NTNU
Innovation and Creativity

Improved Backward Compatibility and API Stability with Advanced Continuous Integration

Erik Drolshammer

Master of Science in Computer Science

Submission date:	June 2007
Supervisor:	Tor Stålhane, IDI
Co-supervisor:	Trygve Laugstøl, Objectware AS

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

Problem Description

Backward compatibility can be tested by integrating projects that depend on the previous version of the service with the new (development) version of the service. If it builds successfully, then the new version is backward compatible for the functionality utilized by these projects. Otherwise, a compatibility issue has been identified. Additionally, if integration is done after every change to the service, it will also be possible to determine what change that caused the build to fail. The concept is thus to utilize projects that depend on the service as test data and build these projects at regular intervals with the new version of the service.

The objective of this thesis is to determine if continuous integration can be used to test backward compatibility for services. Since tool support is necessary to take advantage of the concept, a prototype will be created as proof of concept. The purpose of the prototype is to determine if an implementation based on an existing continuous integration server is feasible with the technology available today. Support for Maven 2 projects is considered important, so it is possible to take advantage of the metadata in Maven's Project Object Model (POM). Continuum is therefore chosen as the underlying build engine, due to its excellent integration with Maven.

Assignment given: 16. January 2007
Supervisor: Tor Stålhane, IDI

Abstract

Services with a stable API and good backward compatibility is important for component-based software development and service-oriented architectures. Despite its importance, little tool support is currently available to ensure that services are backward compatible. To address this problem a new continuous integration technique has been developed.

The idea is to build projects that depend on a service with a new version of the service. This ensures that the development version is compatible with projects that depend on the regular version. A continuous integration server is used to initiate builds. This entails that if a build breaks, the developers get feedback right away, and it is easy to determine which change that caused the broken build.

We show that an implementation is feasible by implementing a prototype as a proof of concept. The prototype use Continuum as the underlying build engine and utilize metadata from the Maven Project Object Model (POM). The prototype has support for multiple services. Services can thus be checked for compatibility with each other, in addition to backward compatibility with the regular version.

Keywords: Continuous integration, Continuum, Maven, Component-based software development (CBSD), Service-Oriented Architecture (SOA), Test-Driven Development (TDD), agile software development

Preface

This Master's thesis concludes my master's program at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The outline for the assignment was proposed by Trygve Laugstøl at Objectware AS and is related to Open Source Software (OSS) and the Apache Maven Project *Continuum*.

I would like to thank Trygve Laugstøl and supervising professor Tor Stålhane for valuable input and feedback. Their help have been invaluable. Additionally, I would like to thank the OSS community for quick and to the point answers regarding Continuum and the technology utilized by Continuum. Finally, I give my thanks to my fellow students at "Ugle" computer lab for great coffee, numerous on- and off-topic discussions, and great suggestions during the work on this thesis.

June 5, 2007

Erik Drolshammer

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This chapter presents the background for the project. The purpose of this chapter is to focus the rest of the report and explain what the problem is, why we want to solve it and how we intend to approach the problem. The motivation section introduce the domain and outline the context for the problem description. The next section contains the problem text, followed by a description of intended audience. The following sections explain the chosen research approach and the project process. The chapter ends with a report outline.

Motivation

As Information Technology (IT) is becoming ubiquitous, software engineering is becoming increasingly more important. Traditionally, preferred development methodologies have been plan-based. Today, the need for more rapid changes has made Test-Driven Development (TDD) and agile[1] development methodologies more popular. Continuous integration is one of the new development practices that is used to support the new work patterns.

Java software projects often utilize services. How can the developers of these services know that projects that depend on their artifact will still work with a new version? They can run all kinds of tests and employ software to check compliance with the Application Programming Interface (API). In addition, they might want to exploit the fact that there are projects that utilize their API. Building projects with the new version of the service will reveal whether they are compatible or not. Ideally, this integration should be done every time the service is changed, which indicates that automation and tool support is necessary.

Problem description

Backward compatibility can be tested by integrating projects that depend on the previous version of the service with the new (development) version of the service. If it builds successfully, then the new version is backward compatible for the functionality utilized by these projects. Otherwise, a compatibility issue has been identified. Additionally, if integration is done after every change to the service, it will also be possible to determine what change that caused the build to fail. The concept is thus to utilize projects that depend on the service as test data and build these projects at regular intervals with the new version of the service.

The objective of this thesis is to determine if continuous integration can be used to test backward compatibility for services. Since tool support is necessary to take advantage of the concept, a prototype will be created as proof of concept. The purpose of the prototype is to determine if an implementation based on an existing continuous integration server is feasible with the technology available today. Support for Maven [2] projects is considered important, so it is possible to take advantage of the metadata in Maven's Project Object Model (POM). Continuum [3] is therefore chosen as the underlying build engine, due to its excellent integration with Maven.

[1] See [1, 2, 3] and the Manifesto for Agile Software Development, `http://agilemanifesto.org`, for more information on agile development.

[2] `http://maven.apache.org/` Last visited: 2007.05.30

[3] `http://maven.apache.org/continuum` Last visited: 2007.05.30

Audience

This report should be interesting for anyone developing services that need to follow a strict versioning scheme, but the primary audience is Java enterprise developers. The concept is independent of the Java programming language, but our prototype is based on Continuum and was made possible by metadata found in Maven's Project Object Model (POM). The reader is assumed to have experience with Maven and general Java development. Familiarity with agile software development, Service-Oriented Architecture (SOA) and Component-based software development (CBSD) is also advantageous.

Research approach

This thesis consists of two main tasks; describe a conceptual solution and write a prototype to determine if an implementation is feasible and how the concept works in practice. In the following, we describe the methods we will apply to solve these tasks.

Concept

The conceptual solution will contain a textual description of the problem and identify inputs and outputs. This data will be presented as dependency graphs to illustrate how and when derived projects should be generated. These models will also be used to validate the prototype. I.e., success is determined by the degree of correspondence between the expected output presented in the conceptual model and the actual output from the prototype. Graphical dependency graphs also facilitate communication on the subject, making it easier to explain and discuss the concept.

Prototype

The purpose of the prototype is to determine whether an implementation based on Continuum is feasible. The goal is to create a proof of concept, and the focus will be on the business logic. Production level code quality is not important, and functionality not essential to illustrate the concept will be omitted. E.g., usability and the Graphical User Interface (GUI) will have low priority.

By employing *evolutionary prototyping* the proof of concept code may be iteratively refined into a fully functional prototype. Evolutionary prototyping is considered better suited than throw-away prototyping, since the prototype is based on existing software. With this approach it is at least *possible* that the prototype can be of some use to the open source community.

Project process

We will begin with the introduction. This ensures that the boundaries for the thesis are in place before delving into the prestudy. Problem definition and research approach are considered especially important. Afterwards, we will study state of the art to avoid reinventing the wheel. Central research topics are existing continuous integration techniques, existing continuous integration servers, versioning schemes, Maven Project Object Model (POM) and the architecture of Continuum.

Before we start prototyping, a draft of the conceptual solution and a development environment for Continuum will be in place. The former will serve as a rough design. The latter is important to make it possible to take small iterative steps and reduce the risks of integration. Prototyping may inspire new thoughts and ideas, so we will update the conceptual solution to reflect any new knowledge after the prototype is finished.

Finally, we will evaluate the prototype and discuss the chosen solution. Future work will also be identified to make it easier for others to continue the work.

Report outline

Below, brief descriptions of the different logical parts of the rest of the report are provided for easier navigation. Sequential reading is advised, but, e.g., skipping the prestudy is possible, if the reader is familiar with the domain.

Part II – Prestudy
The prestudy will explain the concept of continuous integration and its benefits. We will also examine a few of the most popular continuous integration servers and the Maven versioning scheme. It ends with an overview of Continuum's architecture and a description on how to set up a development environment.

Part III – Contribution
This part consists of chapter 3 and chapter 4. Chapter 3 is our proposed solution to the problem definition previously described, while chapter 4 describes a prototype that demonstrates the concept.

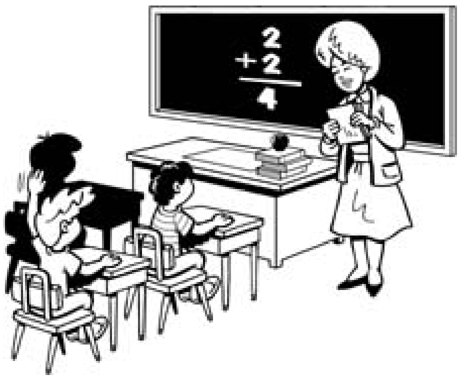
Part IV – Evaluation and discussion
This part consists of chapter 5 and chapter 6. Chapter 5 contains an evaluation of the prototype and the research methods. The focus here is on the project process and the choices we made. Chapter 6 sets the proposed solution into a wider context and evaluates its usefulness.

Part V - Conclusion and further work
This chapter is a summary of what we have done. The focus is on our contributions and why they are interesting. We will also list tasks and topics relevant for future work.

The CD
The CD that is delivered with this report contains the source code for the prototype and this report as pdf.

Chapter 2

Preliminary study



The purpose of this chapter is to give the reader an introduction to relevant background information. As explained in the introduction, the reader is assumed to have experience from the field of software engineering and to be familiar with Maven [4]. Together with the topics presented in this chapter, this makes up the technological baseline for the rest of the report.

The chapter begins with a section that defines important terminology, followed by an introduction to continuous integration and its benefits. Further, section 2.3 contains a state-of-the-art survey. Its purpose is to determine if our proposed extension or something similar already exists. Maven's Project Object Model (POM) and versioning scheme is explained next, before we describe Continuum's architecture and summarise the technology employed. Finally, we present a recipe for how a development environment for Continuum can be set up.

2.1 Important terminology

Concise terminology is important to avoid misconceptions. In the following, we will describe terms that are used throughout this report. The purpose is not to provide formal definitions, but to explain how the terms should be interpreted in this report.

A service is a software product that offers a set of features usable by a user or another software product. We will often use the term *dependency* synonymously with service, and we will call a service packaged as a releasable entity for an *artifact*.

Build - the process of converting source code into standalone software artifacts that can be run on a computer. A build can include different steps, but here we will assume that at least compilation and unit tests are performed.

Integration - the process of piecing together the different parts of an application. The purpose is to check interoperability between sub-components or with other systems. A typical example is integration with the Database Management System (DBMS) that will be used in production.

Versioning is a way to "label" a certain build. Different projects use different approaches and different definitions, but we will stick to the following:

> *"Software versioning is the process of assigning either unique version names or unique version numbers to unique states of computer software."* [5]

A versioning *scheme* is thus a description of which label is appropriate given a certain change. A description of the Maven versioning scheme can be found in section 2.5.

Backward compatibility is defined by the Free Online Dictionary of Computing [6] as *"Able to share data or commands with older versions of itself, or sometimes other older systems, particularly systems it intends to supplant."* For a *service*, this means that projects that depend on this service can expect that functionality written for the old API will still work with the API of the new version.

What is continuous integration?

Continuous integration is perhaps best known as one of the twelve practices of eXtreme Programming (XP);

> ***Continuous Integration*** *- "Integrate and build the system many times a day, every time a task is completed."* [7, p.54]

We will, however, use a more descriptive definition of the term written by Foemmel and Fowler:

> *"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible."* [8]

2.2 Continuous Integration

Continuous Integration (CI) is not a new concept. Custom solutions have probably been around for decades. However, the earliest *formal* description of the concept we have found is Steve McConnell's *daily build and smoke test* practice [9]. He states that the system should be built daily, with subsequent integration testing (smoke test). If we extend this practice by running more frequent, automated builds, and replace the simple smoke test with a comprehensive test suite, we get what we today consider continuous integration.

The Chrysler Comprehensive Compensation project [10] is assumed to be the first known application of modern continuous integration. This assumption is based on the fact that both Kent Beck's *eXtreme Programming eXplained* [7], and Matthew Foemmel and Martin Fowler's article [8] on the subject, reference this project. In addition, famous continuous integration servers like CruiseControl [11] and AntHill [12] were first released in 2001, i.e., 1-2 years after these publications.

2.2.1 Benefits of using continuous integration

This summary is based on the "Benefits of Continuous Integration" from Foemmel's and Fowler's article [8].

Reduced risk

When integration is done only at the end of the project it is impossible to known how difficult it will be and how long it will take. This impose a huge risk affecting delivery cost and schedule. Many small problems distributed over the duration of the development is better than many big problems right before delivery. When continuous integration is used, integration problems are discovered when they are introduced and can be solved consecutively. Continuous integration can thus reduce the risks related to integration.

Easier to find bugs

Continuous integration techniques can be used to run tests every time a new task is completed. This will limit the volume of code that must be reviewed to find a potential bug and allows *diff debugging*. The term diff debugging come from the Unix tool *diff* which compares two text files and reports the differences line by line. The term is used on the practice of comparing the code of the last successful build with the code in the new, unsuccessful build to deduct where the bug might have been introduced.

There is also a psychological phenomenon, known as the *Broken Window Theory*[13, chap1.2], to consider. The idea is that bugs are cumulative and that the effort required to fix a bug increases when a failure is caused by multiple faults. Allowing the number of bugs to escalate is thus demotivating to the developers.

A good (comprehensive) test suite is a fundamental requirement of continuous integration, and it affects bug detection and bug tracking directly. Thus, the better the tests, the greater the benefits.

