as possible.

6.2 Delivery & Quality controls

In the original review of deliveries to both AT and production, the CM team was only required to do random sample checks to determine quality. However this obviously missed some errors, and manual inspection is both slow and error prone. As mentioned before, one common problem was that the delivery missed production merges, something that could easily be checked just by inspecting the version number, but the MD120 Master initially only contained the 3-digit version for production. In order to support such a check a mapping between 3-digit version and the 4-digit version was included. A VB script in Excel was then created to check for problems related to version numbers. The mapping from the MD120 Master would be pasted into the excel document. When the script was run the version numbers would be compared against the production baseline, and the script would report if any of the objects had a missing production merge.

An initial inspection revealed that a lot of documents were missing changes from previous releases. The most likely explanation for this is that most projects did not have any process for merging of documents, and since they had parallel development, changes were made in parallel from a similar baseline, so later releases risked overwriting any earlier changes. In order to deal with this kind of problems document reviews were introduced. Unfortunately word documents are in a binary format, so there are not really any good methods for checking differences and merging. So in the end manual routines had to be good enough. After having introduced this kind of review, the projects became aware of the problem, and it lessened to some degree.

6.3 Project documentation

Since a lot of documentation generated by the development projects were not under version control, and could be hard to track down, one suggested change was to bring these under version control. This was only partially successful, but at the end of the study at least final reports from the test team and release manager were being version handled, which was a step in the right direction.
### 6.4 Comparison

A comparison of affected areas before and after implementation is summarized in Table 1.

<table>
<thead>
<tr>
<th>Action/Problem</th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality control coverage</td>
<td>Sampling</td>
<td>Complete</td>
</tr>
<tr>
<td>Version number checks</td>
<td>Manually for sampled CIs, otherwise not</td>
<td>Automatic for all CIs</td>
</tr>
<tr>
<td>Missed merges</td>
<td>Manually for sampled CIs, otherwise not</td>
<td>Automatic for all CIs</td>
</tr>
<tr>
<td>Documentation</td>
<td>Only delivery verified</td>
<td>Manual review/audit of content</td>
</tr>
<tr>
<td>Check-in/Check-out</td>
<td>Manual without tool support</td>
<td>Partially automated with tool support</td>
</tr>
<tr>
<td>Patch creation</td>
<td>Manual</td>
<td>Partially automated</td>
</tr>
<tr>
<td>Project documentation under version control</td>
<td>None</td>
<td>Test reports, status reports, etc.</td>
</tr>
</tbody>
</table>

Table 1: Comparison of CM process before and after implementation of optimizations
7 Discussion

The goal of the study was to optimize the CM process at System A in order to minimize the number of problems that reached production undetected. System A had a fairly well functioning CM process in place, the only problem was that it was almost completely manual, and therefore did not scale well with the increased workload caused by Project E. System A clearly had started the process of implementing a more modern CM process a few years back, when the code base was moved to a version control tool, but the processes were never adopted to take advantage of this tool.

Therefore the first major change that was discussed was to either fully change the processes to take advantage of what VSS could offer, or to throw out VSS and do the same with different software. The main reason for throwing VSS out was that it really did not have any cost associated with it, since no process really used it properly. In addition a lot has happened with version control software since System A started using VSS, so it would have been easy to find better software without really increasing costs. Another reason was that no one really uses VSS anymore, so it would have been harder to both find skilled people to work with it, and very little support online compared to programs like SVN and Git.

Both these suggestions were deemed unfeasible by management, especially with Project E ongoing, since they were afraid that major changes could cause further problems and delays. Admittedly it would have had a cost in the short run, but a proper implementation would probably have paid for itself in just a year or so. A proof of concept would have been a fairly cheap way to test this hypothesis, but in the end there was no money for this either.

It was surprising to see that an organization which seemed to be fully aware of the importance of CM had let this go on for so long. As already mentioned several times, management was very afraid of change, and this in combination with the fact that the process had worked so well for many years probably was the main cause it took so long before they started looking at changing the CM process.

One of the largest obstacles to change was the unwillingness to do changes that would impact other stakeholders outside the CM team. This was the main reason a proper implementation of version control tool was rejected. However it could have been possible to only partly implement proper usage of version control software, and thus only adopt those parts that only affected the CM team. Check-in and checkout could for example have been implemented properly in VSS, without it having affected external stakeholders. The drawback would of course have been that you would have had a fragmented process where some parts were done using VSS while others were handled manually, as before. The advantage would have been that it would have been significantly easier to do a full implementation at a later stage, when it became feasible again.
7.1 Manual CM Processes

Having a manual CM process also had the disadvantage of making it significantly harder to find new consultants who could easily step in and do the job, since CM professionals these days are working exclusively with automated processes supported by version control software. The same goes for developers working within or interfacing with the CM process. Any developer having worked in any sort of larger project will have experience of working with version control software, even if it is only through plugins in eclipse or similar, but very few have any experience working with manual processes. It might therefore be hard to make them understand the purpose of the manual steps, since a lot of these things are done automatically and those hidden from the developer when working with version control software. And of course without understanding of the process, the risk for errors increase.

