NTNU
Innovation and Creativity

Build and Release Management

Supporting development of accelerator control software at CERN

Petter Enes

Master of Science in Computer Science
Submission date: February 2007
Supervisor: Reidar Conradi, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

1)A review of the literature within the field of software configuration management, including existing tools providing build and release support.
2)A review of the group's software development process and the tools that support it.
3)Improve/extend the functionalities of the tools based on the specific needs of the group and the reviews in 1) and 2).

Assignment given: 18. August 2006
Supervisor: Reidar Conradi, IDI

Abstract

Software configuration management deals with control of the evolution of complex computer systems. The ability to handle changes, corrections and extensions is decisive for the outcome of a software project. Automated processes for handling these elements are therefore a crucial part of software development. This thesis focuses on *build and release management*, in the context of developing a control system for the world's biggest particle accelerator. Build and release cover topics such as *build support, versioning, dependency management* and *release management.*

The main part of the work has consisted of extending an in-house solution supporting the development process of *accelerator control software* at CERN. The main focus of this report is on the practical work done in this context. Based on a literature survey and examining of available tools, this thesis presents the state of the art concerning build and release management before elaborating on the practical work. Based on the experience gained from the work of this thesis, I conclude with a discussion of whether or not it is beneficiary to stick with in-house solution, or if switching to an external tool could prove better for the development process implemented.

Preface

This document represents my master thesis at the Department of Computer and Information Science (IDI), at the Norwegian University of Science and Technology (NTNU) in Trondheim. It has been written as a part of a Technical Student program at CERN, the European Organization for Nuclear Research, in Geneva, Switzerland, in the period from August 06 to February 07. The work has been carried out in the AB/CO/AP section at CERN.

Firstly, I would like to express my thanks to my supervisor at NTNU, Reidar Conradi for supporting my stay at CERN and providing valuable feedback on the report. Secondly, I would like to thank Eugenia Hatziangeli for providing me with an exiting topic for my thesis, and Wojtek Sliwinski for support and guidance throughout the work. I would also like to thank the rest of the members of the AB/CO/AP section for making my stay here at CERN such an enjoyable and rewarding experience.

Last but not least, I thank the academic staff of the IDI Department at NTNU for 5 years of exceptional education and the department's management for their support of my stay at CERN.

Geneva, February 23, 2007

Petter Enes

Contents

List of Figures

List of Tables

Part I

Background and Context

Chapters

Chapter 1
Introduction

1.1 Background

The AB/CO/AP section is responsible the development of the *accelerator control system* used to run and control the particle accelerators at CERN. Control of all the major accelerators is performed from the CERN Control Center, the CCC. The control software was previously written in C and C++, but a migration to the platform independent language Java was initiated in 2002. The development of software in C and C++ had a well defined process and solid tools supporting it, but unfortunately, they were unfitted for the new programming language. The need for a new solution to support Java was agreed upon, and the decision was made to develop an in-house tool. The solution consisted of one new tool, and a re-writing of an existing one. During the last years, these tools have been extended with new functionality as requests from the section have appeared. The tools are now fully operational and are relied upon by all Java developments in AB/CO/AP. However, new functionality requirements are still appearing as the development process for the accelerator control software evolves.

1.2 Motivation

Software configuration management (SCM) addresses the problem of coordinating and controlling change in software projects. The current trend of moving towards component based development, involving independently built and released products, large development teams and even distributed locations increases the need for a set of tools to support the process.

When focusing on build and release management, we find important issues such as software versioning, automated build support and artifact repositories. The strategy of the development of accelerator control software implies a high focus on independently developed packages, including in-house as well as third-party developments. These packages make up the accelerator control system in use in

the CERN Control Center.

To accommodate such a strategy, the AB/CO/AP section has implemented a solid development process, including tools to support versioning, building, management of dependencies and installation of software into a distribution area. These tools are developed in-house, which allows the section to address specific needs in a timely and accurate manner.

1.3 Thesis Outline

Chapter 1 contains this introduction.

Chapter 2 presents an introduction to CERN, the CERN Control Center and accelerator control software.

Chapter 3 takes a look at the state-of-the-art within SCM. Focus is on functionality related to *build and release management*

Chapter 4 presents three external tools developed in different contexts. They represent important functionality within build and release management.

Chapter 5 introduces the development process implemented in AB/CO/AP for accelerator control software. This includes the tools supporting the process. The aim of this chapter is to provide an overall understanding of the tools, before the practical parts of this thesis are elaborated in Chapter 8.

Chapter 6 sums up the state of the build and release tools before the work of this thesis commenced. Furthermore, it presents the areas in which new functionalities were desired, and a formulation of the goals put in place.

Chapter 7 presents the methods used during the work of this thesis.

Chapter 8 elaborates on the practical part of this thesis. The chapter is divided into three sections, one for each extension of functionality.

Chapter 9 evaluates the process and result of the work. A discussion of possible future solutions based on the experience from this thesis is also presented here.

Chapter 10 presents a conclusion of the work of the thesis.

Chapter 11 presents future areas of interest for SCM in general, in addition to specific areas of focus for the build and release tools in AB/CO/AP.

1.4 Limitation of scope

The main goal of this thesis is to extend the *build and release management* tool used by the developers of *accelerator control software* at CERN. This implies a technical focus.

Due to the interest of the AB/CO/AP section and the level of technicality of the project, an in-depth study of the field of Software Configuration Management (SCM) falls outside the scope of this thesis. The study of the area of SCM is limited to cover only those concepts and aspects necessary to develop quality extensions to the build and release management tools used in AB/CO/AP.

The technical aspects will be elaborated.

Chapter 2
CERN... where the web was born

On the French-Swiss border, in the near proximity of Geneva, Switzerland, lies CERN [1], the world's largest particle physics laboratory. Founded as a joint venture by a number of European governments in the period after the Second World War, CERN's goal was to promote European research by gathering the best researchers from all sides of the conflict.

During its years of operation, CERN has achieved three Nobel prizes in physics, as well as providing to the world with one of the most significant innovations in newer history, the World Wide Web. CERN and its research has even been popularized through an important role in Dan Brown's bestseller *"Angels and Daemons."* Although Dan Brown's CERN is not entirely based on reality, one does not have to rely on fiction to make the story about CERN fascinating and impressive [2]. This chapter will take a closer look at CERN's history, mission and current projects.

2.1 Background

CERN was founded in 1954 by 12 European governments. Their mission was to explore the field of particle physics, also known as High-Energy Physics (HEP). At that point in time, America had taken over Europe's previously leading role in science, and to get back on their feet, Europe needed to join "forces." Investments involved in HEP research was beyond the means of any single European country. Throughout the years, more countries have joined the project, and CERN counts today a total of 20 official European members states. In addition, numerous countries from all over the world are involved through various research contributions.

The research of High-Energy Physics means studying the smallest particles of our world, the particles that are the building blocks of the universe. Many of

[1] CERN stands for Conseil Européen pour la Recherche Nucléaire but is also known as The European Organization for Nuclear Research

[2] For CERN's own comments about the book, please visit http://public.web.cern.ch/public/Content/Chapters/Spotlight/SpotlightAandD-en.html

these particles do not exist under normal circumstances in nature, and if they do, they may not be visible. To study these particles, one must create the proper conditions for them to appear, even if it's only for a fraction of a second. By accelerating other particles to almost the speed of light, and then smashing them into each other, large quantities of energy are released, thereby producing new particles. Using sophisticated instruments, these short-lived particles can be detected and data about their characteristics can be collected. Analysis of this data can help the scientific community to explain phenomena such as the Big Bang and gravity.

To accommodate the acceleration of the particles, CERN has, over the years, built a collection of *Particle accelerators*. The research within the HEP domain is closely related to the energy contained by the particles at impact. Following the theories of relativity by Albert Einstein, the energy released during a collision of particles is directly linked to their velocity, the ultimate velocity being the speed of light. More powerful accelerators can accelerate the particles closer to this limit, thus providing researchers with new, interesting data. CERN's role has been to provide the research community with the state-of-the-art within particle accelerators. Since the startup in 1954, 7 accelerators have been built, the next more complex, more powerful and more expensive than the previous. The biggest accelerator so far is the Large Electron-Positron collider (LEP) measuring a total of 27 km in circumference, located between 50 to 175 meters under earth's surface. The LEP was operational between 1989 and 2000.

CERN employs about 3500 people, representing a wide range of skills: physicists, engineers, technicians, scientific fellows, students, craftsmen, administrators, secretaries and workmen. These people are responsible for building, maintaining and operating the intricate machinery that makes up the accelerators and detectors. They also take part in preparing, running, analyzing and interpreting the complex scientific experiments. In addition, some 6500 visiting scientists, half of the world's particle physicists, come to CERN to perform their research experiments. They represent 500 universities and over 80 nationalities.

Among the major achievements resulting from the research at CERN is the discovery of the W and Z boson, for which Carlo Rubbia and Simon von der Meer were awarded the physics Nobel Prize in 1984. Also important was the breakthrough in the field of particle detectors which led to the invention of the *multiwire proportional chamber*, earning Georges Charpak the physics Nobel in 1992. [3]

[3]For an in-depth study of these discoveries, please take a look at the homepage of the Nobel Prize: http://nobelprize.org/nobel_prizes/physics

CERN's focus today is on the installation of their newest accelerator, the Large Hadron Collider (LHC). The LHC project will start its first tests by the end of 2007.

2.2 The Large Hadron Collider

The LHC will be the largest and most powerful particle accelerator ever built, with the cost of installation itself amounting to almost 9 billion euros. Thousands of researchers are working to prepare the experiments that will use the LHC, and even more will take part in analyzing the results. Trying to pinpoint the smallest fragments of the universe, the LHC pushes the boundaries of technology and the scale of science experiments.

Through the use of complex radio frequencies, LHC will accelerate protons to 0.999999991 times the speed of light. A total of 9000 magnets, creating a magnetic field 200,000 times stronger than the earth's own magnetic field, will make sure that the beam of particles follows the circular track. In order to do this, the magnets must be cooled down to about 2 Kelvin, approximately 300 degrees Celsius below room temperature, thereby making the LHC the coldest place in the Universe.

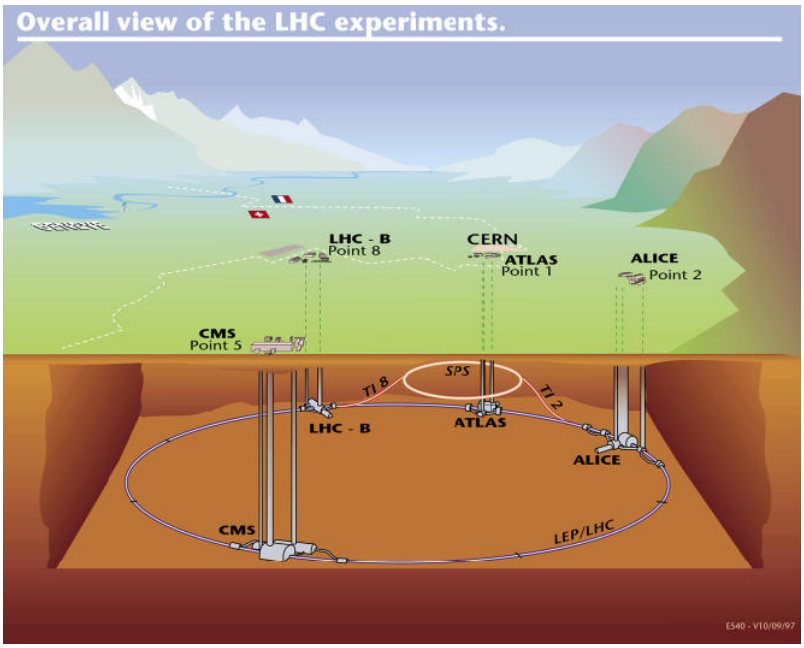

Figure 2.1: Illustration of the LHC and CERN

The LHC is being installed in the same tunnel as the previous accelerator, the LEP. Each particle will travel the 27 km ring 11,000 times per second. Located

around the ring are four detectors, or experiments, which will capture the results of the collisions. The biggest detector, ATLAS, is the size of a five story building. When the LHC is fully operational, collisions will occur one billion times a second, producing an enormous amount of data. The data created corresponds to 10,000 Britannica Encyclopedias per second. This has forced CERN to look for innovative ways for organizing the data processing. The result is the *CERN Computing Grid* which aims to utilize computer power from all over the world through the use of a multi-tiered network of computers.

Some of the questions that are hoped to be answered by the research of the data produced by the LHC includes: why particles have mass; why the mass we can observe only accounts for 4% of the total mass in the Universe; why our Universe is made of matter and not of antimatter; and what occurred during the first milliseconds of Big Bang. It is well outside the scope of this thesis to further elaborate on these questions, and for more information about the topic, I refer to the CERN web pages, www.cern.ch.

2.3 The CERN Control Center

The control of the accelerators at CERN is done from a *control center*. In the beginning of 2006, the control of all the major accelerators, including PS, SPS and LHC (see figure 2.2), were gathered in the newly built CERN Control Center. The CCC is the workplace of the *Operators*, usually consisting of physicists and technicians, who are responsible for producing and maintaining the particle beam for the physics experiments. Operators constantly monitor the quality of the produced particle beam and take corrective actions when it degrades. Similarly, in case of a failure in the highly complex accelerator infrastructure, the Operators have to make a first diagnoses on the nature of the problem, and then call the right experts for help. To accomplish this work, the operators use an *accelerator control system*, a sophisticated system with functionality to supervise the accelerator and to control it.

Figure 2.2 provides an overview of the CERN particle accelerators.

2.3.1 Accelerator Control Software

The accelerator control system consists of what is termed as *accelerator control software*. In this thesis, control software refers mainly to the software developed in the *AB/CO/AP* section at CERN. In collaboration with the *Operations*

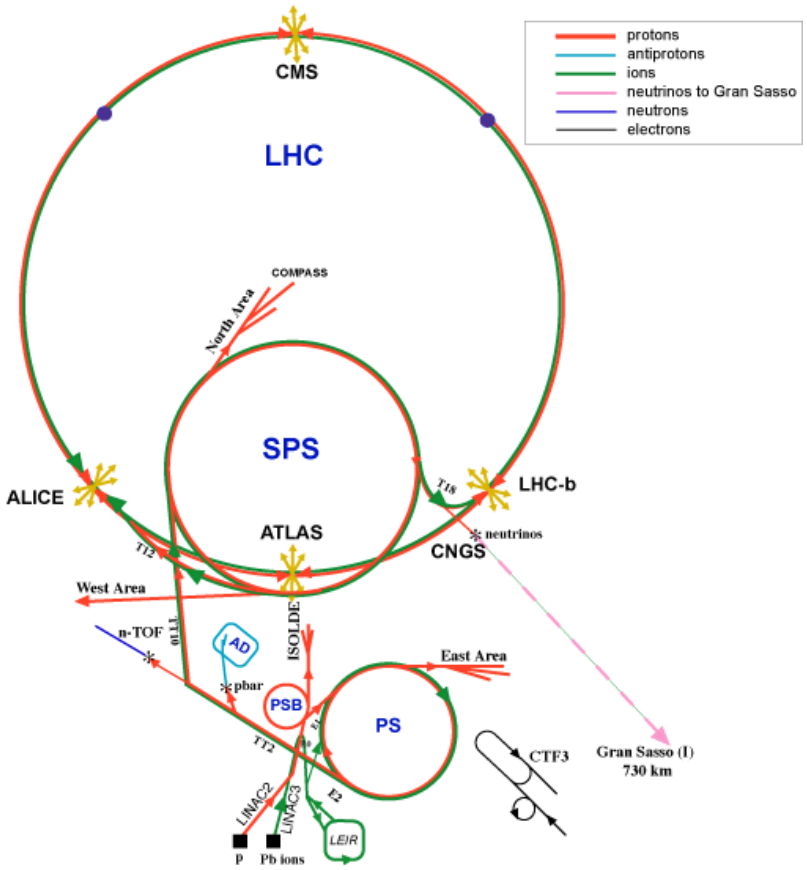

Figure 2.2: Overview of the CERN accelerators

group, this section is in charge of providing the application software for the control of PS, SPS and LHC.