Frequent deployment

"Keep your project releasable at all times." [2, Tip 13]

Continuous integration makes the status of the build plainly visible. If you trust your tests, then this status should be a fairly good indication as to whether the software is ready for deployment or not. This has two advantages:

1. A continuous integration server can be used to deploy the software to a test server after each successful build. This allows rapid feedback from the customer and can be used as basis for discussion between customer and development team. In other words, this practice support agile work patterns and improves collaboration.
2. Frequent builds and frequent deployment reduce the time needed for Quality Assurance (QA) and testing prior to each release. Continuous integration can thus facilitate shorter release cycles and reduce time to market.

2.3 Existing implementations

It is hard to *prove* that a solution to the problem described in the introduction does not exist, but we will show that it is unlikely. For convenience we will use the term ***our suggested extension*** to mean functionality that solve the problem in the introduction.

2.3.1 Search for references

The shallow descriptions found in books on XP [7, 14] and on agile development [1, 2, 3], indicate that continuous integration has not been subjected to extensive research. Searches on “continuous integration” on the ACM Digital Library [1] , IEEE Xplore [2] and Google Scholar [3] support this suspicion, as little comprehensive research on continuous integration was found. The only article we found related to advanced continuous integration concepts was *Continuous release and upgrade of component-based software* [15]. This article explain how to use continuous integration to continuously publish release artifacts and is thus not relevant.

2.3.2 Comparison matrix

Continuous Integration Server Feature Matrix [16] contains a comparison of several continuous integration servers. *Our suggested extension* is not mentioned among any of the features compared, and it is comprehensive list. We believe it is unlikely that this kind of functionality would be omitted on purpose, if any implementations were known. This leads us to the conclusion that it is unlikely that any of the nineteen implementations listed provide anything similar to our suggested extension.

2.3.3 Expert statements

Trygve Laugstøl is a core developer on Maven and Continuum. He is currently employed at Objectware AS [4] , a Norwegian consulting firm. He has worked with software development for many years and has lots of experience with Open Source Software (OSS). T. Laugstøl states that he has not heard of any continuous integration server that contains *our suggested extension*. Discussions with other prominent developers in the open source community all support T. Laugstøl’s statement.

[1] `http://portal.acm.org` Last visited: 2007.05.29
[2] `http://ieeexplore.ieee.org` Last visited: 2007.05.29
[3] `http://scholar.google.no` Last visited: 2007.05.29
[4] `http://objectware.no` Last visited: 2006.01.28

2.3.4 Spot test

We researched three continuous integration servers, in addition to Continuum, as a final verification. These three have been chosen because they were mentioned in the resources[5] we used when we researched continuous integration for section 2.2, or because they have been recommended by friends or colleagues.

CruiseControl

CruiseControl was registered at SourceForge in 2001 [11] and is considered the *granddaddy of continuous integration servers* [17]. CruiseControl has an Ant-based *Build Loop*, which periodically checks a Version Control System (VCS) for changes, builds if necessary and sends notifications with the status of the build. A single XML file is used for configuration, and CruiseControl depends on plug-ins for the actual functionality. This makes CruiseControl flexible at the cost of complex configuration and setup. As of time of writing, neither the Default Plugin Registry [18], nor the list of 3rd party plug-ins [19], include a plug-in that support *our suggested extension*.

Anthill

AntHill was released in July 2001 and was originally OSS. We have looked at the commercial version, AntHill Pro, because it has the most features. According to their own documentation [20] the only supported continuous integration functionality are

- Repository Commit Triggers
- Configurable Quiet Period
- Integration with Testing & Code Coverage Tools
- Robust Notification Schemes
- IDE plug-in

It is thus reasonable to assume that AntHill does not contain *our suggested extension*.

TeamCity

TeamCity is a modern tool which JetBrains describe as an *integrated team environment*. Nothing is mentioned related to *our suggested extension* in their published feature matrix [21]. It is thus reasonable to assume that TeamCity does not provide this functionality either.

[5]Research resources: [7, 8, 13, 2]

2.4 Maven Project Object Model

This section will introduce Mavens' Project Object Model (POM) and describe elements relevant to the prototype. General introductory material can be found in [4, 22], or at the Maven website [6] , and will not be covered here.

2.4.1 POM introduction

The Maven Project Object Model (POM) is an XML representation of a Maven project. This configuration is stored in a single file named `pom.xml` and is defined by the Maven 4.0.0 XML schema definition [7] . The justification for using a single file is that it is necessary to ensure that each Maven 2 artifact is an easily portable unit.

Maven can be considered a build *framework* more than a dedicated build tool. The POM is the heart of this framework, where configuration, for all scripts/plug-ins a build environment needs, is consolidated. The following sections will explain the elements most relevant to the prototype.

2.4.2 Relevant elements

This section will present elements directly related to continuous integration and explain what they are used for.

modules element

A project can be divided into multiple modules. When the parent project is added to Continuum, the parent and all modules are added as separate projects[8]. Any inheritance is also resolved, so from the viewpoint of Continuum each project is complete and can be built separately. This is reflected in the default arguments; *–batch-mode –non-recursive*, which ensures that Maven do not build modules when the parent project is built.

scm element

The Source Code Management (SCM)[9] element is used by the maven-site-plugin [10] , the maven-release-plugin [11] and Continuum. An example is shown in listing 2.1.

[6] `http://maven.apache.org` Last visited: 2007.05.29

[7] `http://maven.apache.org/maven-v4_0_0.xsd` Last visited: 2007.05.29

[8]Projects are added to the same projectGroup, but the concept of a projectGroup is not important in this context.

[9]SCM is used synonymously with Version Control System (VCS) throughout this report.

[10] `http://maven.apache.org/plugins/maven-site-plugin` Last visited: 2007.05.30

[11] `http://maven.apache.org/plugins/maven-release-plugin` Last visited: 2007.05.30

```
<scm>
  <connection>scm:svn:http://svn.someCompany.com/someProject/trunk</connection>
  <developerConnection>
    scm:svn:svn+ssh://svn.someCompany.com/store/svn/someProject/trunk
  </developerConnection>
  <url>http://someCompany.com/view.cvs</url>
</scm>
```

Listing 2.1: SCM configuration

The maven-release-plugin use this information when performing SCM operations on the repository during releases.

The maven-site-plugin use this information as basis for its *Source Repository* report. This report shows the different ways to access the SCM repository.

Continuum utilize the SCM information to check out the project from the Version Control System (VCS) when a project is added to Continuum and when Continuum checks the repository for updates. The fact that any project can specify the SCM information in the Maven POM is one of the reasons why it is so easy to add a new Maven project to Continuum. Extra configuration is only needed if the defaults for schedule, build goals or build arguments are unsatisfactory.

ciManagement element

The ciManagement element holds information on which continuous integration server is set up for the project. The listing below shows an example configuration.

```
<ciManagement>
  <system>Continuum</system>
  <url>http://ci.someCompany.com</url>
  <notifiers>
    <notifier>
      <type>mail</type>
      <configuration>
        <address>dev@someCompany.com</address>
        <sendOnSuccess>false</sendOnSuccess>
      </configuration>
    </notifier>
    <notifier>
      <type>irc</type>
      <configuration>
        <host>irc.codehaus.org</host>
        <port>6667</port>
        <channel>#maven</channel>
      </configuration>
    </notifier>
  </notifiers>
</ciManagement>
```

Listing 2.2: ciManagement configuration

The *system* and *url* elements are used by the maven-site-plugin as basis for a Project Information report named *Continuous Integration*. The *notifiers* section specifies where Continuum shall send feedback after a build. It is also possible to specify which state changes a notifier is interested in. E.g., do not send email on successful builds, as shown in the previous listing; *<sendOnSuccess>false</sendOnSuccess>*.

dependency element

The *dependency* element specifies that a project depends on an artifact. The identifier used is on the form $< groupId >:< artifactId >:< version >$ and is called coordinates. In addition, type and scope can be specified. Type designates how the artifact is packaged, e.g., jar, war, ear, etc. Scope specifies in which scope the dependency is included. An example is shown in listing 2.3, where a dependency to JavaServer Pages Standard Tag Library is defined. This artifact is a jar file identified by $javax.servlet : jstl :$ 1.1.2, and it is available in the runtime scope.

```
<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.1.2</version>
    <scope>runtime</scope>
    <type>jar</type>
  </dependency>
  ...
<dependencies>
```

Listing 2.3: jstl dependency

Coordinates are used when a project needs to reference another project/artifact, typically when dealing with aggregation (multimodule projects), inheritance and dependencies. These concepts are documented in [4, 22], and summaries of dependency scope, transitive dependencies and DependencyManagement can be found in [23, chap. 3.4.2] or in *Introduction to the Dependency Mechanism* [24].

Continuum use the *dependencies* element to elucidate which artifacts, which *services*, a project depends on. This metadata makes it possible for the prototype to create new derived projects identical to the original project, except for a different set of dependencies. Metadata that uniquely identifies dependencies is a prerequisite for the prototype. Combined with the Maven versioning scheme, described in section 2.5, the version attribute makes it possible to create an ordering of dependencies. I.e., the version attribute makes it possible to determine if service A is newer than service B.

2.4.3 Summary

To our knowledge, the metadata described here is only available with the Maven POM. A prototype not based on this POM is thus not feasible, and the prototype should be based on a continuous integration server that utilize Maven projects.

2.5 Maven versioning scheme

Maven uses the same versioning scheme as the XStream project. This scheme can be found in on the XStream home page [25] and in *Better Builds with Maven* [4, p.60] and complies with best practices for agile development, as described by the Open-Closed Principle (OCP), [1, chap.9]. This scheme is on the form *major.minor.micro* and is used for *final* releases. Non-final releases can use *major.minor.micro-qualifier*, while snapshots can also add a build number; *major.minor.micro-qualifier-buildNumber*. The examples in table 2.1 illustrates how this versioning scheme can be used.

Label	**Type of build**
1.2.3-beta-3	build 3 of 1.2.3-beta (snapshot, not a release)
1.2.3-rc	normal release prior to final release
1.2.3-2	normal release prior to final release
1.2.3	final release
1.2.6	final release with more functionality than 1.2.3
2.0.0	final release not backwards compatible with 1.x.y

Table 2.1: Maven versioning scheme examples

There are probably several versionings schemes were the examples mentioned are valid. To correctly apply the Maven versioning scheme, the following interpretation of major, minor, micro, qualifier and buildNumber should be used.

- **Major** identifies the API version. The major version is incremented when the backwards compatibility of the API is broken. Minor and patch versions are then reset to zero.
- **Minor** identifies the backwards compatible versions of the API. Revisions can add new elements to the API, but the original elements cannot be modified. The patch version is reset to zero when the minor version is incremented. Note, elements may be annotated *deprecated* when the minor version is incremented, but not removed until the major version is incremented.
- **Micro** identifies bug fixes and other changes that does not affect the API.
- **Qualifier** identifies a version prior to a final release. The qualifier can be a text string or an integer, e.g. alpha, beta or rc (release candidate).
- **Build number** is an increment used to identify different patches of a build and is not applicable for a normal release.

2.6 Continuum

Continuum is an open source continuous integration server released under the Apache 2.0 license [26]. It is designed especially for Maven, but Ant and shell projects are also supported. This chapter will give an overview of the architecture and technology relevant to the implementation. We will also explain how to set up a development environment and describe packages relevant to the prototype.

2.6.1 Architecture

Continuum has a *component-oriented*[12] architecture where each component has its own Maven module, and each module has one logical responsibility. This improves reuse and support easy "plugability", as illustrated by the Apache Geronimo project GBuild. GBuild [13] is an extension to Continuum which adds support for distributed integration. GBuild is implemented as a Plexus application, reusing components like continuum-store, continuum-notification, etc. See Plexus configuration for GBuild [29] for details.

Relevant components

The latest stable release of Continuum, version 1.0.3, is comprised of 18 components. The components relevant to the prototype are

- **continuum-core** - core functionality
- **continuum-model** - the data model
- **continuum-plexus-application** - IoC-container
- **continuum-store** - persistence
- **continuum-web** - web application used for administration

Descriptions of the classes relevant to the prototype can be found in section 4.2.1.

2.6.2 Technology

When writing an extension to an existing product it is essential to have a basic understanding of the technology employed. We will therefore give a short overview of the technology relevant to the prototype and explain how it is utilized. For detailed documentation the reader is referred to their respective web pages.

[12] See [27] for a quick introduction to components or [28] for a more in-depth approach.

[13] http://cwiki.apache.org/gbuild Last visited: 2007.05.28

Plexus

Plexus is an Inversion of Control (IoC) container similar to for example the Spring Framework. According to the Plexus home page [14] , the following features are found in Plexus, but not in Spring:

- Component life cycles
- Component instantiation strategies
- Nested containers
- Component configuration
- Auto-wiring
- Component dependencies, and
- Various dependency injection techniques including constructor injection, setter injection and private field injection.

These features are not listed because the reader is assumed to know Spring, but because the list indicate what kind of functionality Plexus provides. Explaining how Plexus works is out of scope for this thesis. However, it is not difficult to use the IoC functionality. The first step is to define a Plexus Component Descriptor in `components.xml`, and as the example below shows, the syntax is simple and intuitive.

```
<component>
  <role>org.apache.maven.continuum.DerivedProjectManager</role>
  <implementation>
    org.apache.maven.continuum.DefaultDerivedProjectManager
  </implementation>
  <requirements>
    <requirement>
      <role>org.apache.maven.continuum.DependencyHelper</role>
    </requirement>
  </requirements>
</component>
```

Listing 2.4: Excerpt from components.xml

Listing 2.4 shows that the current implementation of the interface `DerivedProjectManager` is `DefaultDerivedProjectManager` and that it requires a `DependencyHelper`. The next step is the corresponding declaration in `DefaultDerivedProjectManager.java`:

```
/**
 * @plexus.requirement
 */
private DependencyHelper dependencyHelper;
```

Listing 2.5: plexus.requirement

This is all that Plexus needs to wire in the `DependencyHelper` component, no setter is required. A guide that explains what the Plexus Component Descriptor is and how to use it can be found the reference documentation [30].