7.2 Branching and Merging

As mentioned earlier the ambiguous version number system and manual merges were both causes of missed merges. These missed merges caused significant problems since they would reintroduce already fixed problems in production. Without knowing the cause maintenance would often end up having to do extensive analysis in the belief that their original fix had not solved the problem. A proper implementation of a version control tool would have solved this automatically, since branching and merging rely on differences between files, so a merge could never actually be missed. Changing the configuration item level from module to file level would have taken care of a majority of the merges too, and it could have been supported regardless of the process was manual or using the built in function of the tool. It is therefore rather surprising that this implementation was not allowed. One possible explanation is that management was afraid to change anything that could affect the installation procedure, which used PPM, that in turned require the CIs to be on a modular level. This too could have been solved though, by doing check-ins both on a file level and as a package.

7.3 Security

It is possible that the organization never analyzed the consequences of the initial adoption of VSS, and installation of code packages through PPM, since the access restriction combined with the installation methods for deployment of releases were in complete opposition. It is in accordance with best practices to limit access to the repository, but generally only with the intent of preventing unauthorized changes. In this instance even read access was denied to most people outside the CM team, but at the same time almost anyone could request a copy of source code or documentation, which only lead to more manual work for the CM team. The real danger was instead in how the development repository was handled. Here the combination of almost free access to anyone working within a development project combined with installation in production from the repository, could have caused serious problems.
7.4 Software Development Methods

System A would clearly benefit from further study with the aim of improving their software development methods as well. Especially since the CM processes only interfaced with the latter half of the development cycle. By looking at the whole development cycle it would be possible to make changes that would make it possible to find errors in even earlier stages, thus decreasing costs and delays even further. One of the major sources of errors and delays was that project E consistently failed to meet the scope that had been planned for each release, which caused problems for all subsequent releases, since they built upon the functionality that was supposed to have been delivered in the previous release. This caused the scope to constantly change very late in the process, increasing the risk and need for further testing. With a more agile methodology this would have been much less of a problem.

7.5 Earlier Delivery to CM Team

Another possibility for finding problems earlier would have been to make the hand over from development projects CM process to the internal CM process at an earlier stage. We did make several suggestions for how this could have been done, but unfortunately the contracts with the vendors were not set up in such a way to allow this. Otherwise it could have been a good solution to have the projects make the delivery to System A’s CM team at delivery to SIT. With several projects delivering to SIT, this usually required coordination that the System A CM team had to oversee anyway, so it would have been a natural step to take over the whole process, especially since it was very similar to the process for AT.

7.6 Configuration item selection

There are quite a few things that could have been changed in terms of configuration item selection that would have had a positive impact. However all of these were rejected due to management’s lack of interest in changing anything that actually related to VSS.

As discussed earlier, changing the configuration item level from modular to file level would have greatly diminished the number of merges needed, even without using VSS for merging. This could have had a major impact, since there were instances were projects were postponed since there was too great an overlap in CIs between them. This could probably have been avoided with a lower level of configuration item selection. The only real drawback to this would have been that if it had been implemented alone, it would break installations using PPM. This could easily have been fixed by checking in both the files, and the tared and zipped version that PPM required. If needed the packaged objects could even have been checked in to a separate repository that would have been used only for PPM installations.
Since a lot of documentation generated by the development projects were not under version control, and could be hard to track down, we suggested bringing these under version control. Here we are only partially successful, but at the end of the study at least final reports from the test team and release manager were being version handled, which was a step in the right direction.

7.7 Deployments

As mentioned earlier, both the installation in AKFU and production was made with PPM with versions from the development repository. We considered this to be a security risk since too many people had access to the development depository. It also created inconsistencies, since if you looked at the physical machine after installation, you would only see code packages with 4-digit versions, whereas the 3-digit versions were later checked in to VSS. The initial suggestion was to handle development in branches in VSS, but since this was too big a change, the alternative suggestion was to actually check in the code in the production repository earlier, so that it was available for installation both in AKFU and production. This would however had required an earlier delivery, which under current circumstances were not deemed feasible. The delivery to production was often late and contained errors that had to be fixed before the installation was actually ready to be installed.

7.8 Version numbers

If VSS had been implemented and new baselines for development had been created using branches there would never have been any real need for the complicated version number system and the majority of the problems caused by missed or incorrect merges would have vanished. With this solution being denied us, we had to look at other methods for changing the version number system to prevent errors. Some errors were prevented or caught by the change in MD120, but missed merges due to two versions in production having the same version number, i.e. before and after deployment of a new release, could still happen.