Rather than relying on a few major applications, accelerator control software consists of many small interdependent products, spanning from low-level services to end-user applications. This approach allows the products to be implemented, tested and released independent of each other.

This approach does, however, demand solid conventions and guidelines for source organization and structure. The development process of accelerator control software is further explained in chapter 5.

Part II

State of the Art

Chapters

Chapter 3
Software Configuration Management

To increase the possibility of success within the area of software engineering, the ability to manage change is crucial. Large complex systems may consist of several independent components working together to solve a need. The challenges of handling changes, corrections, extensions and adaptations of such systems are the focus of *Software Configuration Management.*

The definition provided by J. Estublier in [12] states that

> *. . . SCM is the control of the evolution of complex systems*

In her article about *"Using a Configuration management tool to coordinate software development* [15], the author states three reasons why proper management of software evolution is so difficult. The *first reason*, is that the developers can very easily change code. *Second,* because of the interdependencies among components, modifications can affect entire systems, and *third,* because software is usually developed in teams, the changes of one person often makes an impact on the work of others.

This chapter takes a closer look at the concepts of Software Configuration Management. After presenting a short history of the subject, and how research has played a part in the evolution of SCM, I will present some important functionalities relevant for this report.

3.1 Evolution of Software Configuration Management

SCM as a discipline emerged in the late 70s and early 80s, soon after the so called "software crises," when it was realized that software development consisted of more than just programming. Issues like architecture, building, evolution and so on were also important, time-consuming parts of the process. The focus of SCM has changed over the years as it has tried to address the different issues of software development.

In the beginning, SCM was needed to support versioning and (re)building in managing critical software by a single person on a mainframe. It later evolved into the support of large-scale development and maintenance by groups of users on Unix systems. Today, SCM supports a multitude of software developments, often including several participants in distributed locations. Figure 3.1 illustrates the evolution of the context in which SCM works.

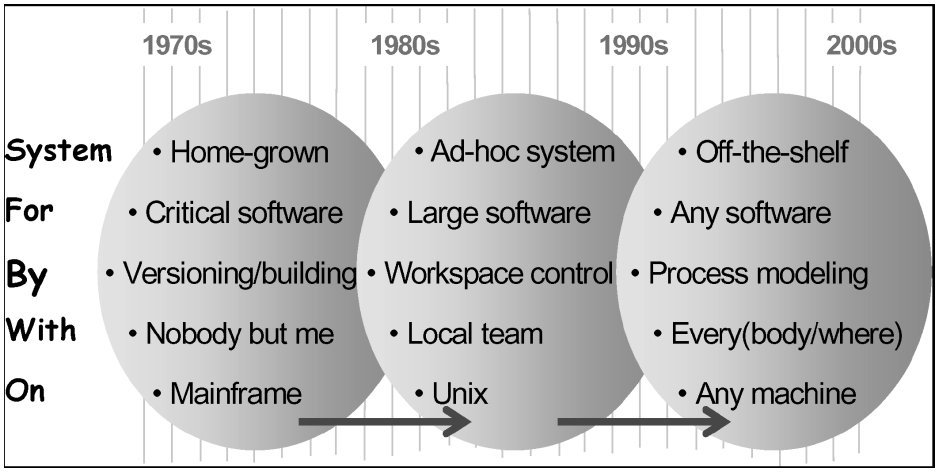

Figure 3.1: Evolution of the context of SCM systems

Today's focus of a typical SCM tool includes: **management of component repositories**, **support for the usual activities of engineers** and **process control and support** [12][13].

The evolution and success of SCM is closely related to the amount of research performed within the domain. Both industry and academia has contributed to the research which has pushed for continuous innovations in the field. Several fundamental techniques, which now form the basis of many of today's tools, were first published in one form or another. This research has contributed to why J. Estublier et al. concludes in [13] that:

> *SCM is arguably one of the most successful software engineering disciplines.*

3.2 Key Functionalities of SCM

Susan Dart presents in her article "Concepts in Configuration Management" [9] from 1991 a list of key functionalities, or operational aspects, of configuration management (See table 3.1). It has taken considerable time and effort to implement these functionalities to their full extent, and it was not until recently

that high-end SCM systems, providing good support for the whole spectrum of functionality, has emerged [13].

Functionalities	
Components	identifies, classifies, stores and accesses the components that make up the product. *Keywords: versioning, selection, consistency*
Structure	represents the architecture of the product. *Keywords: relationships, selection, consistency*
Construction	supports the construction of the product and its artifacts. *Keyword: building, dependencies*
Auditing	keeps an audit trail of the product and its process. *Keywords: traceability, logging*
Accounting	gathers statistics about the product and the process.
Controlling	controls how and when changes are made. *Keywords: change propagation, access control, change requests*
Process	supports the management of how the product evolves. *Keywords: lifecycle support*
Team	enables a project team to develop and maintain a family of products. *Keywords: workspaces*

Table 3.1: CM functionality requirements

The spectrum of functionalities covered by SCM is substantial, and the focus of this report will be on the services included in *build and release management*. In the context of this report, build and release management includes such elements as **workspace**, **versioning**, **building**, **release management**, **dependency management**, **repository management** and **change management**. The remainder of this chapter presents an introduction to these topics.

3.2.1 Workspace

During the software development process, a workspace is where the files are created and edited. It is a programmer's local environment where he can put artifacts that he needs to extend, build or test. Typically, such artifacts are checked out from a versioning system and may include source files, libraries and configuration files. Providing the right files, in the right file system and letting developers work independent of each other is the responsibility of an SCM system [4][12].

3.2.2 Versioning

Versioning is the action of assigning a specific number to a certain aggregate of artifacts for the intention of keeping a history of this aggregate. Version control systems usually include a data repository where the versioned aggregates are stored. From this repository, aggregates can be accessed through the use of the version numbers. In addition to the data contained by the artifacts, it is also a common feature to record additional information such as time of commitment, name of the author(s), the author's annotation etc [31]. Such aggregates are often referred to as configuration items and the concepts and mechanisms used to handle relations between them are termed a *version model* [7].

Versioning can appear on several levels, depending on the granularity of the aggregate. On fine-grained level, versioning keeps track of digital documents, while on a higher level, the aggregate might be a collection of components making up a software product.

3.2.3 Building

Building is the process of compiling source code and making a program runnable. The efficiency of a build process relies heavily of only rebuilding whatever is necessary. A well known and classic building tool, relied upon by many SCM tools, is Make, originally developed for the Unix platform. In recent years, Ant has emerged from the Java community offering a build tool based on the Java language. Berczuk [4] mentions two types of builds. One "private" build process, performed within a local workspace to assure the changes made, and one build process to integrate changes into a larger system.

3.2.4 Dependency management

Dependency management deals with the mechanisms of identifying dependencies of single software components, and analyze and manage their impact on an entire system [26]. Support for managing dependencies is crucial as software development shifts towards a component-based, distributed strategy. The difficulty of analyzing and tracking dependencies increases as component-based systems not only rely on in-house developed components, but also third party packages [25].

3.2.5 Release Management

Release Management is the discipline of deploying software into a software repository. It is the link between a developer and a product store allowing the developer to make his program available for other users in a controlled and consistent manner. This involves building the product, resolving possible dependencies and providing a unique identifier which can be used for accessing the product at a later point (this is typically a version number).

3.2.6 Repository management

In relation to the levels of versioning presented earlier, there are different repositories, or libraries. The level referred to as *controlled library* is used to control the current baselines of a project and manage the evolution of them [16]. A *static library*, or *software repository*, contains released aggregates of artifacts for general use. Repository mangament is the process of maintaining these libraries, providing users the possibility to store and fetch aggregates of data based on version numbers.

3.2.7 Change management

SCM is the control of changes in software projects. Joeris [17], argues that change management is one of the core problems of software development, referring to management of change process, as well as change in artifacts making up a system. Important elements addressed by the change management process include *change requests*, *problem reports*, *change logs* and *notification schemes*.

Chapter 4
Existing Tools

The aim of this chapter is to present three concrete examples of available SCM tools. It will not provide a depth-study of the mechanisms involved, but give an overall presentation of the state of practice related to some of the key functionalities mentioned in 3.2. Over the years, a number of tools have become available, both through open source communities and industry. The three tools presented here do not by any means represent all concepts available through the vast collection of tools existing, but they represent important functionality in a build and release management process. The first tool, *Gentoo* is not an SCM tool in its right definition, but it contains functionalities which are important elements of an SCM tool. Second is *Apache Maven*, a tool developed under the Apache Software Foundation [14]. It was taken into account in AB/CO/AP's decision of making an in-house solution. It was not chosen, but has inspired certain aspects of the section's solution presented later. The last tool is a prototype presented by André van der Hoek and Alexander L. Wolf [24]. They introduce a tool aimed at supporting both developers and users in the process of software release management.

4.1 Gentoo

Gentoo is an open source software for distributing and installing packages on different Unix variants. Although Gentoo is not an SCM tool per se, there are many similarities in the tasks supported by Gentoo and the tasks of a typical Configuration Tool. The key functionality of the Gentoo system is to install packages on a local computer, keep the installed packages up-to-date as new releases are made available and to uninstall existing packages. A package, which is the atomic unit of the Gentoo system is typically a third-party OSS application, like Java or Emacs.

The Gentoo package is comparable to a *computer software configuration item*, defined by [11] as:

> An aggregation of software that is designated for configuration management and treated as a single entity in the configuration

management process.

The Gentoo system's equivalent of a repository is the *Portage tree*. The portage tree is a centralized database, of which a computer running Gentoo keeps a local copy. The users copy of the portage tree is referred to as *portage*, and its functions include resolving dependencies between packages, downloading and unpacking source code, updating packages by deleting old versions and installing new. While the portage tree can be seen as a repository, containing all artifacts available, the local portage contains functionality which would be managed by an SCM's workspace tool [23].

4.2 Apache Maven

Apache Maven is a tool aimed to facilitate the build process of any Java-based project. The areas of concern that Maven tries to deal with includes:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
- Providing guidelines for best practices development
- Allowing transparent migration to new features

Maven defines a *build lifecycle*, which contains important tasks such as compilation, test, package, install and deploy [1]. The tasks in the build lifecycle are the actions that take place when a product is being built.

Maven also defines a Project Object Model, POM, which is the fundamental unit of work in Maven. It is an xml-file containing information about the project in addition to configuration details used by Maven when performing the build. Examples are the build directory, the source directory, the test-source directory.

Maven supports two types of repositories, local and remote. A local repository refers to a local copy of the remote repository including temporary build artifacts not yet released. A remote repository refers to any other type of repository accessed through a variety of protocols, such as http and file. In general, the layout of the repository is completely transparent to the user.

Dependency management is one of the areas of Maven that is best known to its users. When dealing with multi-module projects and applications, consisting of tens or hundreds of modules, Maven claim to be able to help you a great deal in

[1] for a complete definition of the build lifecycle, I refer to the Maven documentation found at http://maven.apache.org

maintaining a high degree of control and stability. After the release of version 2.0, Maven supports transitive dependencies, meaning that is no longer necessary for a user to discover and specify the libraries that your own dependencies rely on. Maven will include them automatically.

In addition Maven supports dependency scope, which is used to limit the transitivity of a dependency. A user can specify for what parts of the build, compilation, test or runtime, a dependency is applicable [19][29].

4.3 Software Release Manager

The emergence of component-based software has increased the level of complexity when it comes to the process of release management. Software is being constructed from pre-existing, independently used, independently developed and released components. Little attention has been paid to the question of how these components should be released, and how users of such components can obtain them in an effective and accurate manner. Hoek et. al. term this problem *Software Release Management*, which has given the name to the prototype they are presenting, *Software Release Manager*, SRM [24].

SRM's focus is on the activities taking place in between the moment a component is developed and when it is installed. SRM does not consider traditional SCM functionality like source code management and installation, but it instead tries to bridge the gap between authoring and releasing components and assembling such components into an application.

The Release Database is the repository in which SRM stores both metadata explaining the components, as well as the release archives themselves. SRM provides location transparency meaning that for a user of SRM, it appears as one single database, while in reality the artifacts are stored in separate repositories spread across different sites.

SRM provides its users with a Release Interface where a user might release, withdraw or modify components. Modifying a release includes modifying its metadata, or even the underlying dependencies. Withdrawing, or removing, a released component from the repository is provided to make obsolete and non-supported components unavailable for download. Only components which are not serving as dependencies for other projects might be removed.

Once released, information about a component is made available through the Retrieve Interface of SRM. This interface is the main access point to the release database and presents information about the components.

Presented in this chapter were three tools covering different aspects of build and

release management. The following chapter will present the solution used by the AP/CO/AP section for supporting their development process of accelerator control software.

Chapter 5
Build and Release management for AB/CO/AP

This chapter presents the process, and the accompanying tools which together make up the build and release management solution for the AB/CO/AP section at CERN. The tools support the development of accelerator control software and are being used by both specialized Java developers in addition to physicists developing operational software for the CERN Control Center. 5.1 introduces a short background before an elaboration of the development process is presented in 5.2. Chapter 5.3 presents the tools. The aim of this chapter is to provide a base of understanding before the presentation of the practical part of the thesis.

5.1 Background

Accelerator control software was previously written in C and C++, but a migration to Java was initiated in 2002. All new software would be written in Java, and when time and resources would allow it, old applications would be rewritten in the new language. For projects based on C and C++, there was in place a well defined and mature development process, but this did not exist for new the Java developments. The build services offered through the IDE's in use were limited and they provided no satisfactory solution to the problem of sharing JAR files between projects. To fulfill this need, it was decided to set up an easy and uniform process that should be used by all future developments. This process should be based on standards, knowledge and tools available in the Java community whenever possible. The tools to support this process were decided to be implemented in-house, thereby allowing custom functionality, and avoid the risk of implementing a tool which would be discontinued in the future. *CmmnBuild* (pronounced: common build) was initiated to support tasks such as the building process and dependency management. The *Release Tool*[1] handles all the interaction with the software repositories. Before elaborating on the tools, the development process they support is explained.

[1]To avoid confusion with release as an action, the tool is referred to as *the Release Tool* or simply *Release Tool*

5.2 The development process of accelerator control software

A software process is defined as the collection of related activities involved in the production of a software system [10]. The process of developing accelerator control software is defined as follows:

> *All activities from the moment a developer starts a new project to the moment the resulting application is running on operational consoles in the control room*

The process includes issues such as **project and source-code organization, build services, dependencies management, release management** and **application deployment.** The development process in this context does imply a certain development method. Rather, it refers to important tasks involved in the process of developing software. Following is a presentation of the key issues in the *accelerator control software development process.*

Projects and code organization

As mentioned in the introduction, the development strategy of accelerator control software implies focus on small interdependent products, implemented, built, tested and released independent of each other. For such a strategy to work, the use of solid conventions and guidelines for source organization and structure is crucial.

A project is to be given a name corresponding to where it belongs. The recommendation is to prefix each package with ”cern” followed by the name of the accelerator the software belonged to. The prefix ”accsoft” should be used for cross-accelerator software.