[14] `http://plexus.codehaus.org` Last visited: 2007.02.13

JDO and JPOX

Continuum uses the Java Data Objects (JDO) [15] specification for persistence. JDO is a *standard* specifying which API an implementation must support. No restrictions are put on the underlying data store, which data stores to support is entirely up to the implementation.

Continuum uses a JDO implementation named Java Persistent Objects (JPOX) [16] . JPOX supports JDO 1 and 2 and support for Java Persistence API (JPA) is planned in version 1.2 [17] .

In the latest stable release of Continuum, version 1.0.3, persistence is based on a snapshot of JPOX 1.1, version 1.1.0-20060413. The *continuum-store* component is responsible for this.

Modello

The *Modello* Data Model toolkit is used to generate Continuum's data model. An excerpt from Continuum's Data Model, the definition of ProjectDependency, is listed below.

```
<class>
  <name>ProjectDependency</name>
  <version>1.0.0+</version>
  <fields>
    <field>
      <name>groupId</name>
      <version>1.0.0+</version>
      <type>String</type>
    </field>
    <field>
      <name>artifactId</name>
      <version>1.0.0+</version>
      <type>String</type>
    </field>
    <field>
      <name>version</name>
      <version>1.0.0+</version>
      <type>String</type>
    </field>
    <field>
      <name>derived</name>
      <version>1.0.0+</version>
      <defaultValue>false</defaultValue>
      <type>boolean</type>
    </field>
  </fields>
</class>
```

Listing 2.6: Excerpt from continuum-model/src/main/mdo/continuum.mdo

The advantage of using a data model toolkit is that the model can be used as basis for multiple models. According to the Modello home page [18] any of the following models

[15] http://java.sun.com/products/jdo Last visited: 2007.02.13
[16] http://www.jpox.org Last visited: 2007.02.13
[17] http://www.jpox.org/docs/jpox_why.html Last visited: 2007.02.13
[18] http://modello.codehaus.org Last visited: 2007.05.13

can be generated:

- **Java Pojos of the DataModel.**
- **Java Pojos to XML Writer.** (provided via xpp3, stax, jdom or dom4j)
- **XML to Java Pojos Reader.** (provided via xpp3, stax or dom4j)
- XDOC documentation of the DataModel.
- XML Schema to validate the DataModel.
- Java Model to Prevayler Store (actually this plugin is in the sandbox).
- Java Model to JPOX Store.
- **Java Model to JPOX Mapping.**

The emphasized models are those utilized by Continuum. Java Pojos are used in continuum-model, XML Writer/Reader in continuum-xmlrpc and JPOX Mapping in continuum-store. Using Modello minimizes redundancy, at the cost of a bit more complex setup.

2.6.3 How to build Continuum from source

This section will explain how to set up a development environment for Continuum and build and run the application. This recipe was included due to the complexity of the setup procedure.

Requirements

Before you can start on the actual procedure it is important to make sure your environment satisfies the requirements. Java, Maven 2 and Subversion is mandatory, but Maven 1, Ant and CVS are only needed by the integration tests and can be skipped. The following software[19] should be installed:

- Java Development Kit [20]
- Maven 2
- Subversion console client
- Maven 1
- CVS console client
- Ant

[19]Links to the software can be found in appendix B.

[20]We had trouble with Sun Java 6 and java-gjc, while Sun Java 1.5 worked fine.

It is possible to use the Windows operating system, but a operating system with a more sophisticated shell is recommended, because the software mentioned above must be run from the console.

After installation and configuration is completed, verify that everything is set up correctly by comparing your output with the output listed below. A line starting with $ is a command and the subsequent lines are the output.

```
$ java -version
java version "1.5.0_08"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_08-b03)
Java HotSpot(TM) Server VM (build 1.5.0_08-b03, mixed mode)

$ mvn -version
Maven version: 2.0.4

$ svn --version
svn, version 1.3.2 (r19776)
```

Listing 2.7: Mandatory software

```
$ cvs -version
Concurrent Versions System (CVS) 1.12.13 (client/server)

$ ant -version
Apache Ant version 1.6.5 compiled on July 5 2006

$ maven --version
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0.2
```

Listing 2.8: Software required for integration tests

Tip: If any of the commands does not return output similar to this listing, check that $PATH and $JAVA_HOME are set correctly.

Checking out from version control

Continuum uses Subversion, and we want the latest stable release, which currently is version 1.0.3. Check out from the console to a folder of you choice. E.g.;

```
mkdir continuum-1.0.3
cd continuum-1.0.3
svn co http://svn.apache.org/repos/asf/maven/continuum/tags/continuum-1.0.3 .
```

Build the project

Try to build the project with the following command:

```
mvn -Denv=test install
```

The command probably has to be run multiple times to download all dependencies, and several *javax.** dependencies must be downloaded and installed manually due to licensing issues. Directions for manual installation is given when the build fails. In addition, one transitive *snapshot* dependency is no longer available. A workaround is to rename the final release and install it manually as the snapshot.

```
wget http://repo1.maven.org/maven2/org/codehaus/plexus/plexus-appserver/1.0-alpha-5/
    plexus-appserver-1.0-alpha-5.jar
mvn install:install-file -DgroupId=org.codehaus.plexus -DartifactId=plexus-appserver
-Dversion=1.0-alpha-5-SNAPSHOT -Dpackaging=jar
-Dfile=plexus-appserver-1.0-alpha-5.jar
```

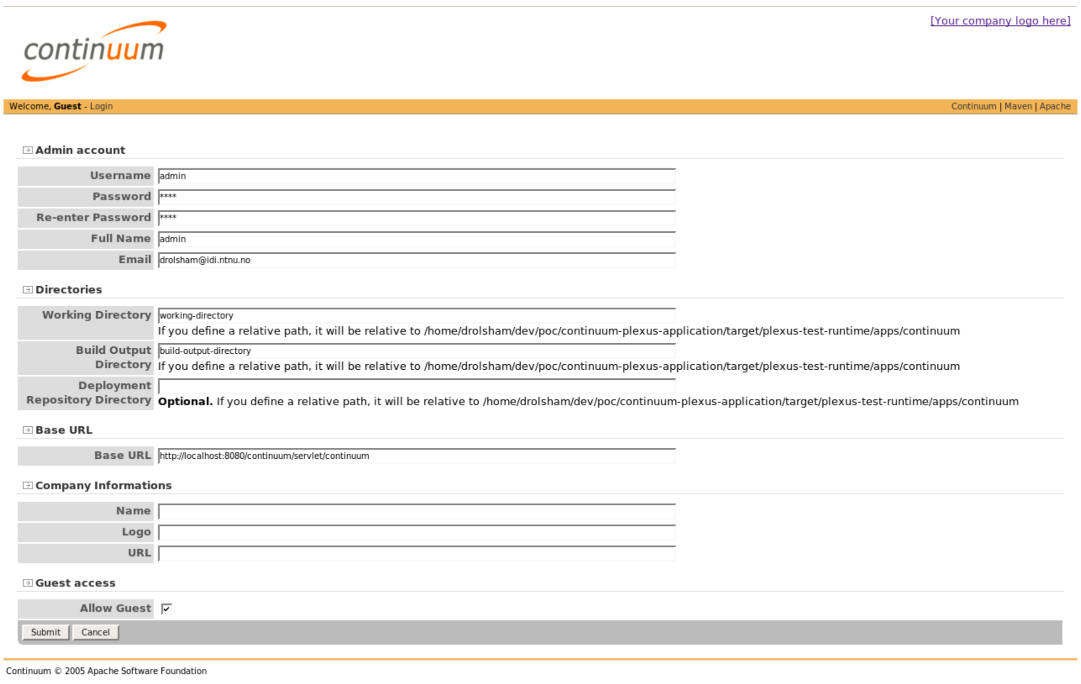
Tip: If the integration tests fail, just comment out *continuum-core-it* from the modules-subsection in the parent pom. These tests are not critical for prototyping.

Start the web application

After the installation is finished, bundle and start the web application with the following commands:

```
cd continuum-plexus-application/
mvn plexus:app plexus:bundle-application
target/plexus-test-runtime/bin/plexus.sh
```

A successful deploy will start a web application at `http://localhost:8080/continuum`. Figure 2.1 shows what to expect. After an administration account and other settings are configured, it is possible to log in and start using Continuum.



Admin account

Username	admin
Password	****
Re-enter Password	****
Full Name	admin
Email	drolsham@idi.ntnu.no

Directories

Working Directory	working-directory If you define a relative path, it will be relative to /home/drolsham/dev/poc/continuum-plexus-application/target/plexus-test-runtime/apps/continuum
Build Output Directory	build-output-directory If you define a relative path, it will be relative to /home/drolsham/dev/poc/continuum-plexus-application/target/plexus-test-runtime/apps/continuum
Deployment Repository Directory	**Optional.** If you define a relative path, it will be relative to /home/drolsham/dev/poc/continuum-plexus-application/target/plexus-test-runtime/apps/continuum

Base URL

Base URL	http://localhost:8080/continuum/servlet/continuum

Company Informations

Name	
Logo	
URL	

Guest access

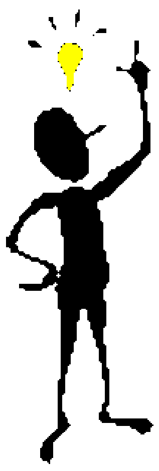
Allow Guest	✓

Continuum © 2005 Apache Software Foundation

Figure 2.1: Initial setup

Chapter 3

Own contribution



In this section we will describe a solution to the problem described in the introduction. First we list a set of assumptions. Second, we describe rules governing how the prototype should react to different use cases. These rules are the actual, conceptual solution and explain when and what the prototype shall do. Examples with dependency graphs are used to support the description and show how the concept can be applied in practice. We have also derived a formula for the complexity. The formula calculate worst case and show how the number of combinations (and thereby derived projects) grow as the number of projects or services grow.

3.1 Conceptual solution

This section will describe a solution to the problem described in the introduction.

3.1.1 Assumptions

To limit the scope of possible solutions we have identified a list of constraints/assumptions. These assumptions are adapted to Continuum's abstract model, but they are not implementation specific. The conceptual solution should thus also be useful for other continuous integration servers which satisfies the same assumptions. We assume that the underlying build engine

- has the concept of a *project*.
- can build an arbitrary number of projects.
- supports the concept that a project depends on $0-n$ services, where a service[1] can be unambiguously specified as a dependency found in Maven, i.e., specified by $< groupid >:< artifactid >:< version >$.

3.1.2 Solution description

The idea is to create a new derived project for each new, unique combination of dependencies. Combinations are found by swapping dependencies from the original list of dependencies with dependencies from the list of derived dependencies. A swap can be made if the derived dependency has the same groupid and artifactid as the original dependency and a newer version. Whether a dependency is newer/higher than another is determined by the versioning scheme described in section **??**. The original dependency might be called *relevant*, but we will also use the term *original* if it can avoid misunderstandings. The list of active derived dependencies, the dependencies which are considered for a swap, is termed the *input list*.

3.1.3 Rules

There are six possible changes that should invoke an action; add/remove projects, add/remove dependencies from a project and add/remove dependencies from the list of derived dependencies. The changes and their corresponding actions are as follows:

1. *A new project is added* $\rightarrow$ new derived projects should be created if its dependencies are relevant.
2. *A project is removed* $\rightarrow$ the original project and all its derived projects should be deleted.

[1]Hereafter the term "service" is used synonymously with "dependency" unless explicitly stated otherwise.

3. *A new dependency is added to the original project* $\rightarrow$ new derived projects should be created if the new dependency is relevant.
4. *A dependency is removed from the original project* $\rightarrow$ all affected derived projects should be deleted.
5. *A new dependency is added to the input list* $\rightarrow$ new derived projects should be created if the new dependency is correspondent with a relevant dependency in the original project.
6. *A dependency is removed from the input list* $\rightarrow$ all affected derived projects should be deleted.

Examples, with representations of how the graphs change, can be found in section 3.2.

3.1.4 Complexity

According to the proposed solution we get two sets of projects; the original projects and the derived projects. The number of derived projects depend on how many relevant dependencies each original project have. Worst case is that all projects depend on all services. Let *P* be the number of original projects, *S* the number of services and *P'* the number of new derived projects. If worst case is assumed, the number of derived projects follows formula 3.1.

$$P' = P * \sum_{s=1}^{S} 2^{s-1} \tag{3.1}$$

Calculation examples

We will here run through a few examples to demonstrate the rationale behind formula 3.1.4.

Example one: Four projects ($P = 4$) all depend on two dependencies ($S = 2$). For each project there are four possible combinations of dependencies; S1 & S2, S1' & S2, S1 & S2' and S1' & S2'. Only three of these combinations are new, since the first is the original combination S1 & S2.

The number of new derived projects are thus $4 * 3 = 12$, which is equal to the result we get by using formula 3.1.

$P' = 4 * \sum_{s=1}^{2} 2^{s-1} = 4(2^0 + 2^1) = 12.$

Example two: Let us increase the number of derived dependencies to six, to illustrate why automation is necessary. I.e., $P = 4$ and $S = 6$, still using formula 3.1:

$P' = 4 * \sum_{s=1}^{6} 2^{s-1} = 4(2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5) = 4 * 63 = 252$

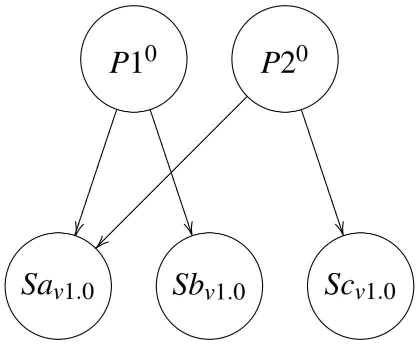
Testing all these combinations manually is clearly not an enticing solution!