Adding the full version number, e.g. X.o.Y would not have worked since it might not have been possible to tell what numbers referred to what anymore, since the dot served as a separator as well. As a solution we suggested referring to versions with release number plus version divided by an underscore, as shown in figure 6.
Figure 6 Suggested changes to version numbers

There are many different solutions to this problem, that would have achieved the same result, and this may not necessarily be the best, but any solution that eliminated this problem would have been a huge improvement.
8 Conclusions and future work

Configuration management (CM) has become increasingly important with the growth in both size and complexity of software systems, increased outsourcing and usage of consultants. The growth of software systems has also drastically increased the costs of bugs and down-time in production. By optimizing a company's CM process more problems can be found at an earlier stage, thus limiting the damage. This has become increasingly important, since outages and problems are costlier the bigger the system is. Previous research has shown that CM can help prevent project failure, reduce costs and avoid delays.

The goal of optimizing the CM process was to limit the number of errors that reached production. During the case study the average number of deliverables increased by a factor 2 or 3 in a release, while the number of patches in production in the same release increased with around 30-50%. We can assume that the relationship between the number of deliverables and number or errors, is fairly linear, so that even though we don't have comparable circumstances it is still possible to draw the conclusion that the established changes to the CM process drastically decreased the number of patches per deliverable, even if the total number of patches increased.

The result of the case study shows that reengineering or optimizing a systems CM process is an effective method for limiting the number of errors that reach production. The case study also shows that it is possible to perform such an operation with successful results even with strong limitations on what type of changes can be done. This in turn means that perhaps even better results can be achieved with less limitations, which is something that could be explored in future studies.

The case study also shows that even fairly mature processes can sometimes benefit from analysis and subsequent optimization. The method used in the case study also showed that it is possible to achieve good results with just looking at the CM process. Most established process such as CMM and ISO/IEC 15504 tend to look at a large set of processes within different fields, not just CM. This can produce great results, but are probably of more value for organizations with less mature systems. In a way the chosen method is somewhat similar to CMMI, in that is allows an organization to independently work on improving a process.

8.1 Discussion

The goal of the study was to optimize the CM process of System A, in order to limit the number of defects that reached production undetected.

The report goes through common best practices for configuration management, and then presents the analysis of System A’s processes with respect to these best practices. Deviations are noted and discussed, and finally solutions are proposed and evaluated where applicable.
Using the case study method we successfully identified that the majority of System A’s problems were caused by the fact that they had never fully implemented their version control software. They were using completely manual processes in the face of a tremendous increase in output in the form of a large transformation program. Since the source of the problem was found only after the transformation program had started it was deemed too late to do any major changes to the CM processes.

The development projects that were part of the transformation program were run by a third party, and no changes were allowed that would impact these projects, and thus risk further delays. So the only possible solution was to do small incremental changes to the existing processes, primarily focusing on finding defects earlier and automating manual and repetitive tasks where humans could be a source of errors.

As Project E ramped up, the number of deliverables in a release increased from around 20 to between 40 and 60, while at the same time the number of patches in production during a release period only increased with around 30-50%. The relationship between the number of deliverables and the number of introduced bugs is not entirely clear, but it is probably fairly linear, meaning that the implemented changes did have a significant positive impact on the number of errors in production.

Poor documentation has been shown to cause significant problem, and this was the case at System A too. This problem diminished once proper review of documentation was established as a part of the delivery to acceptance test. System A did to some degree suffer from a low truck factor, since a lot of individuals were sitting on information that was not properly documented, and the majority of all project documentation was never version controlled.

These changes would in the end benefit customers using self-services interfacing with System A, in the form of less problems and higher up-time.

Although a proper CM implementation with full software support would have been preferable, the study at least shows that it is possible to achieve results with small incremental changes even without full software support.

8.2 Future work

The study also shows how important it is to ensure that one’s processes scale before one starts large projects. Finding some way to measure the scalability of a process could be an interesting topic for further study.

It is unusual to find a fully function, but manual CM process without any tool support. It is far more common to find instances where neither process nor tool support is used, and instances with full tool support with CM processes of various quality. By studying instances of fully manual processes it might be
possible to learn how to better transition from a state of neither process nor tool to one using both a tool and a process, since the tool may often be a distraction from the process.

The case study showed that it was possible to achieve good results just using point-wise and flow-wise improvements, in addition to increased tool support. It would be interesting to be able to measure the added benefit of applying role restructuring and realignment of organization and process structure as well. These kind of changes are likely harder to implement, since they have a greater impact on the organization, which also means higher costs.

Finally it would be interesting to be able to compare the outcome of process improvement using the as-is method and a complete redesign. It could be that the added cost of a complete redesign won’t pay off.
References

1 Kerravala, Z. (2004). As the value of enterprise networks escalates, so does the need for configuration management. The Yankee Group, 4.


5 Coplien, J. O., & Harrison, N. B. (2004), Organizational patterns of agile software development.