Source management

The source version management is handled by the CERN central Concurrent Versioning System (CVS). A dedicated repository for accelerator controls software is maintained with a well defined directory structure. A product's path within the repository corresponds to the package name explained earlier. See figure 5.1. For basic functionality like *checkout* and *commit*, the IDE's are used. More advanced functionality is integrated into CmmnBuild. One example is *tagging*. The CVS constitutes the *controlled library* for AB/CO/AP.

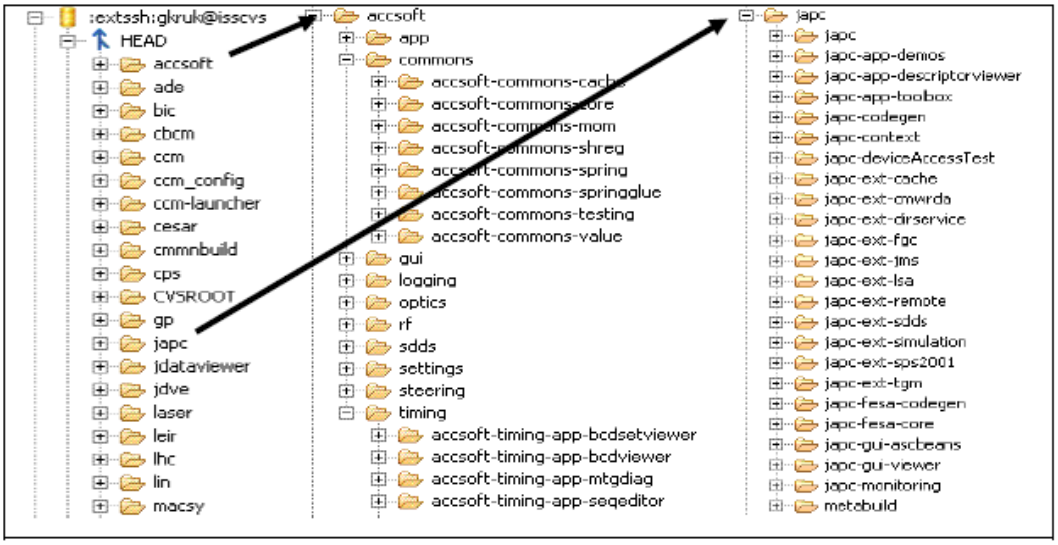

Figure 5.1: CVS repository structure for accelerator control software

Build process

Building software is the process of combining a set of configuration items, belonging to a baseline, together into composites. It serves the purpose of constructing all or parts of a product's deliverables from its components. The build services are used for *prototyping*, *testing of new functionality* and *creating a distribution* before releasing a new version of a product. The build process includes actions such as *compilation, packaging, running of unit tests* and *generation of certain files and documents.* The build process is supported by the CmmnBuild tool.

Dependency management

One of the most important factors of the section's development process is the management of dependencies. In order to simplify the dependency declaration for the user, and facilitate the maintainability of all products, it is crucial to support *transitive dependencies*. This means that a user only specifies the products he depends directly on, without worrying about the libraries his dependencies rely on. In a scenario where a low level product changes its dependency-declaration, all products depending on this product (directly or indirectly), would have to update their dependencies as well. This quickly becomes complicated, especially in environments with many products and complicated dependency-trees.

In addition, the dependency management must take into account that products

might be stored in different repositories. AB/CO/AP separates its own developments from third party tools in two separate locations. CmmnBuild handles the management of dependencies and fetches both direct and indirect dependencies from either of the two product stores. Like all three tools mentioned in chapter 4, CmmnBuild handles the different repositories transparent to the user.

Release management

Release management is the process of installing a built and tested product into the operational distribution area, referred to as the *production repository*. All deliverables are installed and distributed on their corresponding platforms, where they are made available for use in operation. The products are available through an assigned version number. Previous operational versions stay available. The release management is handled by the Release Tool.

Software deployment

The deployment of GUI applications is done using Java Web Start. To enable the use of JaWS, a web server has been installed which makes the production and the third-party repositories accessible through the web. To run an application, all that is needed is the URL to the product's descriptor file (JNLP) residing in the repository. It is possible to run any released version of the product. The JNLP files are created by CmmnBuild during the build and made available on the server by Release Tool. Also available is the generation of a command script for launching an application.

Presented here were the most important aspects of the development process, with an indication of which services is provided by which tool. Following is a detailed explanation of the two tools.

5.3 The build and release tools

This section presents the tools supporting the development process explained in the previous chapter. First is a closer look at the operational environment in which the tools operate. Second is an elaboration of why and how the tools have been developed. The last section explains the technical aspects of the tools, how they interact and what are the services they provide.

5.3.1 Operational aspects

The new tools supporting the development process were meant to support the development of all accelerator control software for the use in the Cern Control Center (CCC). The developers of such software vary in background and software development experience.

User Characteristics

The users of CmmnBuild and Release Tools consists of both specialized Java developers and physicists developing operational software. CmmnBuild and Release Tool have an estimated total of 80 users. This is mostly members of the AB/CO/AP and the AB/OP section at CERN, but it also includes developers in other research institutions developing software for the CERN Control Center. The main example is Fermilab in the USA [2].

In more detail, the users are:

- software developers and project managers who deliver software for the operation and control of the CCC
- people who maintain operational software which is not necessarily developed by them
- the software administrator who will be in charge of the administration of the SCM system
- members of the AB operations group, AB/OP. They are responsible for the operation of the different accelerators, and they also develop a large amount of operational software

Operational environment

CmmnBuild is running on a user's local workstation and must support both Windows and Linux. Release Tool is running on a server and must stay independent of network file system (NFS, AFS, etc.).

[2] http://www.fnal.gov/

5.3.2 Development of CmmnBuild and Release

CmmnBuild and the Release Tool are two separate and independent tools. Both of them can be used without the use of the other and the development of the two has previously been done separately. They are now combined to one project. The next two sections explains the choice of development for the tools.

CmmnBuild

Before starting the development of an in-house tool to solve the need for configuration management of accelerator control software, several tools in the Java community were examined and considered. However, they were all found to be in an early stage of development, with limited functionality, especially regarding dependency management. They were not considered mature enough, and there was no guaranty that these tools would be continued into the future. In addition, no satisfactory solution to the incorporation of 3rd party libraries was found. Considering these facts, it was decided to develop an in-house, custom solution. This would guaranty the section the necessary functionality. It would cost time and resources, but so would customization of an external tool.

The first production version of CmmnBuild was deployed in 2003 with a minimum of functionality. Since then, it has been incrementally developed and new functionality has been added when needed.

CmmnBuild is still under development as new requirements from the users appear.

Release Tool

The Release Tool already existed at the startup of CmmnBuild, though it was not created to support Java-projects. As CmmnBuild evolved and was taken into use by all Java-developments, it was decided to rewrite the Release Tool to better suit the needs of the new language. It was previously based on bash script aimed for C and C++, and modifying this to accommodate the changes were considered more work than to rewrite the tool in Ant. Ant is specifically aimed at supporting Java.

The Release Tool is still under development as new requirements from the users appear.

5.3.3 Technical aspects and functionality

This section takes a look at the technical aspects of the tools, including technical details, functionalities and the constraints of the system.

Client-server design

The relationship between CmmnBuild and Release Tool is based on a client-server model. The tools are, however, not tightly linked and CmmnBuild is just one of several clients using the Release Tool. Other clients are not considered here. Figure 5.2 provides an overall view of the two tools and the interaction between them.

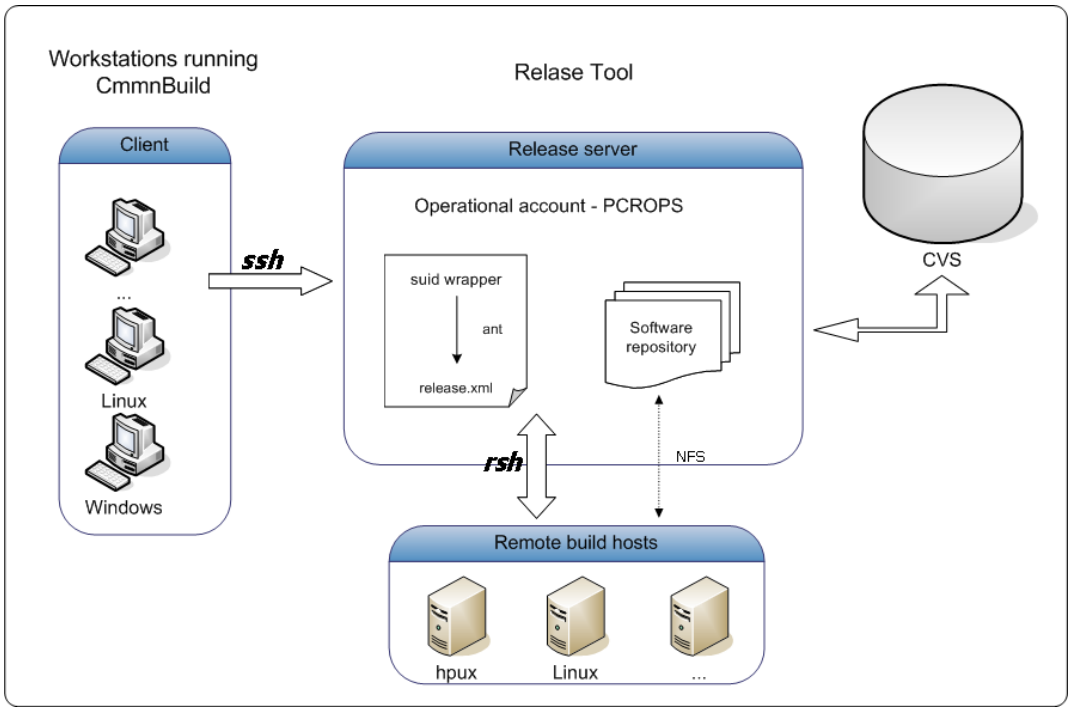

Figure 5.2: Overview of CmmnBuild and Release Tool

Apache Ant

Both tools are based on the Java-based scripting language Ant, stemming from the Apache Software Foundation. A short introduction to the use of Ant in CmmnBuild and Release Tool is presented before elaborating on the technical aspects.

The Ant package provides a set of tasks representing different functionality. Example of such tasks are *javac*, *mkdir* and *echo* [2]. In addition to these core tasks, there are several additional tasks available through external packages easily included into the local Ant installation. Examples being the *ant-contrib* [21] and the *xmltask* [20]. Even more specialized tasks can be written if needed. All underlying logic of Apache Ant is written in the Java language, and extra functionality may therefore easily be written in Java. Most of the underlying logic of the services performed through CmmnBuild and the Release Tool is provided by standard Ant packages or extensions such as xmltask and ant-contrib. In addition, some specialized features are implemented in Java. These are further explained when needed.

Technical details of CmmnBuild

CmmnBuild consists of a set of ant-scripts divided into several files based on their functionality. The main file is the *common-build.xml*. Through these scripts CmmnBuild can provide its services through a set of predefined targets. CmmnBuild offers no graphical user interface to its users, and the targets may be run from command line or through an IDE. This requires an Ant plugin. CmmnBuild includes three main areas of support: creating a *distribution* of a product, *fetching dependencies* and *releasing* the product.

Not all targets are available for direct execution. Some are incorporated into more substantial services, and may be enabled/disabled through the use of properties on a project level. The properties are explained later.

Table 5.1 presents a list of the most important target available for direct execution, but a complete list may be found in Appendix A.

Targets	Action
usage	prints all targets available for the user
compile	compiles all source code
dist	creates a complete distribution of the product
getjars	fetches the dependencies specified from the repository
release	releases the product into production; involves Release Tool
devrelease	releases the product to the Qfix folder
junit	executes all unit tests written for the project

Table 5.1: A selection of CmmnBuild targets

The *dist* target is the main task of CmmnBuild. It performs all tasks which are needed to create a full distribution of a product. The resulting output depends on the properties specified for the project. Included in the dist service are:

- Fetching of dependencies
- Compilation
- Packaging to jar or war file
- Certain code generation
- Generation of Javadoc and java2html
- Style checking
- Compilation and execution of JUnit tests
- Generation of JNLP or command script

Through command line, the command:

```
ant dist
```

would execute the *dist* target.

CmmnBuild is a centralized tool, meaning that it prevents downloading a big buildfile for every new project. An important goal for CmmnBuild is to keep the overhead of using it as little as possible. That is why all that is needed is a small buildfile working as a proxy to the targets defined in the user's installation of CmmnBuild. In addition, the user must adapt a certain directory structure which includes four special configuration files (the buildfile included). See figure 5.3.

```
example-project/
  build.xml
  product.properties
  product.xml
  people
  src/
    java/
    test/
```

Figure 5.3: The directory structure imposed by CmmnBuild

Configuration files

The *build.xml* file is a regular Ant build file that imports the target from the CmmnBuild-distribution installed on the user's computer. This build file is the

same for every product (figure 5.4). Advanced users may extend this file with custom targets.

```
1  <?xml version="1.0"?>
2  <project  name="example-project" default="default">
3    <!--
4       Defines project variables
5    -->
6    <property file="project.properties"/>
7    <!--
8       import the standard target from cmmnbuild
9    -->
10   <property environment="env"/>
11   <property name="cmmnbuild.home"  value="${env.CMMNBUILD_HOME}"/>
12   <property name="cmmnbuild.targets"  location="${cmmnbuild.home}/targets.xml"/>
13   <import file="${cmmnbuild.targets}"/>
14
15 </project>
```

Figure 5.4: The build.xml file

The *product.xml* file is a descriptor file for the product. It contains the name, the version, CVS module name and may also contain a description, a webpage link and a set of dependencies. Product.xml is relied upon by any target in need of metadata about a product. When specifying the dependencies in the product.xml, the user can either specify a certain version number of the product to depend on, or choose to depend on the *production version* (figure 5.5). The production version is explained later.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <products>
3    <product name="example-project" version="3.0.2" directory="example-project">
4          <jar>example-project.jar</jar>
5          <desc></desc>
6          <href/>
7          <dependencies>
8       <dep product="xalan" version="2.7.0"/>
9       <dep product="cern-jjar" version="PRO" />
10         </dependencies>
11   </product>
12 </products>
```

Figure 5.5: An example of the product.xml file

The *project.properties* file specifies properties unique to a project and is used to activate specific services in CmmnBuild during the build process (figure 5.6). An example is enabling the generation of javadoc by setting the property *javadoc.enabled=true*.

The last file is the *people* file, indicating the names of the users who have the right to release the product. It is used during the release process (figure 5.7).