3.2 Examples

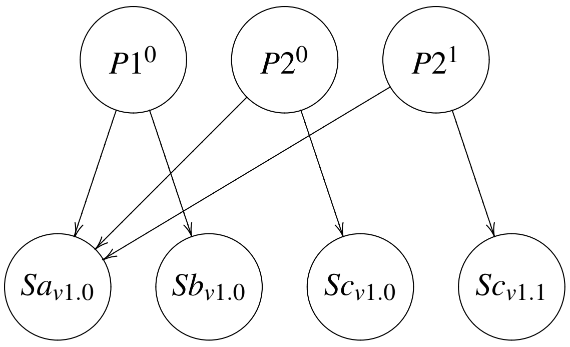
We will illustrate how the dependency graphs change with two examples. The first example is simple and its purpose is to provide a graphical representation to make the solution from section 3.1 easier to understand. The second example is bigger and more realistic and will be used as input to the prototype to validate the implementation.

3.2.1 Example 1

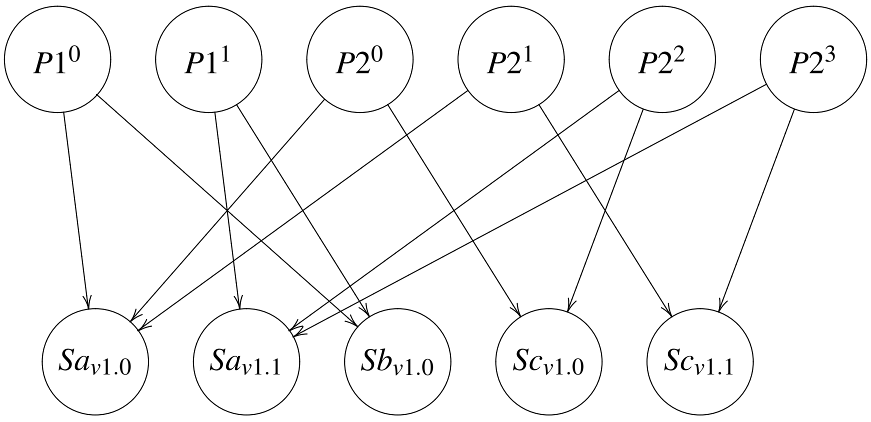
We have two projects, $P1^0$ and $P2^0$. The superscript notation is used to enumerate the generated derived projects. 0 indicate that these projects are original projects. Project P1 depend on two services, Sa and Sb. Project P2 depend on Sa and Sc. The input graph:



If we add a new version of Sc, say Sc v1.1, to the input list, then a new *derived* project, $P2^1$, should be added. This is a new variant of project P2 and it depends on Sa (as the original project P2) and on the new version of Sc. The graph below illustrates this.



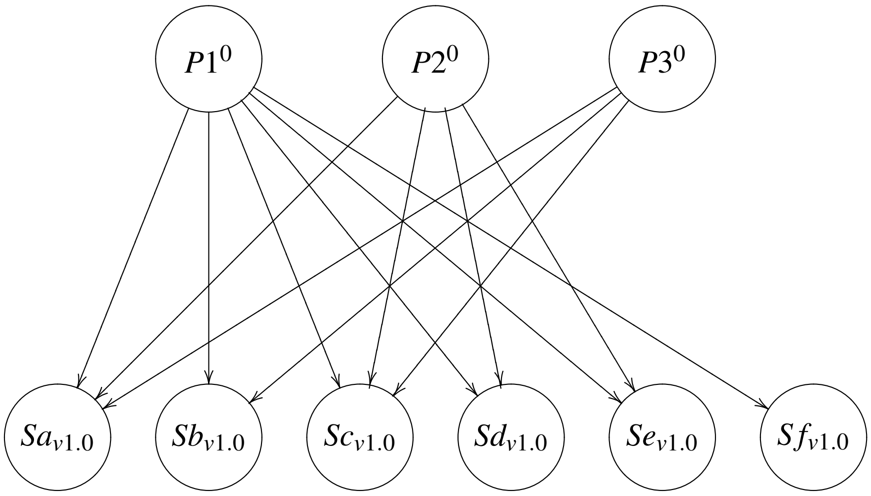
When a new version of Sa is added as well, three additional projects should be generated: $P2^2$, $P2^3$ and $P1^1$. New graph:



Adding two new dependencies to the input list will thus generate $P1^1$,$P2^1$,$P2^2$ and $P2^3$, for a total of four derived projects.

3.2.2 Example 2

We have three projects; P1, P2 and P3. P1 has dependencies Sa-Sf, P2 has dependencies Sa,Sc,Sd,Se and P3 has dependencies Sa-Sc. The input graph is shown below.



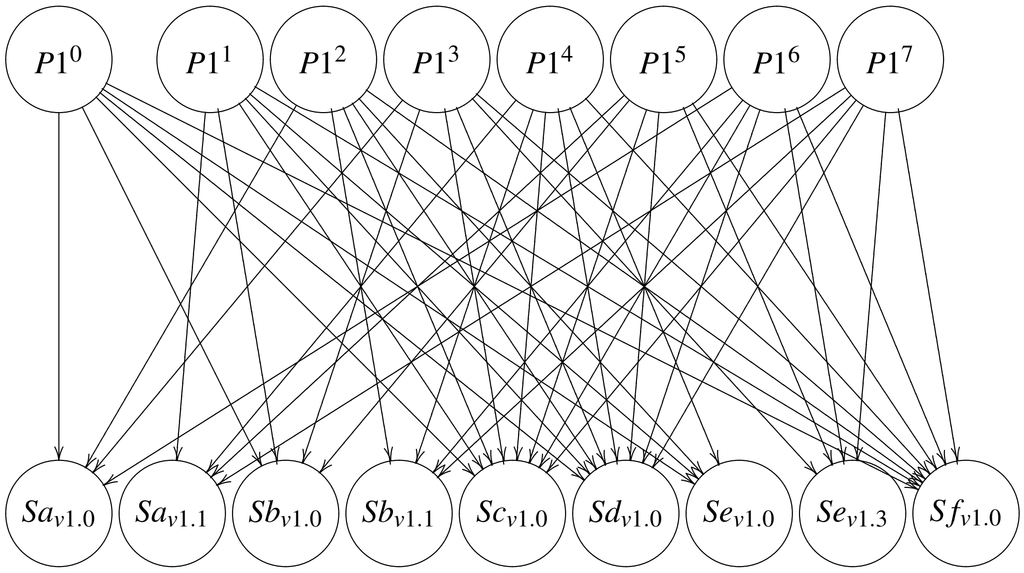
We introduce new versions of the dependencies **Sa, Sb** and **Se**. The new combinations are shown in table 3.1. For convenience **'** is used to denote a derived dependency.

P1	**P2**	**P3**
Sa', Sb, Sc, Sd, Se, Sf	Sa', Sb, Sc, Sd, Se	Sa', Sb, Sc
Sa, Sb', Sc, Sd, Se, Sf		Sa, Sb', Sc
Sa, Sb, Sc, Sd, Se', Sf	Sa, Sb, Sc, Sd, Se'	
Sa', Sb', Sc, Sd, Se, Sf		Sa', Sb', Sc
Sa', Sb, Sc, Sd, Se', Sf	Sa', Sb, Sc, Sd, Se'	
Sa, Sb', Sc, Sd, Se', Sf		
Sa', Sb', Sc, Sd, Se', Sf		

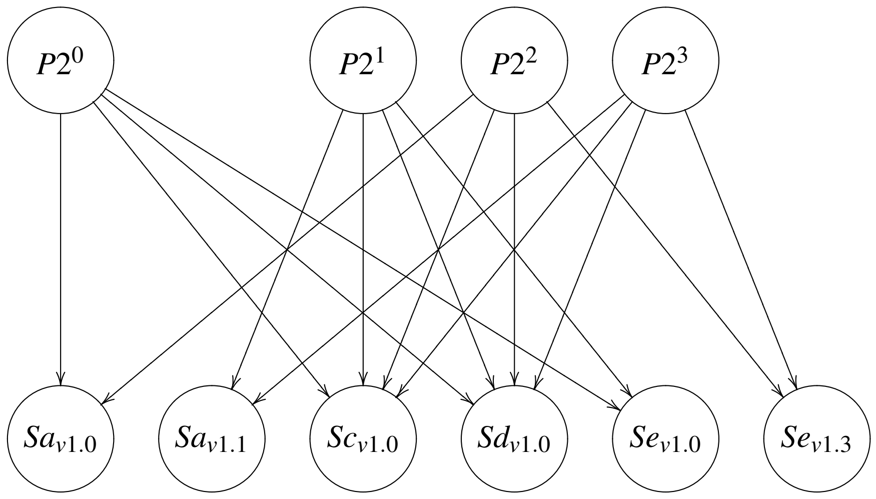
Table 3.1: 13 new combinations

In the following, the graphs for P1, P2 and P3 is shown. Note that the derived depencies are not marked with the shorthand **'** anymore, instead the derived dependencies can be distinguished from the original dependencies by the difference in version.

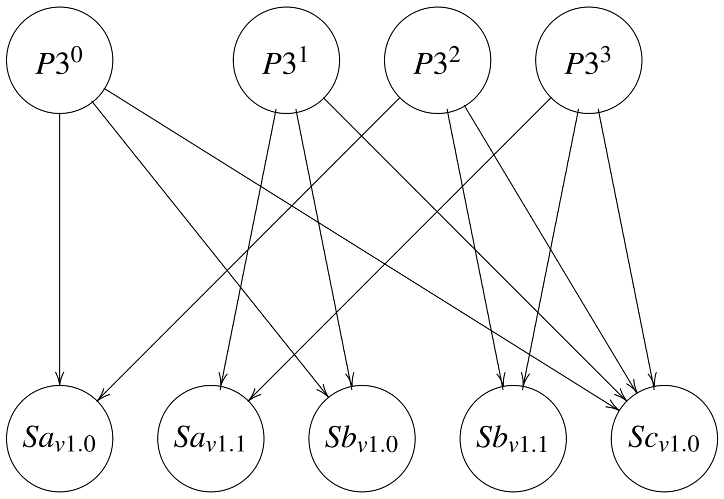
The graph for project P1 shows the original P1 project and seven derived projects.



The graph for project P2 shows the original P2 project and three derived projects.



The graph for project P3 shows the original P3 project and three derived projects.

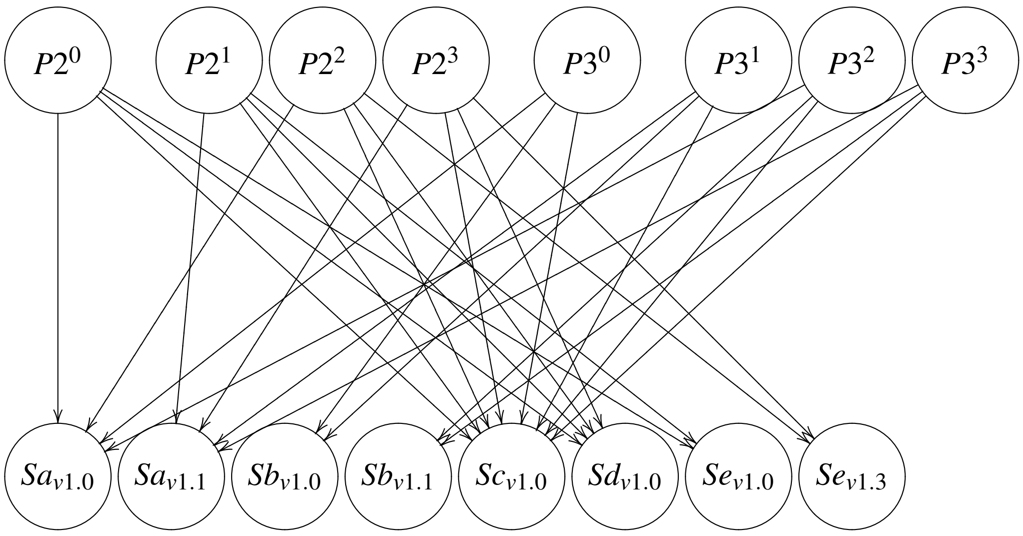


3.2.3 Example 3

The purpose of this example is to show what happens when an original project is deleted. The three final graphs from example 3.2.2 will be used as input graphs for this example. The combined input graph is not shown, because it is too big and complex to fit the page. According to the rules in section 3.1.2, all derived projects of an original project should be deleted when the original project is deleted. Deleting a derived project does not affect any other projects.

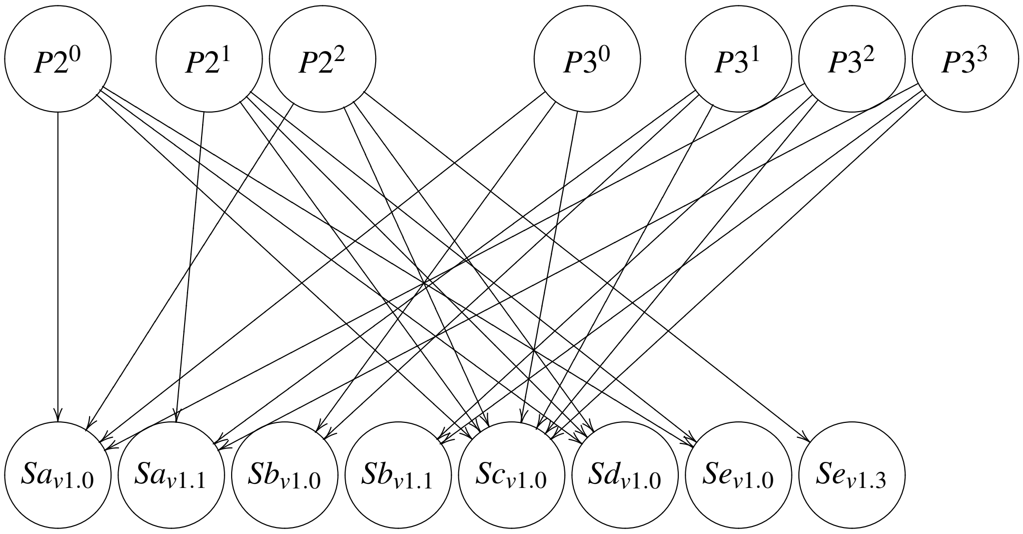
Deleting an original project

When the user deletes $P1^0$, all its derived projects should also be removed, i.e., $P1^1 - P1^7$. The combined output graph is equivalent with the graphs for P2 and P3 from example 3.2.2 and is shown on the next page.



Deleting a derived project

When a *derived* project is deleted, no other projects are removed. The output when $P2^3$ is deleted is shown below.



3.3 Summary

We have described a solution which rely on finding all combinations of dependencies, where one combination corresponds to one derived project. Six rules describe how and when the set of derived projects must change. We have also listed a formula for worst case complexity to illustrate that the number of combinations grows quickly. Examples, with dependency graphs, are used to illustrate creation and removal of derived projects.

Chapter 4

Implementation

This chapter consists of two parts; a strategy for solving the problem with Continuum and a description of the proof of concept implementation. We will also provide screen shots from the implementation to illustrate the connection between concept and implementation. Example 2, described in chapter 3, is used as basis for the screen shots to show that the prototype correctly implements the concept.

4.1 Strategy

We have found three approaches to implement the solution proposed in section 3.1:

1. use Continuum's XML-RPC API
2. build a new application with components from Continuum (like GBuild, [29])
3. modify the existing code base directly

We have chosen the third option, because this will make the extension an integrated part of Continuum. Integration is beneficial because it will make the extension easily accessible, without extra downloads or configuration. Additionally, a separate application would have to be updated whenever a new version of Continuum was released, integration with Continuum avoids this problem. The greatest disadvantage of this approach is that it is intrusive. However, this can be mitigated by enforcing clean separation of concerns and by adding enable/disable functionality.

We will now explain how we plan to implement this solution. Descriptions of the emphasized classes can be found in section 4.2.1.

4.1.1 Divide and conquer

The process of building a project with Continuum is twofold. The first part checks for changes in the VCS, update/create the `ContinuumProject` and saves the project to persistent storage. The second part loads the project from persistent storage and executes the Maven executable. The execution use the `pom.xml` from the checked out project on the file system, while goals and arguments according to the project's `BuildDefinition`s are provided by Continuum. This means that when we have successfully created and stored derived projects, Continuum takes care of the building.

However, we need to modify the original `pom.xml` to build the derived projects with the correct dependencies. We also want the extension to be transparent to the VCS. One approach would be to utilize the *revert* functionality often found in version control systems, but this limits the implementation to version control systems that have this functionality. A better approach is to implement revert with functionality from the `java.io` package. This approach depend only on Java, not on any external VCS. The proposed solution is thus to make a copy of the original `pom.xml` and then revert the changes after the derived project has been built.

The `ContinuumProject` is updated from `pom.xml` after each update from VCS. This means that any changes that might have been made to `ContinuumProject` is lost. This is correct and meaningful for original projects, but this will make all derived projects identical to the original project. The solution chosen is to make a copy of all attributes that separate an original project from a derived project. After the project has been updated, these attributes should be reintroduced to ensure that none of our changes are lost.

Our proposed solution can be summarized as follows:

- Make Continuum aware of derived projects, see section 4.2.2
  - Create/update derived projects according to the rules in section 3.1.
  - Store derived projects to persistent storage.
- Build the derived projects with Maven, see section 4.2.3
  - Make a backup copy of `pom.xml`.
  - Update `pom.xml` with the derived dependencies according to the derived `ContinuumProject`.
  - Build the project.
  - Revert the changes. (I.e., overwrite the modified `pom.xml` with the backup copy)
  - Ensure that the `ContinuumProject` is not reverted to the state of the original `pom.xml`.

4.2 Proof of concept implementation

The proof of concept implementation described in this chapter explain *how* the solution chosen in section 4.1 has been implemented. The purpose is to show that the concept works, so production quality code is not a priority. This realisation affects not only quality attributes, but also which features are implemented. Anything not essential to demonstrate the concept is omitted.

We begin with an overview of files modified or added, before we describe how the functionality can be implemented.

4.2.1 Files added or modified

Table 4.1 show every file that differs from Continuum-1.0.3, omitting tests.

Change	**Module**	**File(s)**[1]
Create derived projects	continuum-core	`components.xml`, `DefaultContinuum`, `DefaultDependencyHelper`, `DefaultDerivedProjectManager`
	continuum-api	`DependencyHelper`, `DerivedProjectManager`
	continuum-model	`DependencyGroup`
	continuum-store	`JdoContinuumStore`
Build derived projects	continuum-core	`ExecuteBuilderContinuumAction`, `MavenBuilderHelper`, `DefaultMavenBuilderHelper`, `MavenTwoBuildExecutor`, `FileIO`
Update data models	continuum-model	`continuum.mdo`
Add *derived* properties to GUI	continuum-web	`Summary.vm`, `View.vm`
Upgrade JPOX	/ & continuum-core	`pom.xml`
	continuum-updater	`AntProject`, `ContinuumBuild`, `ContinuumDeveloper`, `ContinuumNotifier`, `ScmFile`, `ContinuumProject`, `ScmResult`, `MavenOneProject`, `UpdateScmResult` `MavenTwoProject`, `ShellProject`, `CheckOutScmResult`
Add Java 1.5 support	/ & continuum-store	`pom.xml`

Table 4.1: All changed files

[1](A ".java" file extension is assumed whenever file extension is omitted.)

The files are grouped according to the reason for editing them, and the grouping should be correspondent to a mapping between the conceptual solution and the implementation.

The most important files are listed below, before sections 4.2.2 and 4.2.3 explain how we have implemented the pseudo code from section 4.1.1. Changes that are of technical character, or just pure convenience, are explained section 4.2.4.

Important files

DefaultContinuum - this class is the heart of Continuum. All business logic is initiated from `DefaultContinuum`, directly or indirectly.

ContinuumProject - this class is responsible for all project data that Continuum needs. One `ContinuumProject` corresponds to one `MavenProject`. The `MavenProject` is used for the actual building process, while the `ContinuumProject` is used for administration and GUI.

BuildDefinition - this bean holds the details for when and how a project should be built.

DefaultDerivedProjectManager - a controller class which creates and modifies derived projects. `DefaultDerivedProjectManager` utilize `DefaultDependencyHelper` for processing of `ProjectDependency` objects.

ProjectDependency - this bean represents a dependency. It has variables like groupId, artifactId and version.

DefaultDependencyHelper - a helper class which handles the processing of `ProjectDependency` objects.

JdoContinuumStore - a JDO implementation. This class is responsible for persistence.

pom.xml - an XML representation of Maven's POM. This is the central configuration file for Maven projects.

MavenProject - this class holds all information from a `pom.xml` file. This information is the basis for the actual build.

DefaultBuildController - this class is responsible for actions that are initiated for each build.

MavenTwoBuildExecutor - this class controls the build on the file system. `FileIO` and `DefaultMavenBuilderHelper` are utilized as helper classes.

DefaultMavenBuilderHelper - a helper class that handles transformations to and from the `pom.xml` file.

FileIO - a utility class for handling file input and output.

4.2.2 Make Continuum aware of derived projects

When a user adds a Maven 2 project, `addMavenTwoProject()` is called on `DefaultContinuum`. This initiates creation of `ContinuumProjects` from the metadata found in the referenced `pom.xml`. Subsequently, the projects are saved to persistent storage by `JdoContinuumStore`. This sequence is shown in figure 4.1.

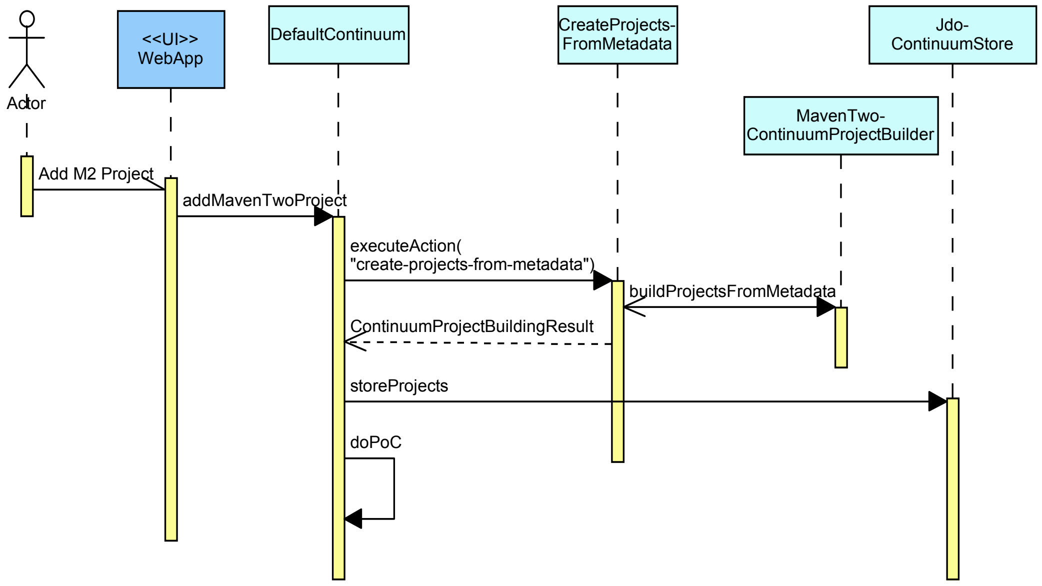

Actor
<<UI>> WebApp
DefaultContinuum
CreateProjects-FromMetadata
Jdo-ContinuumStore
MavenTwo-ContinuumProjectBuilder
Add M2 Project
addMavenTwoProject
executeAction("create-projects-from-metadata")
buildProjectsFromMetadata
ContinuumProjectBuildingResult
storeProjects
doPoC

Figure 4.1: User adds a Maven 2 project

After the original project has been persisted, the figure show that *doPoc()* is called. `doPoc()` is responsible for creating the appropriate derived projects and saving them. Currently it operates on *all* projects in store, but it will not add a new derived project if a derived project with the same groupId, artifactId and combination of dependencies already exists.

The list of derived dependencies are hardcoded[2] directly in the `doPoc`-method. This list is used as input to the `createDerivedProjects` method described next.

Create derived projects

The number of dependencies is always constant, but an original dependency can be substituted with a derived dependency. These combinations of old and new (derived) dependencies are the basis for adding new derived projects. Below, in listing 4.1, we explain how these combinations can be found using a recursive algorithm.

We start by splitting the dependencies into groups. Dependencies with the same groupId and artifactId are placed in the same group. A set obtained by selecting one dependency from each group is called a *selection*. Initially *currentGroup* is 0 and *selection* and

[2]Fetching these from an xml-file is added to the further work, see section 7.2.

uniqueCombinations are empty. Note that the implementation require that *groups* and *selection* are of equal size.

```
private void dependencyCombinations(int currentGroup,
        final List<DependencyGroup> groups,
        List<ProjectDependency> selection,
        List<List<ProjectDependency>> uniqueCombinations) {

    if (currentGroup == groups.size()) {
        uniqueCombinations.add(cloneDependencyList(selection));
    } else {
        ProjectDependency projectDependency;
        DependencyGroup group = groups.get(currentGroup);
        for (int i = 0; i < group.size(); i++) {
            projectDependency = group.get(i);
            selection.set(currentGroup, projectDependency);
            dependencyCombinations(currentGroup + 1, groups, selection,
                uniqueCombinations);
        }
    }
}
```

Listing 4.1: dependencyCombinations - find all combinations recursively

The algorithm chooses one dependency from the first group and then calls itself recursively with the index *currentGroup* incremented by one. The selection now holds one dependency. This procedure is repeated for all dependencies in the first group. In other words, a recursive call is spawned whenever a group holds more than one dependency. *uniqueCombinations* will thus hold one selection for each possible variant. The recursion ends when one dependency has been chosen from every group for all recursive calls.

This code is located in `DefaultDependencyHelper` and is called through a wrapper method named `getAllDependencyCombinations()`. This wrapper ensures that the recursive method is called with the correct parameters. It is also responsible for adding the unaffected dependencies to each combination found.

`getAllDependencyCombinations()` is called from the `createDerivedProjects()` method in `DefaultDerivedProjectManager`, as shown in figure 4.2.