```
1 javadoc.enabled=true
2 java2html.enabled=false
3 war.enabled=true
```

Figure 5.6: An example of the project.properties file

```
1 gkruk
2 wsliwins
3 enes
4 vfram
5 eroux
```

Figure 5.7: An example of the people file

Technical details of the Release Tool

The Release Tool is organized similarly to CmmnBuild with a set of ant-scripts containing targets available through the main file *release.xml*. The release management is handled by the Release Tool. This tool is not exclusively for Java projects, and can also be used for C and C++, depending on the tool specified for building the product. The focus of this report is on Java projects using CmmnBuild. In this case, Release Tool is not used directly by the user, but is called from CmmnBuild when running the `ant release` command. Before CmmnBuild invokes the Release Tool, it tags the the collection of the artifacts making up the product in the CVS with a version tag.

The release tool is running on the *release server*, in an operational account referred to as *PCROPS*. In addition to hosting the Release Tool, PCROPS also contains two software repositories, *production repository* and *3rd party repository*. Production repository contains all released products of accelerator control software developed in-house. The 3rd party repository contains third party libraries used by developers in their projects. The 3rd party repository resembles the Gentoo Portage Tree in the way that it stores and maintains external software. Example of 3rd party libraries are *JUnit* and *Log4J*. The release tool only supports releases to the production repository. Information about the content of the repositories is contained in xml files, referred to as *repository.xml*. This is further explained later.

When a product is released, the Release Tool extracts this product from the CVS to the dedicated production repository and builds the product. The product is extracted from the CVS based on a CVS version tag assigned in the first step of the release process. In case of Java products, the release will call its own local CmmnBuild installation to build and prepare the product. To maintain the loose coupling between the Release Tool and CmmnBuild, a different build or make tool can easily be used instead.

Figure 5.8 illustrates how a product in the production repository is structured. The directory structure corresponds to the structure in the CVS repository. For every new version, the Release Tool creates a separate directory, taking the version as name, without modifying old versions. This leaves the opportunity of going back to an older version at a later point.

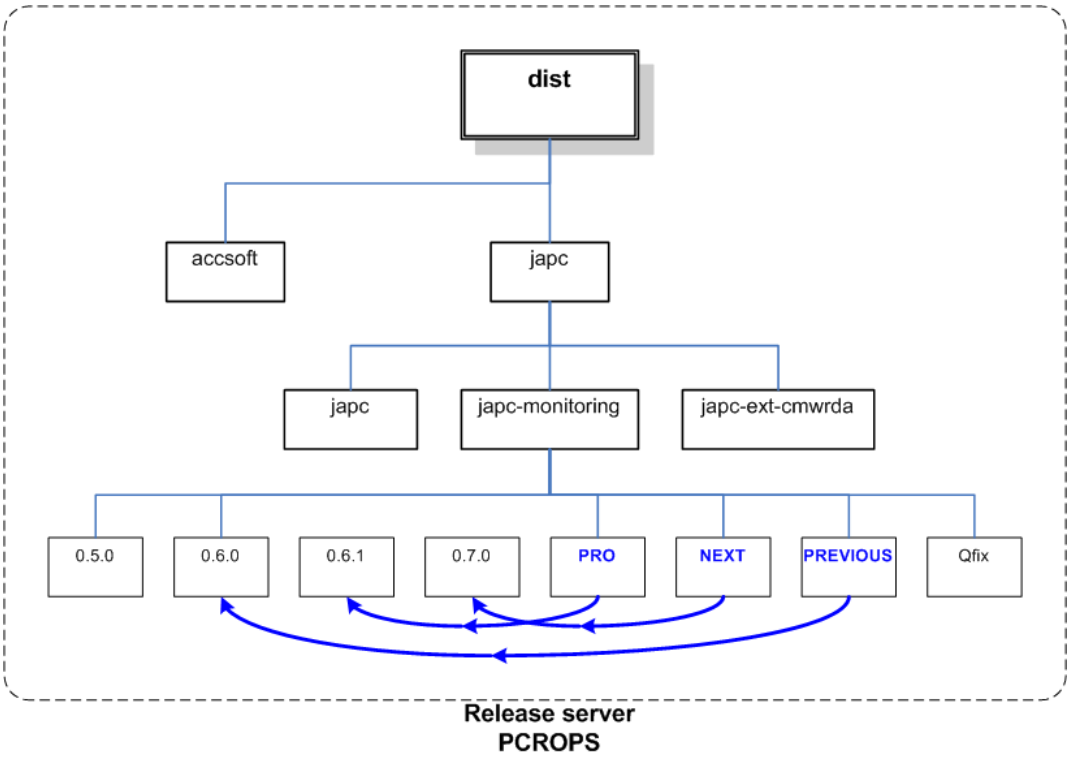

Figure 5.8: Structure of the distribution area

To indicate which version is the current *production version*, the the Release Tool maintains a set of symbolic links, referred to as aliases:

PRO - points to the current production version

NEXT - points to the next version that may become production version

PREV - points to the previous production version (version replaced by PRO)

The alias of which a product shall be released with, is specified in the command line when launching the release process. As default, a product when released, becomes the production version, but there is the possibility to release a product as the *next version* (pointed by the NEXT alias). This feature may be used to evaluate a release in its real environment before releasing it as the production version. The PREV alias allows the user to easily revert to the previous version in case the newly released production version does not work as planned. The

following command executed in the *example-product* folder, would release the product as NEXT

```
ant release -Dalias=NEXT
```

In addition to a normal release, there is the possibility of releasing a product to a folder called *Qfix.* The motivation for this possibility is to supply the developers with a place to quickly release a bug fix or untested functionality. The Qfix folder has full write access which means that a developer can modify his code immediately, even without being at his usual workstation. Products released to Qfix are not related to any version number and will be overwritten at next Qfix-release. The Qfix can be affiliated with any of the aliases.

The repository.xml file

The information about the content of a repository is kept in an xml file named *repository.xml* (also referred to as *repository descriptor*). Both 3rd party and the production repository keep such a file mirroring their content. These files are the source of information when interacting with the repositories. CmmnBuild uses them to resolve and fetch dependencies, and the Release Tool uses the repository.xml of the production repository when installing a new version of a product. In addition, these files are available, through the use of *XSLT*, for a user to view via a web-page. This is similar to the *Release Interface* in SRM from chapter 4.3, but differs by the fact that the repository web interface is only for viewing. Actions are performed through ant commands. Consistency between the repositories and their respective xml files mirroring their content is crucial for the operation of CmmnBuild and Relaese. Figure 5.9 shows a section of the repository xml in its normal appearance, while figure 5.10 shows the view available for a user through a web-browser. For the remainder of this report, when referring to the repository.xml, its referred to the file mirroring the *production repository.*

This chapter has presented the development process implemented in the AB/CO/AP section, and the tools supporting it. The following chapter will summarize the functionality available, and introduce the areas in which new functionality is desired.

```
<product directory="accsoft/rf/accsoft-rf-logging" link="PRO"
  name="accsoft-rf-logging" version="0.7.1">
  <jar>0.7.1/build/dist/accsoft-rf-logging.jar</jar>
  <desc>Logging Application. Store CMW iterfaces data into local files and
    memory, read it and show it in plots.</desc>
  <dependencies>
    <dep local="true" product="commons-logging" version="1.0.4"/>
    <dep product="japc" version="0.14.13"/>
    <dep product="japc-context" version="0.9.5"/>
    <dep product="japc-ext-dirservice" version="1.1.8"/>
    <dep product="japc-gui-viewer" version="1.0.11"/>
    <dep local="true" product="log4j" version="1.2.9"/>
  </dependencies>
  <releaseDate>Monday, November 27, 2006</releaseDate>
</product>
<product directory="cmmnbuild/cern-jjar" name="cern-jjar" version="3.5.3">
  <jar>3.5.3/build/dist/cern-jjar.jar</jar>
  <desc>CERN JJar facility</desc>
  <href>http://cern.ch/test-release</href>
  <dependencies>
    <dep local="true" product="ant" version="1.6.5"/>
    <dep local="true" product="xalan" version="2.7.0"/>
  </dependencies>
  <releaseDate>Monday, November 27, 2006</releaseDate>
</product>
```

Figure 5.9: The repository.xml file

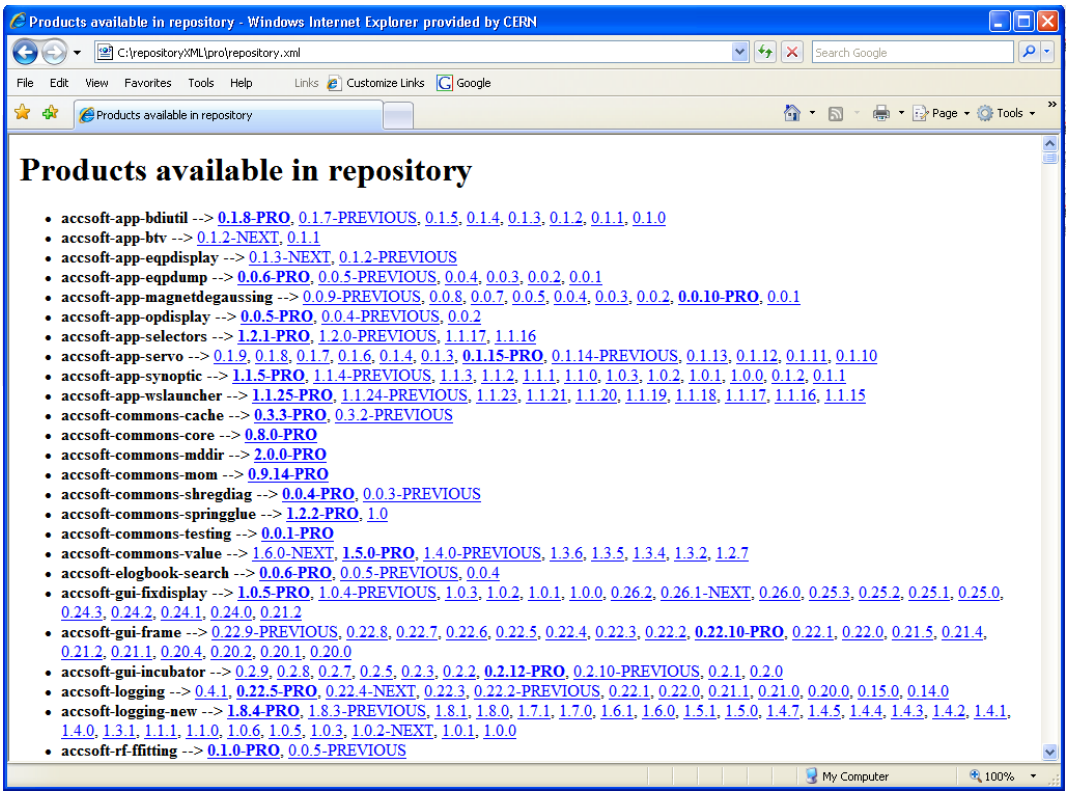

Figure 5.10: The repository user interface

Part III
Own Contribution

Chapters

Chapter 6
Problem Elaboration

This chapter sums up the current state of the two tools, CmmnBuild and Release Tool, referring to the state before the practical part of this thesis was started. What services were already supported, and what were the areas in which new functionalities were desired. Furthermore this chapter presents the problem formulation for this thesis. What were the goals to achieve. To improve the software development process implemented in the section, the focus of this project has been to implement and deploy desired extensions to the group's build and release management solution. The goal of each extension has been to develop a functional feature, and making this available for users in the newest versions of the tools.

6.1 Current state

Both CmmnBuild and Release have been operational for several years, and many projects developing control software for different accelerators have been entirely based on these tools. The tools have provided support spanning from the startup of a new project, or a new version, to the installation of that product in the production repository. This included all steps in the development process presented in chapter 5.2. Table 6.1 summarize the current services available in CmmnBuild and Release Tool. There are, however still services not supported by the tools and three of them were addressed during the work of this thesis:

The strategy taken by AP/CO/AP when it comes to maintaining released versions is simple; a released version is never patched or fixed in responds to a bug-report or similar. Instead, a new version is released with the proper changes. This might cause a problem if the version with the bug is old, and the project is in such a state that a bugfix cannot be provided any time soon in the newest release. The CVS provides a service called *branching* to allow a developer to create a parallel development path of a project based on a version tag [6]. *Incorporating the branching functionality into CmmnBuild was the first problem addressed.*

The production repository contains all products developed for accelerator

control in the AP/CO/AP section. With the startup of the LHC closing in, and the installation of the new control center, much software is produced and the amount of software kept in the repository increases continuously. To keep the repository in an easily maintainable state, *functionality to remove unnecessary products was wanted*.

The AP/CO/AP section relies on JIRA, which is a third party, issue-tracking, project management application [3]. It is used by developers and users to register issues about a product. Such issues can consist of bug-reports, wanted functionality or a change log. Currently, this tool works independently of CmmnBuild and the Release Tool. An overall goal is to merge the tools in order to support the full product lifecycle. As a start, *an automatic release notification scheme was looked into*.

These issues led to the following problem definition.

6.2 Problem formulation

Since the startup of this project, AB/CO/AP has undertaken an iterative, incremental development approach towards CmmnBuild and Release Tool. This project represents three such increments responding to the problems presented in the previous section. The overall goals for the increments are shown in table 6.2. The purpose of this thesis has been to implement support for each of the goals presented, and make this available in new versions of the tools. An elaboration on the issues and specific requirements for each goal are presented in chapter 8.

Before the decision was made to develop an in-house solution for build and release management, several external tools were considered. One reason for the final decision was that the tools were found immature, unstable and lacking key functionality. Today, however, many of these tools have grown to become widely used SCM tools, and the question has been asked whether it might be beneficiary for this section to switch one of these tools. A final goal of this thesis is therefore:

Based on the experience of the last 6 months improving the SCM solution, could it be beneficiary to migrate to an external tool to solve the sections SCM needs?

The goal is not to give a final answer to such a question, but to shed some light on the subject which might be of value in an upcoming discussion.

Current state of CmmnBuild and Release Tool	
Subject	**Functionality**
Code organization and Source Management (CmmnBuild)	A certain *file structure* is imposed by CmmnBuild. For *source management*, CmmnBuild relies on CVS. Certain CVS actions are available through CmmnBuild, like *tagging*, but for simple interactions, such as *check in/check out*, the developers use their IDE or command line
Build services (CmmnBuild)	Provides support for *compiling, generation of code, compilation and running of unit tests and verification of code.* Packaging to *jar* and *war* files is possible. In addition, generation of *javadoc, jnlp* and *html* is available. Some of these are available as stand-alone services. All are incorporated in the creation of a *distribution* and may be enabled/disabled in the projects properties declaration. Some of these services rely on customized Java packages, while the rest are based on the functionality provided through Ant. One exception is *code generation*, which is a customized target relying on *XSLT*
Dependency management (CmmnBuild)	Provides the service of *fetching the dependencies* of a product from either the *production repository* or the *3rd party repository*. This happens transparent to the user. This service is made available through a customized Ant task developed in Java
Release management (CmmnBuild & Release Tool)	*Builds and installs* a product in the production repository. Aliases may be specified, to decide whether or not to release as production version
Software deployment (CmmnBuild & Release Tool)	A *JNLP file* may be generated during the build process and made available on a web server during the release process. Java Webstart is used to run the application, based on this JNLP. Also available is the generation of a *command script* for launching the application

Table 6.1: Current state of CmmnBuild and Release Tool

Desired extensions to CmmnBuild and Release Tool	
Subject	**Goals**
Incorporation of branching	incorporation of the CVS branching feature into CmmnBuild facilitating the process of providing a bugfix
Repository maintenance	provide functionality to limit the amount of manual labor necessary to keep the repository-server in an easily maintainable state
Release notification	provide support for automatic release notification. A first step in a more extensive solution

Table 6.2: Goals for the desired increments

Chapter 7
Research Method

This thesis consists of two parts. Presented in Part II is the current state of Software Configuration Management with emphasize on key functionality for build and release management. Part III is a detailed explanation of the implementation of a set of selected features in an existing build and release tool used by the AB/CO/AP section at CERN.

Figure 7.1 illustrates the overall work plan of the project. The two parts are further explained in the following section.

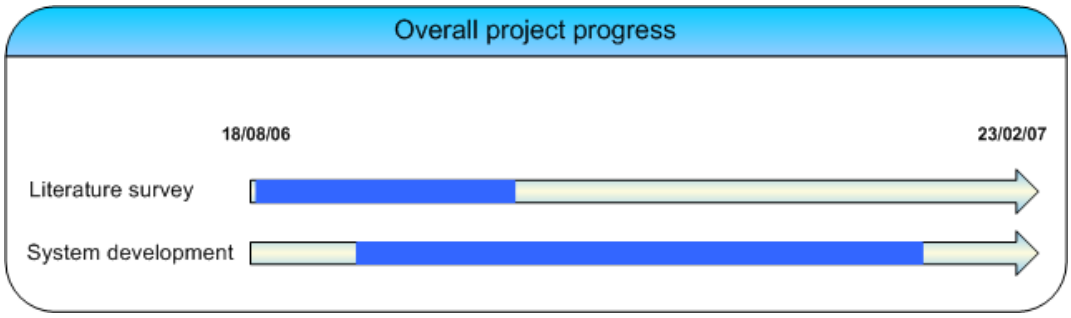

Figure 7.1: Overall progress of the project

7.1 Literature survey

During the first part of the thesis, a literature survey was conducted to get an overview of the concepts of software configuration management. In addition, important features and functionalities of build and release management were explored through studying a set of external tools in addition to tools presented in this thesis.

The search for articles was carried out on search-engines covering databases and on-line article repositories accessed through licenses of NTNU. The search was limited to a set of well known repositories, mainly IEEE and ACM. In addition NTNU and CERN's document servers were used.

7.2 System development

The development process for the practical part of this thesis was never formally defined. Since the project consisted of developing independent extensions to the tools, it was natural to assume an iterative, incremental strategy on the overall level. The overall level referring to the development of each new feature.

There are several approaches to software development depending on the type of project, the size and location of the development team, customers etc. Three well known approaches are the *waterfall model* [22], *iterative development* [18] and *agile methods* [1]. The methods are often recognized by their abilities to adapt in changing environments. The waterfall method is the most predictive of the three, while agile methods are considered most adaptive. A mixture of both may be found in iterative development.

Agile methods emphasize real-time communication over written documents, with face-to-face communication as the preferred way. Development using agile methods consists of dividing work into small iterations, each iteration often representing a mini-increment of the software [30].

The development team has mainly consisted of two people, the author of this report, responsible for the implementation, and a project leader, responsible for the tools in total, and acting as an advisor on implementation issues. In addition to working close to our ”customers,” which are the developers in our section, we also represent the customers ourselves. Because of these factors, it was natural to adapt an agile development strategy based on face-to-face communication, both within the team and with customers. To be able to react fast in change of requirements, each extension implemented consisted in several iterations which were planned, implemented and tested along the development path. Figure 7.2 illustrates the development process.

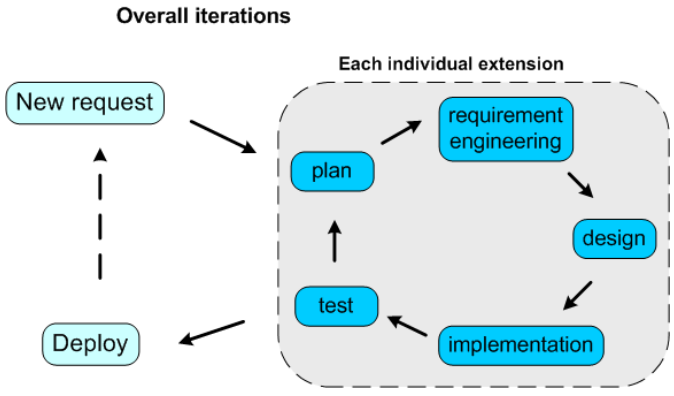

Figure 7.2: Illustration of the development process

Chapter 8
Extending the build and release management solution

This chapter presents the practical work of this thesis. Included here is the implementation of the three new features of the build and release management solution presented in chapter 6.

Section 8.1 elaborates on the incorporation of branching into CmmnBuild. It was the first extension implemented. The branching feature does not represent a grand implementation and served therefore as a good introduction to the tool. As for all three implementations, scenarios and specific requirements are presented in their respective section.

The most substantial extension implemented was the repository maintenance, section 8.2. The feature was to be incorporated into the *cern-jjar* package, containing logic for interaction with the repositories. The package was already used to fetch dependencies and to update the descriptor file for the production repostitory during release. The functionality for updating of the descriptor file was rewritten during this implementation. The old solution of this function is not an object of focus in the elaboration of the implementation.

Last is the implementation of the release notification scheme. Automatic mail creation is a type of feature which is not always welcome, as it might lead to heavy traffic in the inbox of programmers involved in many projects. It is therefore not only important to offer developers means to choose only certain projects for subscription, but also provide them a way to stop the notifications.

Chapter 9 evaluates the process and the result of all three sections.

8.1 Branching

AB/CO/AP relies on CVS for storing and sharing code. One of the features provided by the CVS is called *branching*. Branching provides the functionality of creating a parallel programming path, independent of the main branch (figure 8.1) [6]. This feature of the CVS can easily be used manually through command line or an IDE (in our case, Eclipse), but incorporation of this feature into CmmnBuild was desired, thereby making sure it is done in a consistent and standardized way.

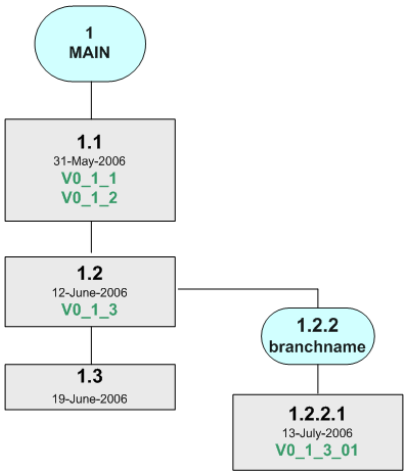

Figure 8.1: Illustration of a CVS branch with the name *Branchname*

When a version of a product is released and installed in the production repository, that particular version is ”out of the hands” of the developer. A specific version is not maintained and updated in the repository. New versions may be released, and old versions may be deleted, but apart from that, a released version is static. This means that a version is never patched to suit the need of a user, even if it contains a bug. Instead, a new version is released. A specific scenario for the use of the branching-feature is further explained in the following section.

8.1.1 Branching scenarios

The main scenario for this functionality was bug fixing. A concrete example follows.

Fixing a bug in an old version

Frank, responsible for a large low-level application, receives a call from an operator reporting a bug in Franks software. The operator himself was developing a new feature to a program, when he realized that there was a bug in one of his dependencies. He tells Frank that he is depending on version 3.0.2. This version, which dates six months back, was the last version released by Frank before a major change of the application was initiated. The feature desired by the operator was not available in the version before, and has not yet reappeared after the restructuring. To avoid going back on his new changes to fix this one bug, Frank creates a branch at the point the version which the operator is using. By creating a branch, he will be able to fix this bug in an independent and parallel programming path. This way, he can provide the operator with a working product until he will release his new and improved version.

8.1.2 Requirements for Branching

A branch must be based on the version of a product. To minimize the manual actions necessary, a checkout of the new branch should automatically be performed after the branch-creation.

Default values should be provided for both branch tag name and check-out folder, but users should also be allowed to specify this themselves. No default can be provided for the version number to branch.

The following functional requirements were defined.

Functional requirements

- **F-B1** - The feature must provide a means to create a branch at a given version
- **F-B2** - The new branch should be automatically checked out from the CVS after creation.
- **F-B3** - Must provide the possibility to specify both the name of the branch and the target directory of the checkout. If not, default values will be used.

8.1.3 System design - Branching

The branching feature is created as extension to CmmnBuild and consists purely of ant-script. The functionality is placed in the script which is also responsible for releasing a product. The branch command is made available as a direct target for the user. The parameters used to perform the branch and the following checkout; product version, branch name and checkout folder, can be supplied by the user as system variables in the command (ex: -Dversion=1.2.1). If no version is specified, a popup box will appear for the user to provide it. For the other properties, default values are used in case they are not specified.

Figure 8.2 shows the flow of events when the branch-command is executed.

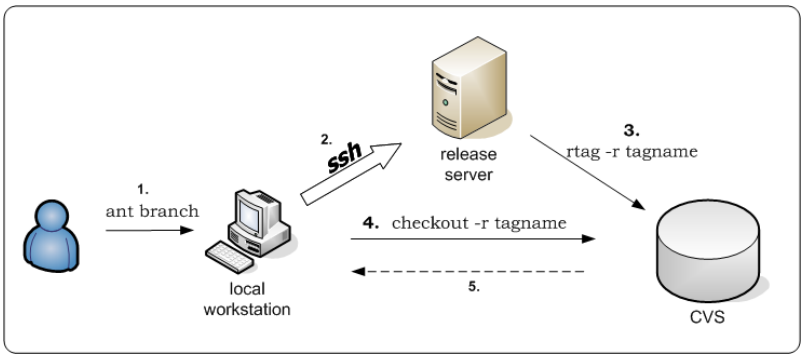

Figure 8.2: Flow of events for Branching

1. User executes the branch-command, and gets prompted for version to branch and password
2. CmmnBuild connects the user to release server using SSH
3. The server tags the desired version in the CVS with a branch tag
4. Control is returned to CmmnBuild, which checks out the branch that was just created.
5. The product is installed in the specified, or default, directory.

8.1.4 Implementation - Branching

Because of security reasons, any CVS write-operations performed by CmmnBuild are executed through a server using Kerberos, an authentication scheme based on symmetric key cryptography. This is to avoid installing the

necessary software on every machine using CmmnBuild. The checkout is a read-operation and is executed from the local machine.

Following is the implementation of the Branching. It is based purely on standard Ant which provides functionality for both SSH and CVS commands (lines 56 and 76 respectively)