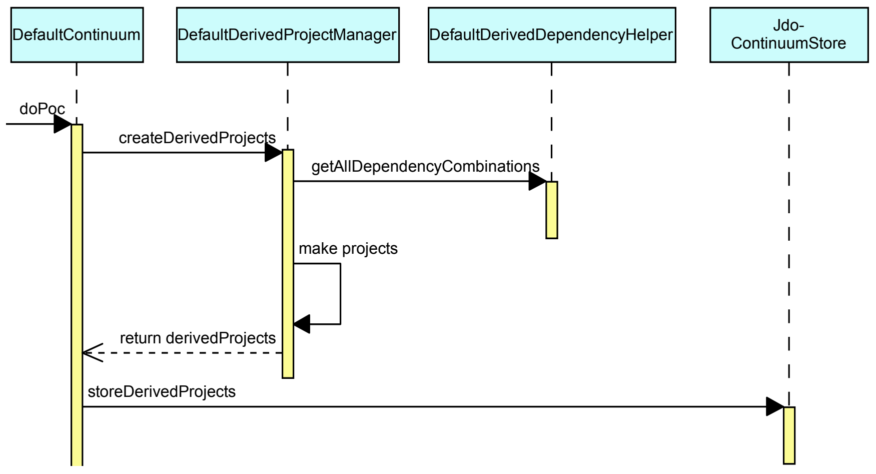


Figure 4.2: doPoC

In `createDerivedProjects`, one derived project is created for each combination, if the combination does not already exists. The only combinations that already exists come from the original projects, so no derived project is created with this combination of dependencies. The rest of the attributes of the new derived projects are cloned from the original project.

4.2.3 Build derived projects

A project can be built either when triggered by a schedule or when the user *force* a build. Forcing a build is done by manually selecting “Build all" or “Build now" in the GUI. Either way, `DefaultBuildController` is responsible for managing the builds.

First the VCS is checked for changes. If none are detected, the build is canceled. Otherwise, the `ContinuumProject` is updated from the checked out copy of the project, and the project is built. Figure 4.3 shows these two actions.

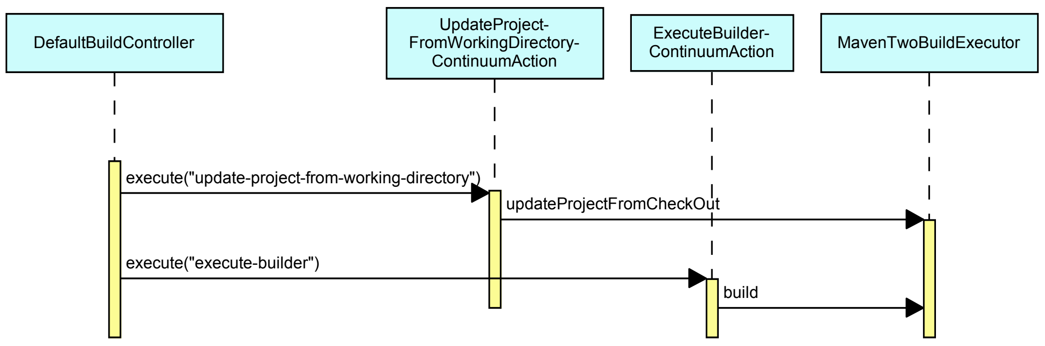

DefaultBuildController
UpdateProject-FromWorkingDirectory-ContinuumAction
ExecuteBuilder-ContinuumAction
MavenTwoBuildExecutor
execute("update-project-from-working-directory")
updateProjectFromCheckOut
execute("execute-builder")
build

Figure 4.3: build in DefaultBuildController

We check whether the project is derived in the `build()` method in `MavenTwoBuildExecutor`. If it is, `buildDerivedMavenProject()` is called instead of the simple `executeShellCommand()`. This fork is shown in listing 4.2.

```
public ContinuumBuildExecutionResult build(Project project,
  BuildDefinition buildDefinition, File buildOutput )
  throws ContinuumBuildExecutorException {
...
ContinuumBuildExecutionResult result = null;
if (project.isDerived()) {
    result = buildDerivedMavenProject( project, executable, arguments, buildOutput);
  } else {
    result = executeShellCommand( project, executable, arguments, buildOutput);
  }
  return result;
}
```

Listing 4.2: Intercepting normal flow

This is consistent with the requirement of not being overly intrusive. Only the build process for derived projects are affected, all original projects are built as before.

Build process for derived projects

The actual building of derived projects is handled by the `buildDerivedMavenProject()` method, while delete, copy and move is handled by the classFileIO class. FileIO is an utility class which use the `java.io`-package to perform file system operations. The structure of `buildDerivedMavenProject()` can thus be structured according to the strategy in section 4.1.1.

First we take a copy of `pom.xml`, then the dependency list is updated with the new versions, before the modified `MavenProject` is written to disk, overwriting `pom.xml`. After the project has been built with the Maven executable, we revert our changes to `pom.xml` and return to the normal execution with the result. See listing 4.3 for source code.

```
private ContinuumBuildExecutionResult buildDerivedMavenProject(
      Project project, String executable, String arguments, File buildOutput)
      throws IOException, MavenBuilderHelperException, ContinuumBuildExecutorException {

   ContinuumBuildExecutionResult result = null;
   File workingDirectory = getWorkingDirectory( project );
   String pomPath = workingDirectory + File.separator + "\class{pom.xml}";
   String pomBackupPath = workingDirectory + File.separator + "\class{pom.xml}.bak";

   FileIO.copy(pomPath, pomBackupPath);          //Backup

   MavenProject mavenProject = builderHelper.getMavenProject(new File(pomPath));

   List<Dependency> updatedDependencyList = builderHelper.createNewDependencyList(
      mavenProject.getDependencies(), project.getDependencies());

   mavenProject.setDependencies(updatedDependencyList);

   PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(pomPath)));
   mavenProject.writeModel(out);

   result = executeShellCommand( project, executable, arguments, buildOutput );

   FileIO.move(pomBackupPath, pomPath);          //Rollback
   return result;
}
```

Listing 4.3: Build derived projects

The last thing on the list is to ensure that our changes to the `ContinuumProject` are not overwritten. The relevant code is located in `DefaultMavenBuilderHelper`, where the method `mapMavenProjectToContinuumProject()` is responsible for updating the `ContinuumProject` with the changes from the `MavenProject`. Our modifications here ensure that the version of each project retain the appended counter and that the derived dependencies are not incorrectly replaced.

4.2.4 Other changes

In addition to the previously described changes, a few modifications were made to speed up development.

Added Java 1.5 support to speed up development. This was only convenience and should be reverted to Java 1.4 syntax before a backport is attempted.

Upgraded JPOX to version 1.1.6. This was done due to multiple bugs in the older JPOX version, a 1.1.0 snapshot. Upgrading to a current and proper release also made it easier to get support from the community.

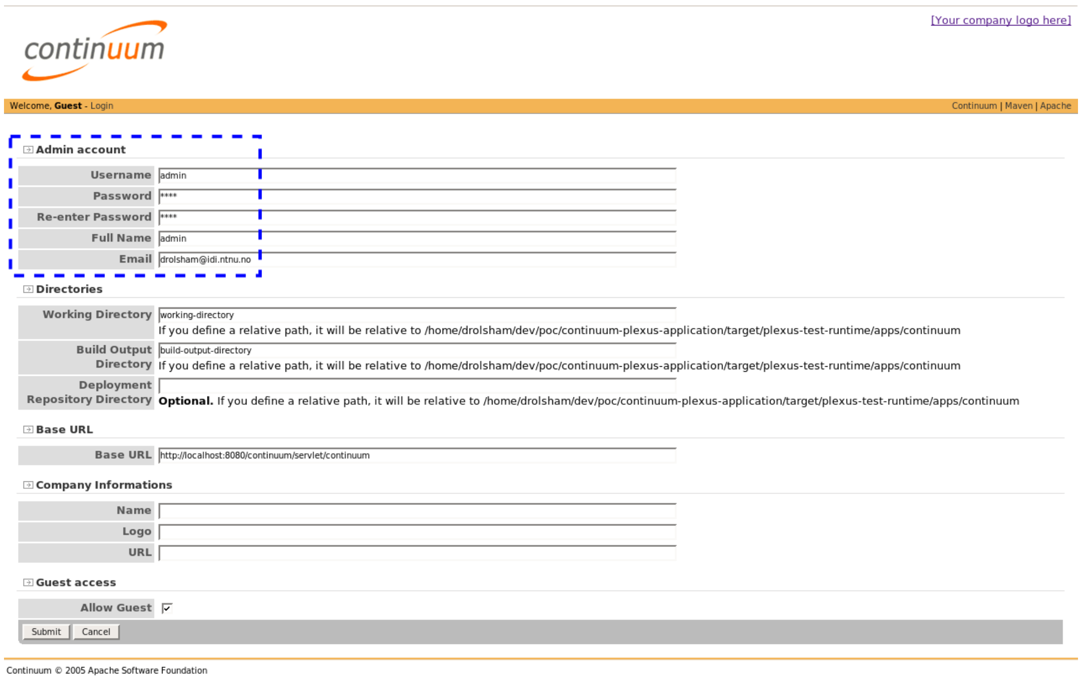
Added derived properties to the GUI to emphasize the differences between *original* and *derived* for dependencies and projects. The logical implementation is not affected by these changes. They merely make it easier to see what is happening.

4.3 Screenshots

We will now show a sequence of screen shots. They are taken after the application has been started[3] and it shows the steps initial setup, project checkout and the building of all projects. We have also included a couple of screen shots that show the differences between original projects and derived projects.

4.3.1 Initial setup and project checkout

After start-up an administration account must be set up.



Admin account

Username	admin
Password	****
Re-enter Password	****
Full Name	admin
Email	drolsham@idi.ntnu.no

Directories

Working Directory	working-directory If you define a relative path, it will be relative to /home/drolsham/dev/poc/continuum-plexus-application/target/plexus-test-runtime/apps/continuum
Build Output Directory	build-output-directory If you define a relative path, it will be relative to /home/drolsham/dev/poc/continuum-plexus-application/target/plexus-test-runtime/apps/continuum
Deployment Repository Directory	**Optional.** If you define a relative path, it will be relative to /home/drolsham/dev/poc/continuum-plexus-application/target/plexus-test-runtime/apps/continuum

Base URL

Base URL	http://localhost:8080/continuum/servlet/continuum

Company Informations

Name	
Logo	
URL	

Guest access

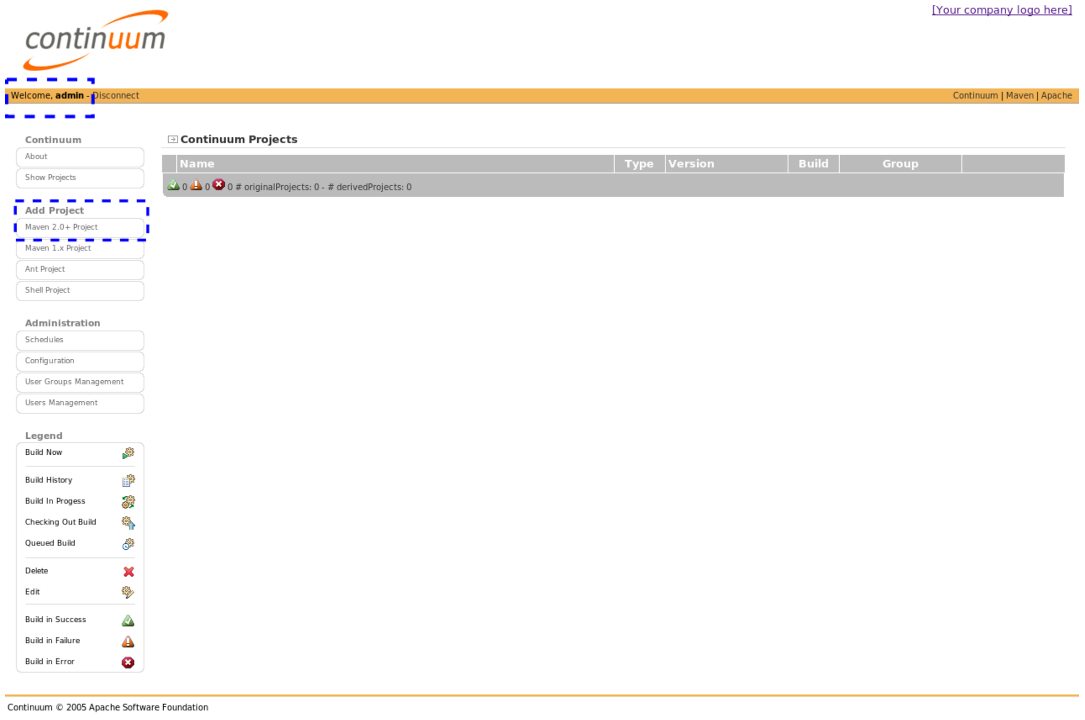
Allow Guest	✓

Continuum © 2005 Apache Software Foundation

Figure 4.4: Configure administration account

[3]See section 2.6.3 for a description on how to build Continuum from source and start up the server.

Log in with this account to obtain administration privileges. The next step is to add a new Maven2 project. Click **Add Project** → **Maven 2.0+ Project**, as emphasized in the lowermost rectangle.



Continuum Projects

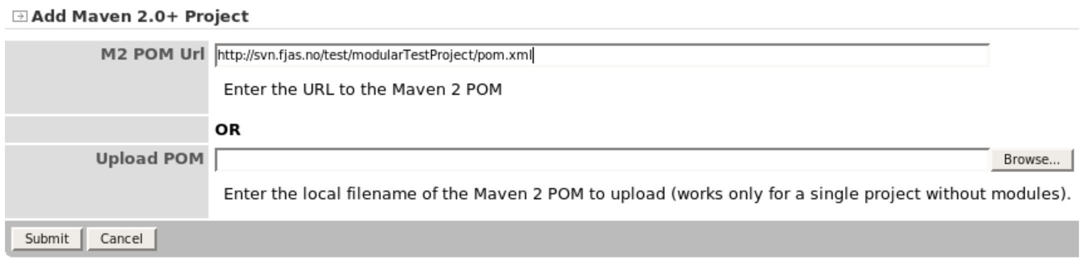
	Name	Type	Version	Build	Group	

0 0 0 # originalProjects: 0 - # derivedProjects: 0

Continuum © 2005 Apache Software Foundation

Figure 4.5: Logged in as Admin

A project can be added by submitting an URL to a `pom.xml` file or by uploading a `pom.xml` file directly. We must choose the first option, because the file upload option does not support multi-module projects. I.e., a project is added to Continuum by submitting an URL to a `pom.xml` file.