```
1  <!--
2    create branch
3  -->
4  <target
5    name="branch"
6    depends="properties-init, readUsername, readPassword, branch-impl,
7      branch-checkout-impl, update-product.xml-impl"
8    description="Creates a branch at a specified version" />
9
10 <target name="read-branch-version" unless="branch.version">
11   <taskdef name="query" classname="cern.cmmnbuild.ant.Query"
12     classpathref="antext.classpath"/>
13   <query name="branch.version"
14     message="Enter the versionnumber where you want to branch"
15     password="false" />
16 </target>
17
18 <target name="recreate-branch-version-tag">
19   <property name="branch.version.temp" value="${branch.version}"/>
20   <propertyregex property="branch.version.temp" input="${branch.version}"
21     regexp="\." replace="_" override="true"/>
22   <property name="branch.version.tag"
23     value="${release.cvs.tag.prepend}${branch.version.temp}"/>
24 </target>
25
26 <target name="generate-branch-tagname" unless="branch.tagname">
27   <property name="branch.tagname" value="${branch.version.tag}_fix" />
28 </target>
29
30 <target name="generate-branch-dir" unless="branch.dir" >
31   <property name="branch.dir" value="${product.name}_${branch.tagname}"/>
32 </target>
33
34 <target name="branch-init" depends="read-branch-version,
35   recreate-branch-version-tag, generate-branch-tagname, generate-branch-dir">
36   <echo>
37   branch.version=${branch.version}
38   branch.tagname=${branch.tagname}
39   branch.directory=${branch.dir}
40   </echo>
41 </target>
42
43 <target name="branch-impl" depends="branch-init">
44 <!--
45   create branch
46 -->
47   <property name="branch.command"
48     value="cvs -d :kserver:${release.cvs.host}:${release.cvs.rootdir}/
49     ${release.cvs.repositoryname} rtag -r ${branch.version.tag}
50     -b ${branch.tagname} ${product.directory}" />
51   <echo>
52   Branching ${product.name}, version: ${branch.version} with ${branch.tagname}
53   Command: ${branch.command}
54   </echo>
55
```