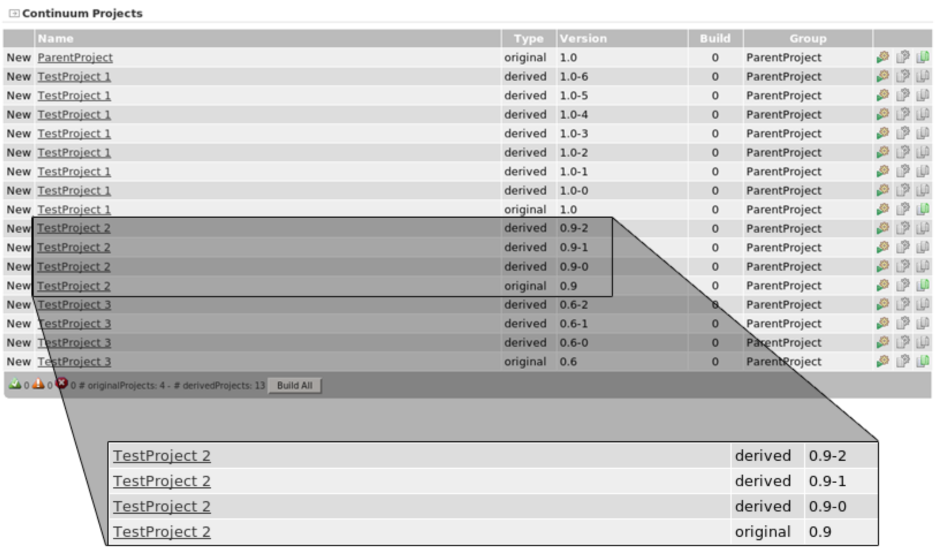
Add Maven 2.0+ Project

M2 POM Url	http://svn.fjas.no/test/modularTestProject/pom.xml Enter the URL to the Maven 2 POM
	OR
Upload POM	 Enter the local filename of the Maven 2 POM to upload (works only for a single project without modules).

Figure 4.6: Add Maven 2 project from Url

4.3.2 Build all projects

The unmodified version of Continuum would have created only the four original projects, while our version also adds derived projects. The original Test Project 2 and its three derived variants are emphasized in figure 4.7.



Continuum Projects

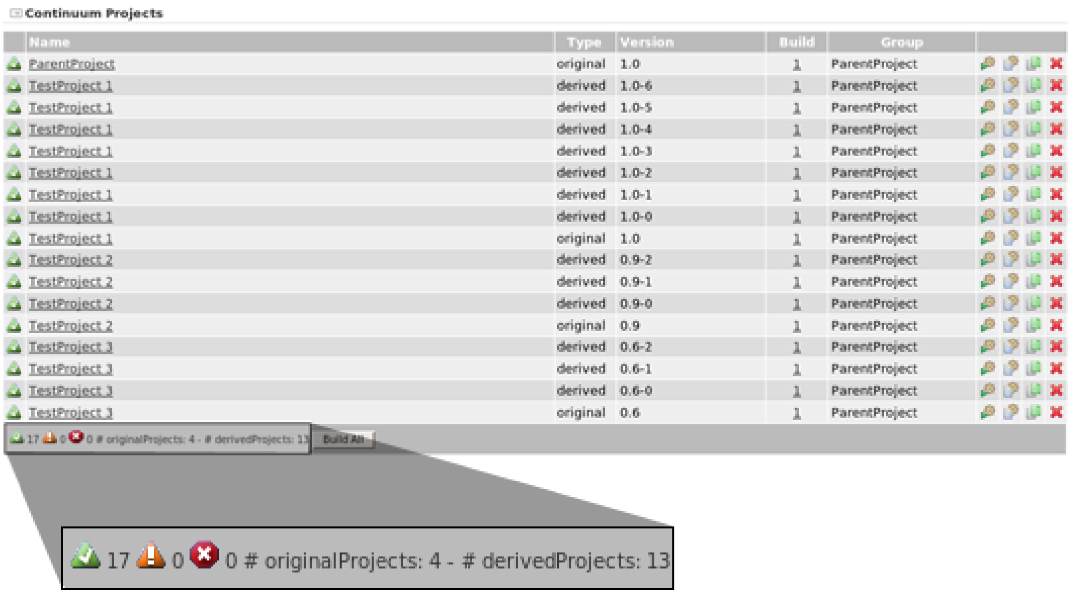
	Name	Type	Version	Build	Group
New	ParentProject	original	1.0	0	ParentProject
New	TestProject 1	derived	1.0-6	0	ParentProject
New	TestProject 1	derived	1.0-5	0	ParentProject
New	TestProject 1	derived	1.0-4	0	ParentProject
New	TestProject 1	derived	1.0-3	0	ParentProject
New	TestProject 1	derived	1.0-2	0	ParentProject
New	TestProject 1	derived	1.0-1	0	ParentProject
New	TestProject 1	derived	1.0-0	0	ParentProject
New	TestProject 1	original	1.0	0	ParentProject
New	TestProject 2	derived	0.9-2	0	ParentProject
New	TestProject 2	derived	0.9-1	0	ParentProject
New	TestProject 2	derived	0.9-0	0	ParentProject
New	TestProject 2	original	0.9	0	ParentProject
New	TestProject 3	derived	0.6-2	0	ParentProject
New	TestProject 3	derived	0.6-1	0	ParentProject
New	TestProject 3	derived	0.6-0	0	ParentProject
New	TestProject 3	original	0.6	0	ParentProject

0 0 0 # originalProjects: 4 - # derivedProjects: 13 Build All

TestProject 2	derived	0.9-2
TestProject 2	derived	0.9-1
TestProject 2	derived	0.9-0
TestProject 2	original	0.9

Figure 4.7: Projects created

On screen shot 4.8, the status of the last build for each project is shown to the left of the project name. The magnified section show that there are four original projects and thirteen derived projects, and all built without errors.



Continuum Projects

	Name	Type	Version	Build	Group
✓	ParentProject	original	1.0	1	ParentProject
✓	TestProject 1	derived	1.0-6	1	ParentProject
✓	TestProject 1	derived	1.0-5	1	ParentProject
✓	TestProject 1	derived	1.0-4	1	ParentProject
✓	TestProject 1	derived	1.0-3	1	ParentProject
✓	TestProject 1	derived	1.0-2	1	ParentProject
✓	TestProject 1	derived	1.0-1	1	ParentProject
✓	TestProject 1	derived	1.0-0	1	ParentProject
✓	TestProject 1	original	1.0	1	ParentProject
✓	TestProject 2	derived	0.9-2	1	ParentProject
✓	TestProject 2	derived	0.9-1	1	ParentProject
✓	TestProject 2	derived	0.9-0	1	ParentProject
✓	TestProject 2	original	0.9	1	ParentProject
✓	TestProject 3	derived	0.6-2	1	ParentProject
✓	TestProject 3	derived	0.6-1	1	ParentProject
✓	TestProject 3	derived	0.6-0	1	ParentProject
✓	TestProject 3	original	0.6	1	ParentProject

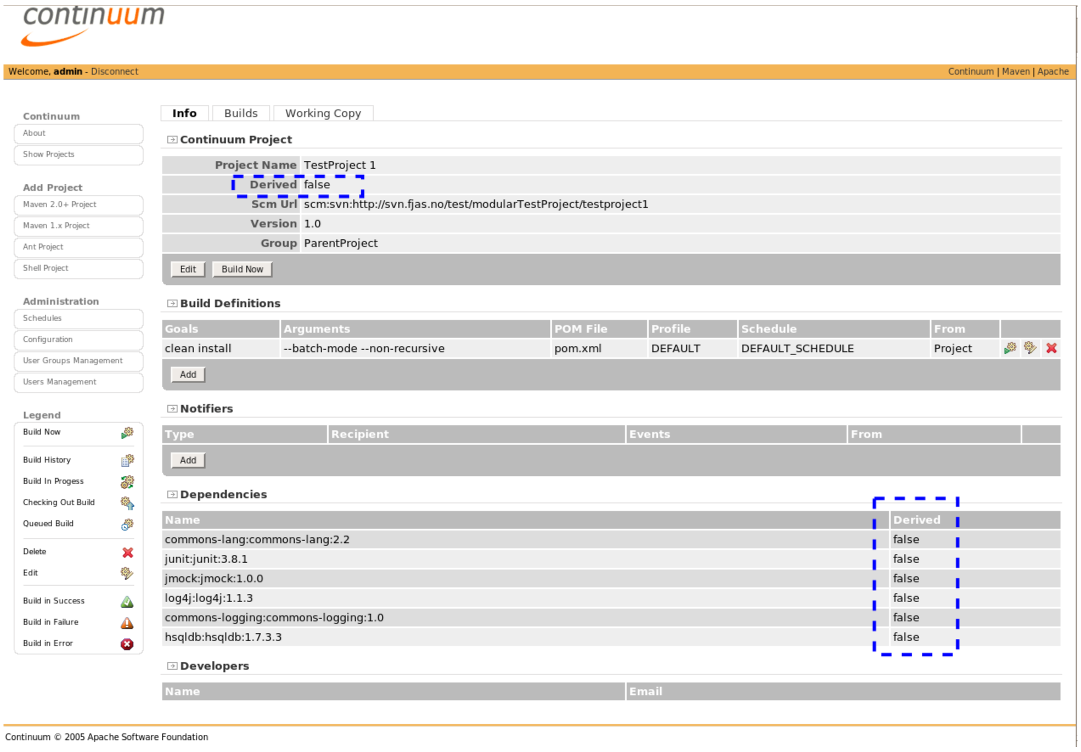
17 0 0 # originalProjects: 4 - # derivedProjects: 13 Build All

17 0 0 # originalProjects: 4 - # derivedProjects: 13

Figure 4.8: All projects was built successfully

4.3.3 Details on original and derived projects

The next screen shot shows the details of Test Project 1. We can see that it is a original project, since it is marked “Derived false” in the section for project details, and because no dependencies are derived.



Continuum Project

Project Name	TestProject 1
Derived	false
Scm Url	scm:svn:http://svn.fjas.no/test/modularTestProject/testproject1
Version	1.0
Group	ParentProject

Build Definitions

Goals	Arguments	POM File	Profile	Schedule	From	
clean install	--batch-mode --non-recursive	pom.xml	DEFAULT	DEFAULT_SCHEDULE	Project	

Notifiers

Type	Recipient	Events	From	

Dependencies

Name	Derived
commons-lang:commons-lang:2.2	false
junit:junit:3.8.1	false
jmock:jmock:1.0.0	false
log4j:log4j:1.1.3	false
commons-logging:commons-logging:1.0	false
hsqldb:hsqldb:1.7.3.3	false

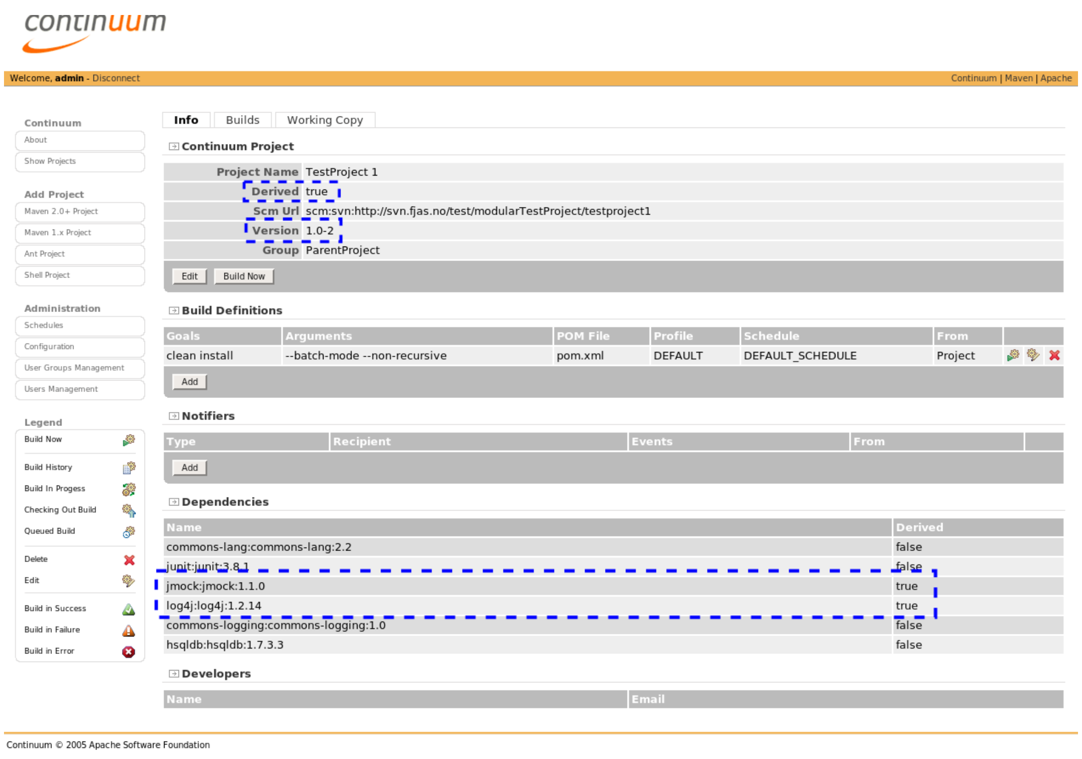
Developers

Name	Email

Continuum © 2005 Apache Software Foundation

Figure 4.9: Detailed view of the original Test Project 1

A derived project looks somewhat different. The project itself is marked “Derived true”, and the version is post-fixed by a counter. In addition, one or more of the dependencies will be marked “Derived true” and have a version different from the corresponding dependency in the original project.