```
56    <sshexec host="${release.hostname}" username="${username}"
57       password="${password}" command="${branch.command}"
58       trust="yes" failonerror="true"/>
59  </target>
60
61
62  <!--
63     Checkout branch
64  -->
65    <target name="branch-checkout-impl" >
66      <property name="branch.checkout.command" value="cvs" />
67      <property name="branch.checkout.cvsroot"
68         value=":pserver:anonymous:@${release.cvs.host}:
69         ${release.cvs.rootdir}/${release.cvs.repositoryname}" />
70      <echo>
71     Checking out ${branch.tagname} into ${basedir}/../${branch.dir}
72     Command: ${branch.checkout.command} -d ${branch.checkout.cvsroot}
73     co -r ${branch.tagname} -d ${branch.dir} ${product.directory}
74      </echo>
75
76    <cvs cvsRoot="${branch.checkout.cvsroot}" dest="${basedir}/.." >
77      <commandline>
78        <argument line="-d" />
79        <argument line="${branch.checkout.cvsroot}" />
80        <argument line="co" />
81        <argument line="-r" />
82        <argument line="${branch.tagname}" />
83        <argument line="-d" />
84        <argument line="${branch.dir}" />
85        <argument line="${product.directory}" />
86      </commandline>
87    </cvs>
88  </target>
```

8.2 Maintaining the production repository

All operational software developed for the CCC is contained in the production repository. Whenever a release is executed, a new version is built and installed on this server in the correct place. The problem of any repository is the lack of infinite storage space, and every release is place consuming. More important, a repository growing uncontrolled is harder to maintain. There has never been in place an automated process to keep the repository in a maintainable state by removing obsolete products. This has been done manually by the administrators of the Release Tool. The process includes both removing the physical files from the server, and updating the file mirroring the repository, repository.xml. As the xml file grows, the chance of human error during editing increases.

In the current state of the CCC, new software is being developed and tested continuously. Table 8.1 shows the increase in artifacts stored in the production repository during the last year, and this growth is likely to continue as the LHC starts its test by the end of 2007. The growth further is illustrated in figure 8.3.

Repository data			
Month	**Data1**	**Data2**	**Data3**
March	204	478	7591
May	207	913	12756
August	242	1221	15534
November	260	2046	27051

Table 8.1: **Data1** is to the *number of products* in the repository, **Data2** is the total *number of artifacts* and **Data3** presents the *size of the repository descriptor* (repository.xml) in lines (of xml-code)

An automated operation to remove unnecessary products from the repository at every release was wanted. This would reduce manual labor of keeping the number of artifacts from growing uncontrolled, and decrease the possibility of leaving the repository and the mirroring file in an inconsistent, unmaintainable state. The following definitions were put in place:

Obsolete version - a version which is not depended on by any other products, and that is not tied to an alias.

Number of versions to keep - the number of obsolete products the user wants to keep from being removed from the repository

The release of a product should only clean obsolete versions of that specific product. Obsolete versions of other products should remain unchanged. The process, hereby referred to as *AutoCleanUp* resembles the functionality referred to as *withdrawing a release* in SRM [24].

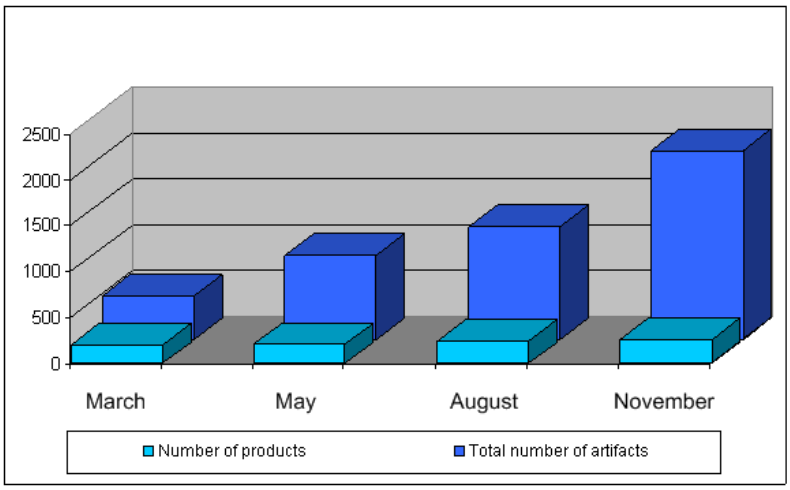

Figure 8.3: Products and artifacts in the production repository

8.2.1 Scenarios of Repository Maintenance

Following are two scenarios describing real cases of the use of the release server focusing on the autocleanup of the repository.

Cleaning all obsolete products

Lisa is ready to release the newest version of her product. During the product's lifetime, it has been released many times and list of versions kept in the repository has grown large. She wishes to release her newest version as production version, alias PRO, and she would also like some of the older version to be removed. She considers her newest release quite stable and concludes that there is no point of keeping a long history of versions. Knowing that the default *number of versions to keep* is set to two, she overrides this property in her project.properties and sets it to zero. When releasing, all obsolete versions will be deleted, saving only the ones that are being depended on, or the ones affiliated with an alias.

Saving obsolete products

John is ready to release. He would like to release some new features of his product as a development version, thus with the alias NEXT. Before he releases,

he takes a look into the repository and sees that not all obsolete products should be removed. Two of the latest version numbers, not be depended on by any other project, and not containing an alias should not be deleted. Knowing that the default value of *number of versions to keep* is two, he does not have to reset this property in his project.properties. During the release, two of the products considered obsolete will be kept.

8.2.2 Requirements for Repository Maintenance

Cleaning versions of a product from the repository consists of two parts; deleting the physical files from the server, and deleting the occurrence of those files in the repository.xml. In order for CmmnBuild and Release Tool to work properly, it is crucial that the file system and the repository.xml stay consistent.

In case an obsolete product is deleted against the will of a developer, it is important to know exactly which versions were deleted. This will allow us to re-release the versions in question based on the source files kept in the CVS. Keeping track of which files are deleted and when is therefore important.

If for some reason, the automatic cleanup of the repository is no longer desired, it should be possible for an admin of the tools to switch off the removal of obsolete versions.

In addition to the definitions presented earlier, this led to the following functional requirements.

Functional Requirements

- **F-AC1** - Calculate obsolete versions (see definition in 8.2)
- **F-AC2** - Allow users to specify a property stating how many obsolete version not to delete. Provide a default value of 2 (see definition in 8.2).
- **F-AC3** - Delete, from the repository, the versions deemed obsolete, after ”saving” the number of version specified in F-AC2
- **F-AC4** - Update the file mirroring the repository (repository.xml) in a consistent matter
- **F-AC5** - Record which versions of which product were deleted and when
- **F-AC6** - Provide the admin of the Release Tool with a ”switch” to easily turn on and off the cleanup functionality

8.2.3 System design - Repository Maintenance

The repository maintenance feature is an extension to the Release Tool, but properties will be set by the user in CmmnBuild. Cleaning obsolete products is only supported for users releasing through CmmnBuild.

The logic for the AutoCleanUp is placed in the ant-script *updateandclean.build.xml*. In addition to use a set of standard Ant targets, this service relies on the *cern-jjar* package to fulfill its requirements. The cern-jjar package is written in Java and provides custom functionality to CmmnBuild and Release Tool. Its services are used from the ant-scripts. To accommodate the functionality needed for the AutoCleanUp, cern-jjar was extended with the *ReleaseTask*. This task handles the update of the repository.xml and the calculation of which versions of a product are obsolete. The list of obsolete versions is returned to the ant-script where the deleting is carried out. An overall illustration of the process is provided in figure 8.4.

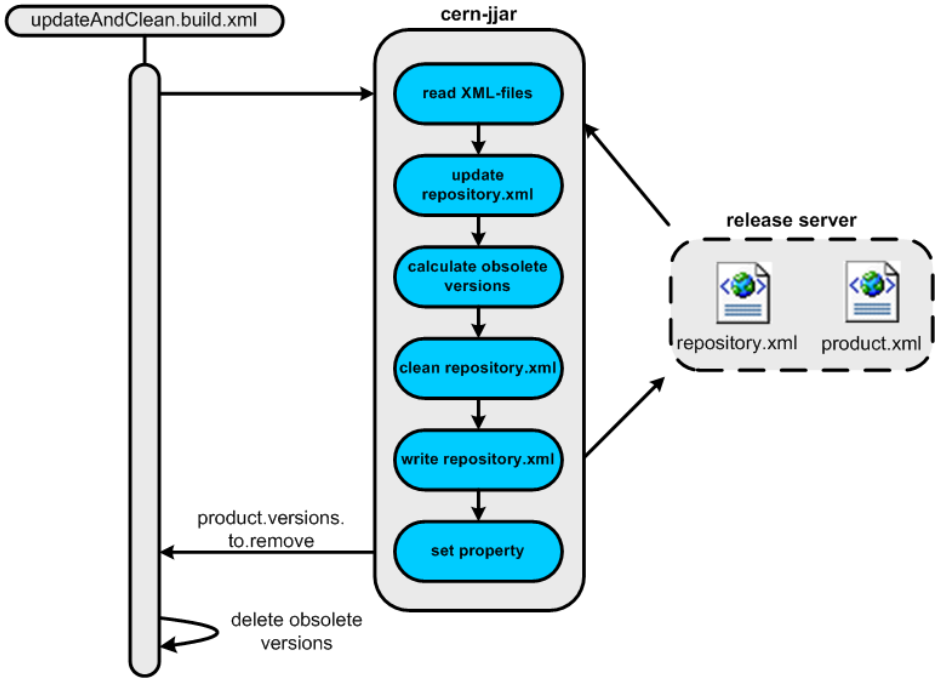

Figure 8.4: Overview of the repository maintenance process

8.2.4 Implementation - Repository Maintenance

This section provides a detailed view of the implementation of the autocleanup functionality. It is divided into sections based on the sequence of actions making up the whole process. Figure 8.5 shows a class diagram of the classes involved in the process. A detailed illustration of the sequence may be found in Appendix B.1.

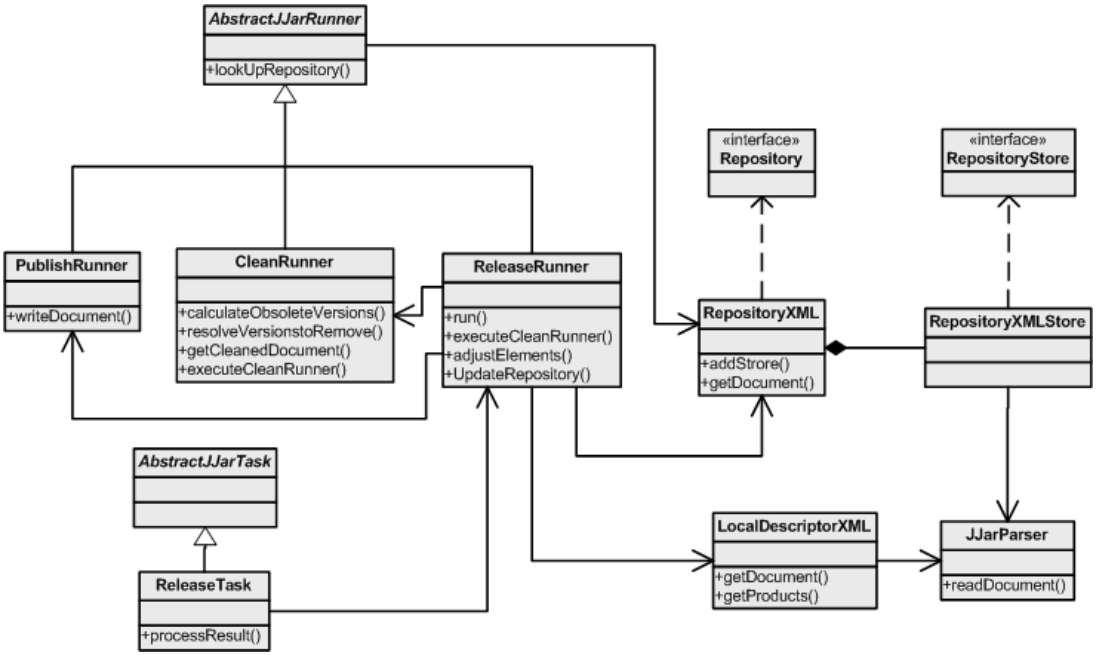

Figure 8.5: A selection of the classes in the *cern-jjar* package

Updating repository.xml

All information about the contents of the repository is gathered from the repository.xml. This file is parsed using the JJarParser which uses the Document Object Model, DOM [27], providing the possibility to change and update the information contained by the file.

The information about the new project being released is taken from the product.xml. After parsing, this information is modified using XPath [28] to fit into the repository.xml. The new product info is then added to the file, followed by an update of the aliases for the current product. The total process of updating the repository descriptor is done in two steps. First, the new product

is added and the aliases are updated, and second, the obsolete versions are removed from the document. The document is not written to file between these actions.

Calculating obsolete versions

The decision to only consider versions of the current product when calculating obsolete versions, limited the need for a complex algorithm. After the obsolete versions of the product have been calculated, the user-specified number of obsolete versions to save are removed from the list of versions to delete. Following is the section of the AutoCleanUp responsible for computing the obsolete versions of a product:

```
1  public Product[] computeObsoleteVersionsForProduct(Product product) {
2    final MultiVersionsProduct mvp =
3      getProductionStore().getProduct(product.getName());
4    // in case of an initial release of a new product
5    if (mvp == null) {
6      return new Product[0];
7    }
8
9    //removes all versions which have an alias. These are not obsolete
10   Product[] versionsFilteredOnAlias = filterVersionsOnAlias(mvp.getProducts());
11
12   Set allDependecies = getAllDependenciesInAllRepositories();
13
14   List obsoleteVersions = filterVersionsOnDependencies(versionsFilteredOnAlias,
15     allDependecies);
16
17   return (Product[]) obsoleteVersions.toArray(
18     new Product[obsoleteVersions.size()]);
19 }
20
21 private List filterVersionsOnDependencies(Product[] versionsToFilter,
22     Set allDependecies) {
23
24   List filteredList = new ArrayList();
25   for (int i = 0; i < versionsToFilter.length; i++) {
26     String productName = versionsToFilter[i].getName();
27     Version productVersion = versionsToFilter[i].getVersion();
28
29     //to find if the version of the product we are checking is obsolete, we
30         //create a new instance of Dependency based on the productinfo and check
31     //if it is in the list
32     if (!allDependecies.contains(new Dependency(productName, productVersion))) {
33       filteredList.add(versionsToFilter[i]);
34     }
35   }
36   return filteredList;
37 }
38
39 private Product[] filterVersionsOnAlias(Product[] products) {
40   List list = new ArrayList();
41
42   for (int i = 0; i < products.length; i++) {
43     if (!products[i].hasAlias()) {
44       list.add(products[i]);
```

```
45      }
46    }
47    return (Product[]) list.toArray(new Product[list.size()]);
48  }
49
50  private Set getAllDependenciesInAllRepositories() {
51    final Set allDeps = new HashSet();
52    for (int i = 0; i < repositoryStores.size(); i++) {
53      RepositoryStore store = (RepositoryStore) repositoryStores.get(i);
54      MultiVersionsProduct[] allMultiVersionedProducts = store.getProducts();
55
56      for (int j = 0; j < allMultiVersionedProducts.length; j++) {
57        Product[] multiVersionedProduct = allMultiVersionedProducts[j]
58                        .getProducts();
59
60        for (int k = 0; k < multiVersionedProduct.length; k++) {
61          Dependency[] deps = multiVersionedProduct[k]
62                              .getDependenciesAsArray();
63            allDeps.addAll(Arrays.asList(deps));
64        }
65      }
66    }
67    return allDeps;
68  }
```

Deleting the versions

The deletion of the versions is done in the ant-script. Ant provides a core-task for deleting files based on a relative path. The list of versions to delete is given to the ant-script as a comma-separated list of paths. It is provided by setting a property in *ReleaseTask* called *product.versions.to.remove*. Following is a section of the ant-script responsible for calling ReleaseTask and deleting the versions deemed obsolete.

```
1
2  <target name="removeObsoleteVersions"
3    depends="updateRepostioryXMLandCalculateObsoleteVersions"
4    if="use.product.xml">
5
6    <var name="log.output"
7      value="RELEASE: Calculating obsolete versions of the product:
8      ${release.product.name}" />
9    <echo message="${log.output}" />
10   <antcall target="writeToLog" />
11
12   <var name="log.output" value=" number of obsolete versions to keep:
13     ${numberofversionstokeep}" />
14   <echo message="${log.output}" />
15   <antcall target="writeToLog" />
16
17   <if><equals arg1="${product.versions.to.remove}" arg2="empty" />
18     <then>
19       <var name="log.output" value=" No versions to remove for this product" />
20       <echo message="${log.output}" />
21       <antcall target="writeToLog" />
22     </then>
23     <else>
```

```
24        <var name="log.output" value=" obsolete versions to remove:
25          ${product.versions.to.remove}" />
26        <echo message="${log.output}" />
27        <antcall target="writeToLog" />
28
29        <var name="log.output" value="RELEASE: Deleting obsolete versions ..." />
30        <echo message="${log.output}" />
31        <antcall target="writeToLog" />
32
33        <for list="${product.versions.to.remove}" delimiter=","
34          param="version.to.delete" >
35          <sequential>
36          <delete dir="${release.areas.distribution}/@{version.to.delete}" />
37
38            <var name="log.output"
39              value=" deleting ${release.areas.distribution}/
40              @{version.to.delete}" />
41              <echo message="${log.output}" />
42              <antcall target="writeToLog" />
43          </sequential>
44        </for>
45
46        <var name="log.output" value=" Deleting completed" />
47        <echo message="${log.output}" />
48        <antcall target="writeToLog" />
49      </else>
50    </if>
51  </target>
52
53
54  <target name="updateRepostioryXMLandCalculateObsoleteVersions"
55    depends="updateRepostioryXMLandCalculateObsoleteVersions-init" >
56    <var name="log.output" value="RELEASE: Updating repository.xml" />
57    <echo message="${log.output}" />
58    <antcall target="writeToLog" />
59
60    <trycatch property="exception" >
61      <try>
62        <jjarReleaseProduct
63          numberofversionstokeep="${numberofversionstokeep}"
64          keepallversions="${keepallversions}"
65          resolvealias="${resolvealias}"
66          localdescriptor="${release.product.localdescriptor}"
67          localrepository="${release.product.localrepository}"
68          defaultdependencyversion="${which.version}"
69          filedirectory="${release.product.version}/${release.product.dist.dir}">
70
71          <repository url="${release.files.repository.production}" />
72          <repository url="${release.files.repository.thirdparty}" />
73        </jjarReleaseProduct>
74      </try>
75      <catch>
76        <property name="release.error" value="${exception}" />
77        <antcall target="cleanup"/>
78      </catch>
79    </trycatch>
80    <var name="log.output" value=" repository.xml updated successfully" />
81    <echo message="${log.output}" />
82    <antcall target="writeToLog" />
83  </target>
```

Consistency

For the release-server to function properly it is crucial that the repository.xml mirrors the content of the repository precisely, and that this stays consistent. If for any reason, an exception should occur in the ReleaseTask, it must be possible to perform a roll-back leaving the situation as it was before the release was initiated. Ant provides a *try/catch* which will catch any exception thrown from the ReleaseTask. If an exception is thrown, a proper message is displayed, and the release of the product will be rolled back, leaving both the repository and the repository.xml in the state they were before the release was initiated.

Logging

A logging of all releases was already in place. All that needed to be done was to add a list of which versions of the product being released was deleted.

Admin switch

A property, `switch.off.repository.cleaning`, was added to the property file of the Release Tool. Default value is `false`, but if set to true, the cleanup feature will be deactivated for all releases. This property is, as the name implies, only available for administrators of the tool.

8.3 Notification of release

As mentioned in the previous section, software developed for the control center is under constant evolution. This includes several low-level applications relied upon by many other projects. Each time a product is re-released with changes in form of extensions or improvements, this might affect anybody depending on this project. The section uses a JIRA which contains information about each project under development. The JIRA tool offers users the possibility to register issues in form of bug reports or requests, while it provides the developers the opportunity to update information of the current state of their product. This however, requires that the developer actually updates the information and that the person interested checks to see if there are any updates. No notification scheme is now in place, and developers solve this by sending emails to mailing lists informing about a new release. This, of course, means that the information reaches many developers with no interest of this particular piece of software. To solve this problem, and to provide an easier way to send a notification of a release, an automated release-notification functionality included in the release process was thought of.

The overall goal is to merge the functionality of JIRA and Release Tool. The notification scheme described here will constitute a first step.

8.3.1 Scenarios of Release Notification

Presented here are two scenarios of the use of the notification scheme.

Subscribe to a product of interest

John is developing a new product, ”japc-ext-remote”. He depends on the functionality of four other products, product A, B, C and D, developed in his section. Products A and C are stable and the rate of change is low. The last two however, B and D, are still in a state where new functionalities are added frequently. To be able to exploit these improvements to the full extent, John is interested in knowing when a change has occurred, and what has changed. Preferably as soon as possible after a release. He goes to the web page for subscribing to change-notifications and registers product B and D with his email address. Next time any of the two products are released, he will receive an email explaining the changes done.

Unsubscribe to a product no longer interesting

Lisa is currently subscribing to three different products, A, B and C. Her product which depended on product B and C has now been handed over to a new developer, and the updates for these products are no longer interesting for Lisa. Product A, on the other hand, is still interesting, as she depends on this in another project she is still active in. To avoid ”spam” every time product B and C are released, she goes to the web page, clicks on the button to view her subscriptions, and then unsubscribes to B and C. She continues to subscribe to product A.

8.3.2 Requirements for Release Notification

Seeing that most developers depend on several other projects from the section, being able to subscribe to multiple products is an obvious necessity. If a product, for some reason is no longer of interest to a developer, unsubscribing to products must be an opportunity.

Considering the magnitude of projects available to subscribe to, and the number of potential subscribers, the *subscriptions.xml* has the potential of becoming quite big. Therefore it should only contain products which, at any given moment, has subscribers.

Because of the limited functionality of the feature, no security in form of username and password is necessary and it should support all email addresses.

For an admin, it might be interesting to see a list of all products under subscription and by whom.

It was early decided that focus would be on getting this functionality up and running, and that advanced features could be added at a later point. The first priority was therefore to create a working prototype. This led to the following functional requirements.

Functional requirements

- **F-AC1** Register email to subscribe to a release notification of a product
- **F-AC2** Possibility to subscribe to several products
- **F-AC3** Possibility to unsubscribe
- **F-AC4** Possibility to view all subscriptions per user

- **F-AC5** Products with no subscribers should not appear in the subscription descriptor file
- **F-AC6** Possibility to view all products subscribed to, with all subscribers registered for each.

8.3.3 System design - Release Notification

The release notification scheme consists of two parts. One part is responsible for registering subscriptions from a user, while the other part sends a release notification mail during a release. The sending of mails is included in the Release Tool.

The subscription feature is offered to the users through a web page running on the release-server. This page lists all available products in the production repository. The information about the content of the repository is gathered from repository.xml. Because of the simplicity of this feature, it was decided against using any form of web service framework for the implementation.

Information about subscriptions, consisting of products currently being subscribed to, with a list of subscribing emails attached, is stored in *subscriptions.xml* (figure 8.6).

```
1  <subscriptions>
2    <product name="lhc-lbds">
3      <subscribers>
4        <sub email="enes@cern.ch"/>
5            <sub email="olaho@cern.ch"/>
6            <sub email="golebiov@cern.ch"/>
7      </subscribers>
8    </product>
9    <product name="sps-multiq">
10     <subscribers>
11       <sub email="enes@cern.ch"/>
12           <sub email="vidarfro@cern.ch"/>
13     </subscribers>
14   </product>
15   <product name="sps-multitq">
16     <subscribers>
17       <sub email="enes@cern.ch"/>
18           <sub email="golebiov@cern.ch"/>
19     </subscribers>
20   </product>
21 </subscriptions>
```

Figure 8.6: An example of the subscriptions.xml file

While the web page provides the user interface for the notification service, all the logic is implemented in Java, in the *cern.release.notification* package.

Figure 8.7 shows an overview of the system.

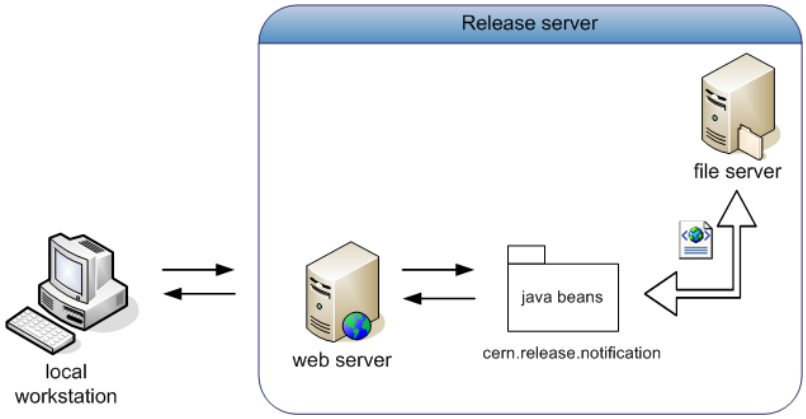

Figure 8.7: Overview of the release notification scheme

8.3.4 Implementation - Release Notification

The web interface

The web interface is written in JSP with the use of HTML forms to collect data from the user. The page is left simple, with only the necessary components available. All the logic is performed by Javabeans included in the *cern.release.notification* package explained in the following section.

When entering an email, the user may choose to view his subscriptions, or he may choose one or several products to register for. When viewing ones subscriptions, one might also choose to unsubscribe to selected products. Figure 8.8 shows the web interface after *enes@cern.ch* has entered his email and chosen to view his subscriptions.

For the use of the administrators of the Release Tool, it is possible to view all subscriptions registered. When entering "*" as email, a list of all products being subscribed to, with their subscribers, will be displayed.

The cern.release.notification package

The cern.release.notification package contains a subpackage Javabeans which handles all interaction with the JSP pages. Figure 8.9 provides an overview of the content of the package.

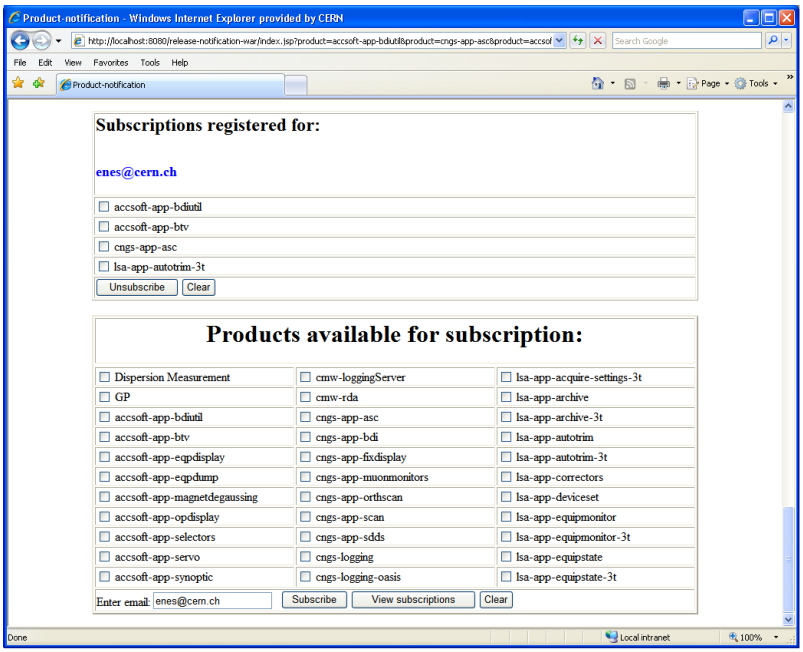

Figure 8.8: The web interface, while viewing subscriptions for *enes@cern.ch*. The number of products available for subscription is reduced for illustration purposes.

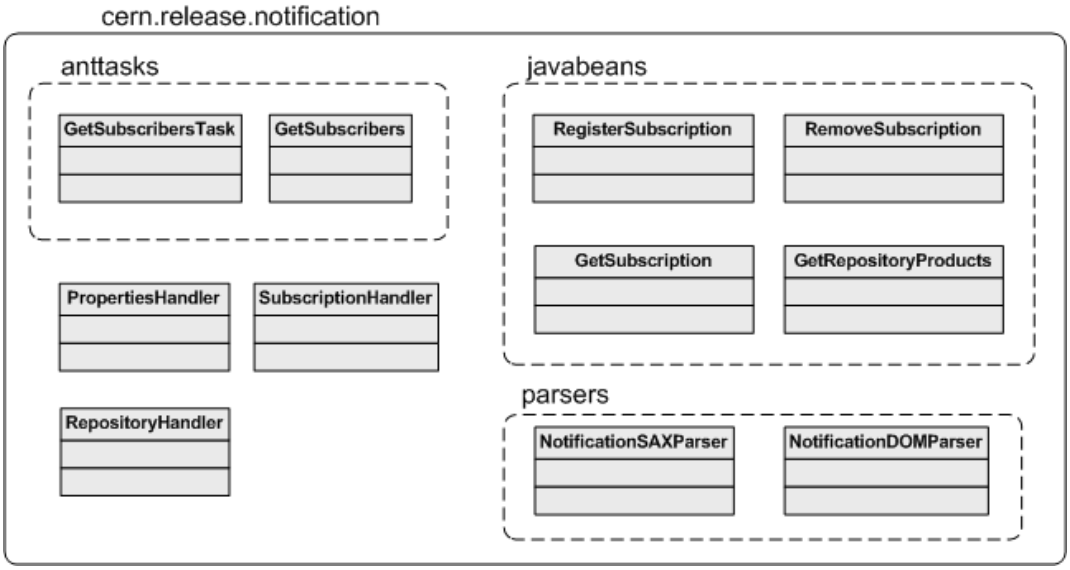

Figure 8.9: The cern.release.notification package

The subpackage *anttasks* contains the logic for returning a list of products based on a product name. It is used from the Release Tool in the ant-script *notification.build.xml*. By returning to the ant-script a comma-separated list of recipients, the `<mail>` target in the script is able to notify the subscribers. Figure 8.10 shows the flow of events of the release notification when a product is released. Notifying subscribers is the last thing done in the release process.

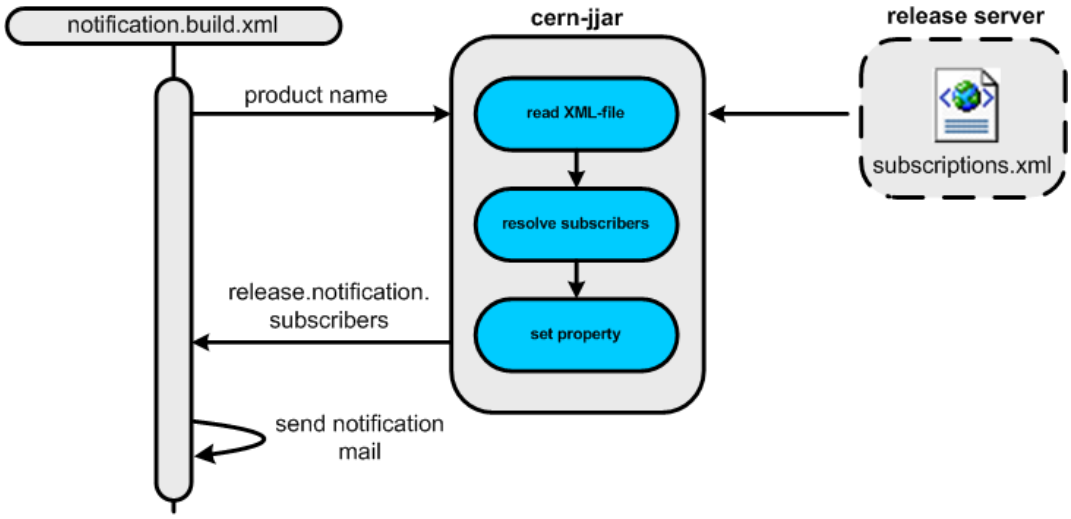

Figure 8.10: Overview of the mail sending process

Figure 8.11 shows the notification.build.xml, including the `<mail>` target, while figure 8.12 shows the mail-template used to notify the users. *$release.notification.subscribers* is set in *notificationGetSubscribers* implemented in Java. The current state of the mail-template does not include any user comments, but this will most likely be added later.

```
<target name="sendNotificationMail" depends="getSubscribers, createSubject" >
  <mail mailhost="${release.notification.mailhost}" subject="${subject}"
        messagefile="${release.files.notification.mailtemplate}"
        tolist="${release.notification.subscribers}"
        >
          <from address="Release"/>
  </mail>
</target>

<target name="createSubject">
  <property name="subject" value="New version of ${release.product.name}
    just released." />
</target>

<target name="getSubscribers" >
  <notificationGetSubscribers
        productname="${release.product.name}" >
  </notificationGetSubscribers>
</target>
```

Figure 8.11: The ant-script sending mail

```
NEW RELEASE:
==========================================================================
Product:        ${release.product.name}
Version:        ${release.product.version}
Alias:          ${which.version}
==========================================================================

A new version of "${release.product.name}" has just been
release with version number "${release.product.version}"
The new version was released as "${which.version}"

For more information about this product, please visit
${release.url.production.repository}/${release.files.repository.descriptor}
#${release.product.name}-${release.product.version}

Kind regards,

Release

To unsubscribe to this product, go to ${release.url.notification}
```

Figure 8.12: The mail-template used to notify users of a release

Chapter 9
Evaluation and discussion

The approach assumed for the practical parts of this thesis constituted of an agile development process based on face-to-face communication with many small iterations defined along the way. Even though the overall goals remained the same throughout the project, this allowed us to adjust quickly to changes in requirements, and made facing difficulties easier. In addition, such an approach has allowed me benefit from the expertise and experience of both my supervisor and other developers representing the customers.

Following is an evaluation of each of the new features implemented during the work of this thesis:

Branching

Overall goal:

> *Incorporation of the CVS branching feature into CmmnBuild, facilitating the process of providing a bugfix*

The branching extension of the CmmnBuild tool was the first implementation of the project, and acted as an introduction to both CmmnBuild and Ant. The technical documentation available for CmmnBuild and Release at the startup of my thesis was very limited. Therefore, starting with a small extension provided me with a good opportunity of *learning by doing,* which was valuable throughout the rest of the project.

Branching is not a highly used feature, but it was desired due to the strategy of never to patch released software. The desired functionality was implemented, and branching through CmmnBuild has been available for users since September. For the specific requirements implemented, I refer to chapter 8.1.

Repository maintenance

Overall goal:

> *Provide functionality to limit the amount of manual labor necessary*

to keep the repository-server in an easily maintainable state

The repository maintenance constituted the biggest part of this thesis, and was the most important extension implemented. There were no requirements as to measure the success or failure of the implementation other than support for the requirements presented in 8.2. All the functional requirements were implemented and a new version of the Release Tool and CmmnBuild, including the AutoCleanUp extension was released in late November 2006.

Table 9.1 is an extension to the table presented in 8.2. February has been added to the list, and the data shows a clear moderation of the growth. Even though the number of products has *increased* by 65, which is the highest delta in the column, the number of artifacts in total has *decreased* by 13. This clearly shows that the repository did contain a lot of software no longer used. Even though the descriptor file has grown, due to the many new products, it has grown at a much lower rate than it would have without the repository cleaning.

Repository data			
Month	**Data1**	**Data2**	**Data3**
March	204	478	7591
May	207	913	12756
August	242	1221	15534
November	260	2046	27051
February	325	2033	30897

Table 9.1: **Data1** refers to the *number of products* in the repository, **Data2** is the total *number of artifacts* and **Data3** presents the *size of the repository descriptor* (repository.xml) in lines (of xml-code)

There are no data indicating the exact number of artifacts released in the period between November and February, but we can assume that data from previous months gives us an indication. In that case, the cleaning of the repository have been successful in relieving the manual labor normally spent trimming the repository.

Release notification

Overall goal:

Provide support for automatic release notification. A first step in a more extensive solution

Release notification scheme was the last extension implemented, and the only one considered a prototype. Due to lack of time, a full notification solution

including support for release-notes, issue tracking, etc. was not implemented. A working service, covering the functionality requirements defined, has been implemented as explained in chapter 8.3, but the service is not yet made available for users. All functionalities were implemented as planned, but an evaluation of the feature based on its use is so far not available.

In chapter 6.2, I posed the question of whether or not it could be beneficiary to switch to external tools to handle build and release management. During the last few years, many tools have appeared in the build and release category, and the topic is getting more attention now than earlier. Estublier et. al. [13] argues that high-end SCM tools, providing good support for all concepts presented by Dart [9], are now emerging. Component-based, distributed development increases the need for solid tools to support the software process.

Accelerator control software certainly addresses a very specific field of operation, but the question is if the development process in use differ so much from that of other domains. However, CmmnBuild, together with the Relase Tool have been in use for several years and are well implemented by the developers. Developing an in-house tool cost time and effort, but so does the migration to and customization of an external solution. One option is to combine the customization of a tool with the participation in an open-source project or similar, and thereby collaborate with other institutions in developing desired features. As mentioned, the CmmnBuild tool is already in use in Fermilab.

Any solution involving existing tools will have to be started with a thorough analysis of what's available. A natural place to start is Apache Maven considering it was taken into account in 2002. Since then, Apache Maven has evolved to become a serious actor providing SCM solutions focusing on the support of the build life-cycle. Support for transitive dependencies has even been available since Apache released the 2.0 version [19].

A conclusion of the thesis, including this question is presented in chapter 10.

Part IV

Conclusion and Further Work

Chapters

Chapter 10
Conclusion

The theory of SCM emphasizes the importance of a well implemented infrastructure supporting the evaluation of complex systems. Automated processes for handling changes, corrections, extensions and adaptations are agreed upon to be important factors of software projects [12]. Proper tools is important for the overall management of software development, but also for the efficiency of the single developer [4].

This thesis has presented an extension of two tools supporting the build and release process in the AB/CO/AP section at CERN. The tools, CmmnBuild and Release Tool, provide build and release support for the development of *accelerator control software*. Accelerator control software is used by the CERN Control Center to run and control the major accelerators at CERN.

The purpose of the this thesis was to implement and deploy new functionality to CmmnBuild and Release. Three new features were defined with overall goals and specific functional requirements.

Based on the evaluation and discussion presented in chapter 9, this project has succeeded in providing valuable extensions to the tools in question. All three features were implemented, and the *branching feature* and *automated repository cleanup* have been included in the production version of their respective tools. The last feature, the *notification scheme* is at the current moment not yet made available for use, but is ready to be deployed when time allows it.

In 2002, the AB/CO/AP section posed the question of whether to implement an in-house solution, or select an existing external tool. An in-house solution was chosen at that time, but as new SCM tools are emerging, offering a wider spectrum of functionality the earlier, this question has reappeared.

A discussion of the question was presented in chapter 9, indicating different possible solutions. A final answer involves a discussion of the future of CmmnBuild and Release Tool. This project has given an indication of the cost, in form of time and resources, required to extend the functionalities of the tools. Whether or not the acquisition of an external tool results in less work can only be the result of a thorough analysis of available tools in addition to a clear vision of what future requirements might include. In any case, I think the

allocation of enough time and resources will be a key issue in providing the developers of accelerator control software with quality tools to support their development process.

Chapter 11
Further Work

11.1 Further work for CmmnBuild and Release

The immediate work of CmmnBuild and Release Tool includes the deployment the release notification scheme implemented. The next issue to address is the incorporation of JIRA into CmmnBuild and Release. This will not only provide support for a developer along the path of developing a new version of a product, it will also provide support for proper change management, change requests, issues tracking etc. It will bridge the gap between the release of one version to the start of the next, which is so far not supported by CmmnBuild and Release. It will provide complete support for the product lifecycle.

In addition, it is the discussion of whether or not to change to an external tool. I tried to shed some light on this question in Chapters 9 and 10, but a final decision lies in the hands of AB/CO/AP.

11.2 Further work for SCM in general

To survive, SCM must evolve to address new concepts and trends as they appear within the field of software development. To a higher extent than earlier, projects involve distributed locations and independently developed components. Presented here are two areas which might be of interest for SCM in the future, however it is important to note that this is only a fraction of possible areas in which SCM might extend.

The research of Computer Supported Cooperative Work, CSCW, addresses the field of how collaboritive activities might be supported through computer systems [5]. One area of interest wihtin SCM is support for cooperative work across distributed locations by incorporating CSCW elements into SCM tools [8].

Process support is another area in which SCM may advance. Estublier states that specific formalisms are needed in which different aspects of SCM processes

may be defined, tailored and enhanced [12]. Conradi et. al. argues that advanced SCM tools offer faciclities for process support, but raises the question if this might be made independent of basic product management [8].

The basic concepts and technologies of SCM may be agreed upon, but the work of expanding them will continue as software development evolves. New areas of interest will apear, and relations to other domains will be explored. It's a continuous process in order to provide quality support to the complex field of software development.

Appendices

Appendix A
CmmnBuild targets

Targets	Action
Compilation / packaging	
compile	compile all source files
dist	creates the full distribution (including all documentation and junit)
clean	removes all generated files
Deployment and release	
tagcvs	tag the cvs repository with the current version
branch	create a branch at a specified versionnumber, and check out this branch
release	tagcvs and release in production. Default alias is PRO. Can be changed using -Dalias=NEXT
devrelease	release the head of cvs and place the product in Qfix replacing whatever was before.
nextmajor	change the major of the version in the descriptor to the next one (1.2.3 => 2.0.0)
nextminor	change the minor of the version in the descriptor to the next one (1.2.3 => 1.3.0)
nexttiny	change the tiny of the version in the descriptor to the next one (1.2.3 => 1.2.4)
nextmodifier	change the modifier of the version in the descriptor to the next one (1.2.3beta1 => 1.2.3beta2)
setversion	set the version in the descriptor file to one given in the version variable ex: ant setversion -Dversion=1.0.0
dist-release	dist target called by the release for building the product in production

Jar management	
cleanjars	removes all jars from the local lib directory
getjars	get dependencies in the local lib directory. Default alias is PRO. It is possible to specify the alias to use via -Dalias=DEV
getjarsdev	get dependencies in the local lib directory using the DEV alias
getjarsnext	get dependencies in the local lib directory using the NEXT alias
publishjars	publishes the jar and the descriptor of this product to the dev jars repository
JUnit Test	
junit	execute all unit tests written for the project
Documentation / Code Quality	
javadoc	creates the javadoc
java2html	creates html files from the java source files
jdepend	analysis of the dependencies in the source code
pmd	Runs a set of static code analysis rules on source code
checkstyle	checks the style of the source code
NetBeans modules management	
makenbm	creates and packages the nbm file produced by this project
publishnbm	publishes the nbm file produced by this project to the remote update center

Table A.1: The complete list of targets in CmmnBuild

Appendix B
Sequence diagram

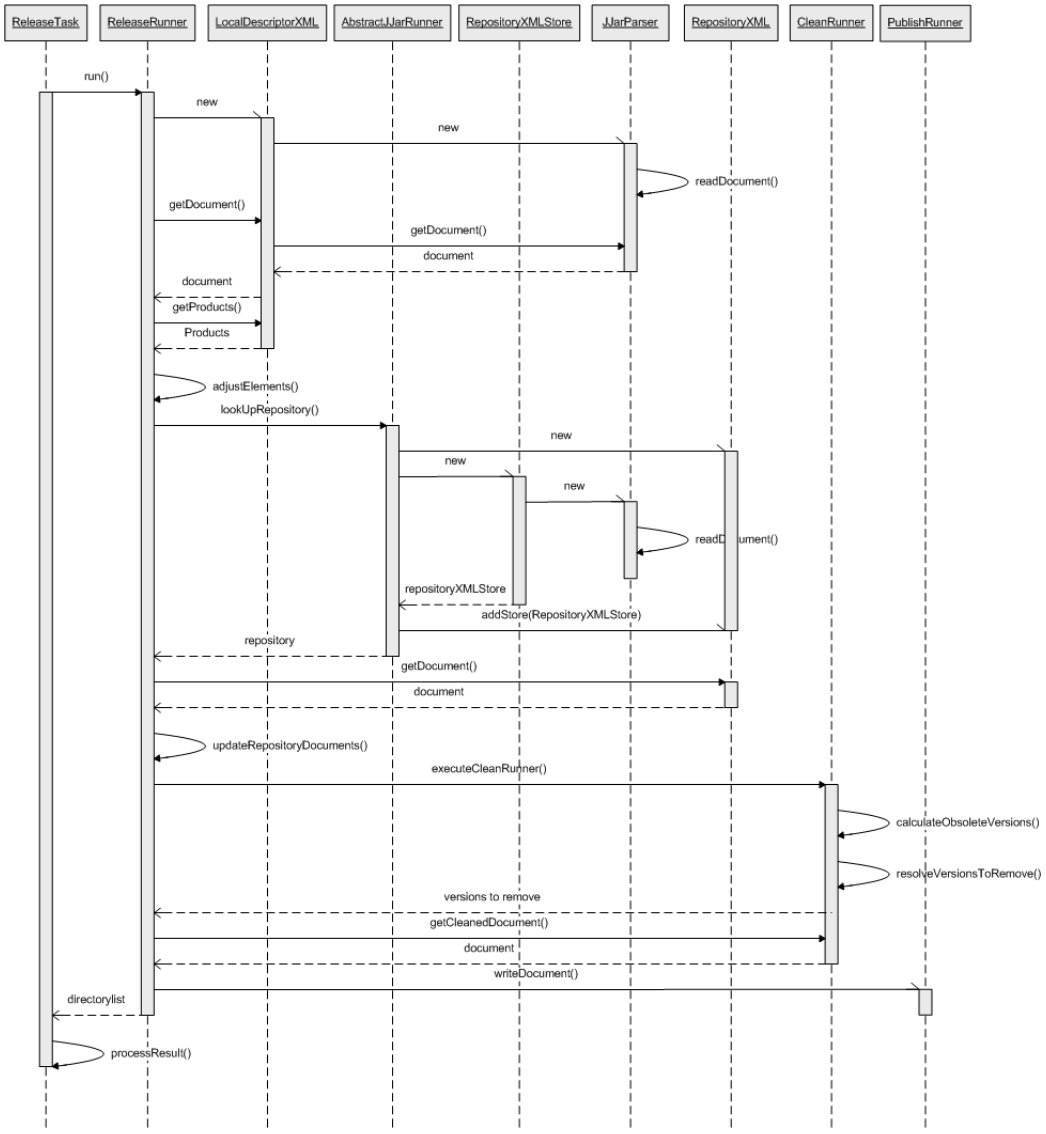

Figure B.1: Repository maintenance sequence diagram

Bibliography

[1] Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen. New directions on agile methods: a comparative analysis. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 244–254, Washington, DC, USA, 2003. IEEE Computer Society.

[2] Apache Ant. Apache ant 1.7.0 manual. *http://ant.apache.org/manual/index.html*. [Accessed August 2006].

[3] Atlassian. Jira. *http://www.atlassian.com/software/jira/*. [Accessed December 2006].

[4] Steve Berczuk. Pragmatic software configuration management. *IEEE Software*, 20(2):15–17, 2003.

[5] P. Carstensen and K. Schmidt. Computer supported cooperative work: New challenges to systems design, 1999.

[6] Per Cederquist. Version management with cvs. *http://ftp.gnu.org/non-gnu/cvs/source/stable/1.11.22/cederqvist-1.11.22.pdf*. [Accessed December 2006].

[7] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, 1998.

[8] Reidar Conradi and Bernhard Westfechtel. Scm: Status and future challenges. In *System Configuration Management*, pages 228–231, 1999.

[9] Susan Dart. Concepts in configuration management systems. In *Proceedings of the 3rd international workshop on Software configuration management*, pages 1–18, New York, NY, USA, 1991. ACM Press.

[10] Mark Dowson. The software process and software environments (panel session). In Jack C. Wileden, editor, *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 302–304, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[11] Institute O. Electrical and Electronics E. (ieee). *IEEE 90: IEEE Standard Glossary of Software Engineering Terminology*. 1990.

[12] Jacky Estublier. Software configuration management: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 279–289, New York, NY, USA, 2000. ACM Press.

[13] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.*, 14(4):383–430, 2005.

[14] Apache Software Foundation. *http://www.apache.org/*. [Accessed September 2006].

[15] Rebecca E. Grinter. Using a configuration management tool to coordinate software development. In *COCS '95: Proceedings of conference on Organizational computing systems*, pages 168–177, New York, NY, USA, 1995. ACM Press.

[16] IEEE. Ieee guide to software configuration management. ANSI/IEEE Std 1042- 1987. IEEE, New York.

[17] Gregor Joeris. Change management needs integrated process and configuration management. In *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 125–141, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[18] Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, June 2003.

[19] Apache Maven. *http://maven.apache.org/*. [Accessed September 2006].

[20] oops consultancy. Xmltask. *http://www.oopsconsultancy.com/software/xmltask/*. [Accessed September 2006].

[21] Ant-Contrib Project. Ant-contrib. *http://ant-contrib.sourceforge.net/*. [Accessed October 2006].

[22] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[23] Thomas Ósterlie (unpublished). Problems and solutions: An ethnography of large-scale software maintenance work. Draft of PhD Thesis.

[24] André van der Hoek and Alexander L. Wolf. Software release management for component-based software. *Softw. Pract. Exper.*, 33(1):77–98, 2003.

[25] Marlon Vieira and Debra Richardson. Analyzing dependencies in large component-based systems. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 241, Washington, DC, USA, 2002. IEEE Computer Society.

[26] Marlon Vieira and Debra Richardson. The role of dependencies in component-based systems evolution. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 62–65, New York, NY, USA, 2002. ACM Press.

[27] W3C. Document object model (dom). *http://www.w3.org/DOM/*. [Accessed October 2006].

[28] W3C. Xml path language (xpath). *http://www.w3.org/TR/xpath*. [Accessed October 2006].

[29] Wikipedia. *http://en.wikipedia.org/wiki/Apache_Maven*. [Accessed September 2006].

[30] Wikipedia. Agile software development. *http://en.wikipedia.org/wiki/Agile_software_development*. [Accessed October 2006].

[31] Xiaomin Wu, Adam Murray, Margaret-Anne Storey, and Rob Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 90–99, Washington, DC, USA, 2004. IEEE Computer Society.