Continuum Project

Project Name	TestProject 1
Derived	true
Scm Url	scm:svn:http://svn.fjas.no/test/modularTestProject/testproject1
Version	1.0-2
Group	ParentProject

Build Definitions

Goals	Arguments	POM File	Profile	Schedule	From	
clean install	--batch-mode --non-recursive	pom.xml	DEFAULT	DEFAULT_SCHEDULE	Project	

Notifiers

Type	Recipient	Events	From	

Dependencies

Name	Derived
commons-lang:commons-lang:2.2	false
junit:junit:3.8.1	false
jmock:jmock:1.1.0	true
log4j:log4j:1.2.14	true
commons-logging:commons-logging:1.0	false
hsqldb:hsqldb:1.7.3.3	false

Developers

Name	Email

Continuum © 2005 Apache Software Foundation

Figure 4.10: Detailed view of a derived version of Test Project 1

4.3.4 Conclusion

Figure 4.8 shows that there are one parent project, three original test projects and thirteen derived projects. TestProject 1 generated seven derived projects and TestProject 2 and 3 generated three each. This corresponds to example 3.2.2 from chapter 3. We have also shown one example of a derived project, figure 4.10, which had a different kind of dependencies than the original project in figure 4.9. In other words, these screen shots verify that the prototype is consistent with the conceptual solution.

Chapter 5

Evaluation

In this chapter we will evaluate the prototype and the preliminary research methods. The purpose of this evaluation is to look back on the process and discuss what went well and what could have been improved. We will also indicate whether alternative paths could have been more beneficial.

5.1 Preliminary research

This section will evaluate the research done in chapter 2. We will evaluate how we researched state of the art, the theory on continuous integration and how we studied Continuum.

5.1.1 State of the art

The value of a proof of concept implementation may be characterised by a number of factors, but its value drops quickly if it has been done before. It is thus important to do a thorough "state-of-the-art" study. It is not possible to prove that something never has been done before, but we can show that it is *unlikely*.

The first approach was to search "continuous integration" in digital libraries like The ACM Digital Library, IEEE Xplore and Google Scholar, but we found nothing relevant. A comparisons matrix maintained at codehaus.org was examined next. The matrix presents the features of multiple continuous integration servers, but our suggested improvement was not mentioned. The third approach was to ask around in the open source community. Still no indication that anything similar had been done before. An expert, Trygve Laugstøl at Objectware AS, was cited to support the wide range of informal statements we received. The fourth approach was to explore the documentation of four popular continuous integration servers. Again we found no indication that this concept had already been implemented. Finally we skimmed a number of books on the subject. The books did not contain anything relevant and were therefore not mentioned in chapter 2.

As far as we have been able to uncover, no solution to the problem described in chapter 1 exists, but our research cannot be used to state this as a fact. The sources are much to informal and incomplete for that. However, the research indicate that if such functionality exists, then it is not well-known. Writing a prototype to demonstrate the concept should thus be valuable.

5.1.2 Theory

General theory on continuous integration is relevant background information, since the conceptual solution and the prototype is based upon these concepts. We also discussed Maven POM and Maven's versioning scheme, since they have been used throughout the report.

The reader is assumed to have prior knowledge of continuous integration, versioning and Maven. The concepts described were included to allow the reader to brush up on his/her knowledge. Another objective was to ensure a common understanding of the terms used.

Our research indicate that there are few comprehensive sources on continuous integration available. The overview in section 2.2 is based on the most comprehensive source we found, an article by M. Fowler and M. Foemmel [8]. Since continuous integration is not a complex concept, and the reader is assumed to have some development experience, our summary should be sufficient as an introduction to our suggested new technique.

5.1.3 Get to known Continuum

It was essential for both planning and implementation to know how Continuum works. Some documentation was found in the Continuum distribution, but it turned out to be obsolete. We worked with the latest stable release, version 1.0.3. The only other alternative would have been trunk, the primary development source code. We tried this approach, but trunk was too unstable. It would not even build successfully when we tried it, nor could we find any useful documentation there either.

We tried to use the *maven-site-plugin* to extract information. We succeeded in running the plug-in and adding multiple reporting plug-ins, but the reports did not provide much new information. E.g., when the source code is lacking Javadoc, the Javadoc report do not contribute much.

Information was acquired by studying the source code and asking questions in the community. Trygve Laugstøl, the user and developer email lists and the Internet Relay Chat (IRC) channel #continuum @ irc.codehaus.org were the primary sources.

The greatest advantage of reading source code is that it is 100% accurate. The code *is* the software. A disadvantage is that reading source code is a time-consuming task. Speaking to developers and users that have experience with the software can explain *what the software can do, why something implemented a certain way, how the different components interact, known bugs, etc*. This information may be inaccurate, and it may be hard to get answers from the most clever people. This method is valuable nonetheless, because humans can clear up misunderstandings and inspire new, more precise questions in a way no form of written documentation can do.

It was hard to get an overview of Continuum by reading source code and asking questions. High-level documentation, had any existed, would have been better suited for a general introduction to the architecture and important design decisions. Direct questions were better suited to research implementation details. Especially with little or none Javadoc available, the opportunity to ask questions about the source code was invaluable. While studying the source code and asking questions were tedious methods, they worked.

What we learned

Architecture and the technology employed was researched first. During this process we learned that Continuum had a component-oriented structure. The next logical step was therefore to identify the components, and their most central classes, relevant to the prototype . We considered documenting how these central classes interacted, but decided against it. The expected gain was not expected to outweigh the required effort. We did however document a couple of relevant sequences to explain the prototype, see section 4.2.

While studying the most relevant components, we had to learn the basics behind the technology employed. Again, the available documentation was less than satisfactory, but - again - the community was helpful. Communication through email and IRC turned out to be most effective.

Summary

Working with Continuum was frustrating due to the lack of documentation. We explored all options known to us, but none seemed more enticing than studying the source code and asking questions in the community. Reading documentation and books on the subject was never an option, because none were found. Was it not so, we would have expected to get an overview of Continuum far easier and quicker than figuring it out on our own. On the other hand, documentation and books might get out of synchronisation with the code. Detailed descriptions of the implementation is especially hard to maintain, so the source code is often the only reliable reference. Given that we were to implement a prototype, studying the source code was inevitable.

In retrospect, we do not see that we could have approached the problem in any other way, given that we did not find any resources that we did not use or at least evaluate.

5.2 Prototyping

This section describes the prototype and the strategy for the implementation. The focus is on design decisions, and if we would have chosen differently knowing what we know now.

5.2.1 Strategy

The strategy explains the most important choices we made before and during the work on the prototype. This was not a poor attempt at a design. The purpose was simply to describe the choices we made and explain why we chose as we did.

Problem text and underlying build engine

Maven and Continuum is mentioned in the problem definition, so already at this stage we had an idea that Continuum and its tight coupling to Maven would be an integral part of the solution. Even though we examined other continuous integration servers, Continuum was considered part of the baseline, and we never found any reason to consider replacing it with another continuous integration server.

RPC, component reuse or extension?

As mentioned in section 4.1, we found three viable approaches to utilize Continuum; use Continuum's XML-RPC API, build a new application with components from Continuum, or modify the existing code base directly. We chose the last option, because we wanted tight integration with Continuum. We still believe this was a good idea and have found additional arguments in favour of this choice:

Easily accessible test data: Our extension is useless without test data in the form of real projects. Having one product, instead of two, ensures that all available projects can be used as test data.

Fine-grained reuse: Since we modified Continuum directly we could reuse code on a fine-grained level, not being limited to whole components as in alternative two. This reuse saved a lot of effort and made the implementation lean and easy to understand. One product is also easier to maintain than two.

The tight coupling to Continuum is an evident trade-off point. Benefits have been mentioned, so we will now address two drawbacks:

Intrusiveness was listed as a potential problem in section 4.1. Since we were aware of this from the beginning, we were able to take measures to mitigate the problem. The tactics we used is discussed in section 5.2.2. The tactics worked, and we consider the prototype to be minimally intrusive. This disadvantage can thus be neglected.

Limitations inferred by the existing implementation is another drawback. I.e., frameworks and libraries have been chosen, Java version and coding style has been set, the architecture is locked etc. We could ignore some, but not all, of these limitations, since the implementation was only a prototype. Dealing with JPox was the biggest problem we encountered. In retrospect, we have identified two main causes:

1. The JPox version used in Continuum 1.0.3 is full of bugs. However, after upgrading to a newer version, most of the bugs disappeared.

2. It is necessary to define which fields of a class that should be fetched. This definition is called a *fetch group*[1]. By default only primitives and object wrappers of primitives are part of the fetch group, e.g., String, boolean, Boolean, long, Long, etc.

   This puts extra work on the developer, because he/she must specify which fields to fetch. The prototype required additional fields not fetched in the original Continuum implementation. We were thus forced to reload some fields or change the existing configuration. If all fields were lazily loaded by default, this had not been a problem. Instead a lot of time was wasted, because it was difficult to identify what changes were required and where they should be applied. In the end, this caused reduced readability, since modifications were not co-located.

In spite of the impediment JPox turned out to be, the benefits of reusing the data access logic outweighed the estimated effort of writing the data access logic from scratch. The problems we encountered were thus unfortunate, but not something we could have avoided all together.

The Continuum way or the hard way?

Given Continuum's architecture, it was reasonable to create a new project for each combination of dependencies. This is due to the fact that Continuum can build any project that has been successfully stored. This approach made it possible to reuse existing functionality and kept our modifications from spreading all over.

Another possibility would have been to store only the original project and build this project with all possible combinations of dependencies. No extra projects are generated with this approach, so no extra checkouts from the Version Control System (VCS) are needed either. Updates should also be easier, since there is only one project that holds all the combinations. The biggest disadvantage is that this solution requires modifications several places in the source code. This makes it intrusive and hard to maintain. This property also makes it harder to ensure correctness, because it is difficult to identify where changes must be made.

We consider the first option more suitable than the second, because it makes it easier to follow good object-oriented practices, as described in [1].

[1]Fetch groups are explained in [31].

5.2.2 Implementation

For an actual product it would be appropriate to evaluate functional and non-functional requirements. However, the purpose of this proof of concept implementation, is to demonstrate that the theoretical concept is possible to implement. We will therefore focus on what is adequately implemented and what is not. The prioritised list of improvements identified can be found in section 7.2.

Non-functional evaluation

Quality attributes are in general not given much weight in this prototype, only *reusability* was given a high priority. Reusability was considered important, because the proof of concept implementation was based on the latest stable version of Continuum, which is quite old. In other words, it is likely that the prototype must be ported to a newer version of Continuum. The tactics chosen were to enforce strict separation of concerns and to make as few modifications to the existing code base as possible. The same tactics also reduced intrusiveness. The functionality that create derived projects illustrates this:

The flow of execution is intercepted in *DefaultContinuum* and `doPoc()` is executed. This method holds a list of derived dependencies and utilize functionality in *DefaultDerived-ProjectManager* and *DefaultDependencyHelper*. Locating the correct interception point in version 1.1 and calling `doPoc()` should then be sufficient to port the prototype to the new version. As shown in table 4.1, some additional setup must be made, but these are technicalities related to Modello and Plexus.

Functional evaluation

According to the rules in section 3.1 there are six changes that should result in an action. The first two, add and remove projects, are supported. This is enough to demonstrate the concept. The four other rules describe updates to existing projects and were omitted due to time constraints. An update to either the input list, or the dependency list in the original project, requires that the set of derived projects must be recalculated. I.e., derived projects must be added or removed to reflect the changes in dependencies. This is important functionality, but not critical for demonstrating the concept.

We also made two other design decisions related to completeness.

- **The input list** was put directly in the code. This was done to reduce the total programming effort. This list should be refactored and put in a configuration file. The format for *dependency* described in Maven's project descriptor [32] seems appropriate.
- ***Any* version change** results in new derived projects. The abstract solution states that only *upgrading* a dependency should be possible, but we decided to create new derived projects whenever the version was different from the original. We believe that this is appropriate, since the purpose of the prototype is to illustrate the concept and stimulate new ideas. In an actual implementation, however, it seems natural to

let a derived dependency with the same or older version be ignored, with an option to enable support for any version change.

Both are simple tasks, but not necessary to demonstrate the concept and were omitted.

Summary

The strategy was to extend Continuum and modify the existing code base directly. This choice was made after carefully considering three alternatives, and in retrospect it still seems to be a reasonable approach. We focused on reusing functionality in Continuum and making our extension easy to port to a new version.

The prototype support addition and removal of projects, which should be sufficient to demonstrate the concept. Support for updating dependencies was not implemented, due to time constraints.

Chapter 6

Discussion

We will here explain our claimed contribution to the field of System Engineering. This includes not only chapter 3, but also the results, i.e., the prototype presented in chapter 4. The purpose of this section is thus to put the work we have done into context and discuss its usefulness.

Before we explain theoretical advances, we will discuss utility value. This ordering is advantageous, since it is easier to discuss usefulness when the context of practical applications is fresh in one's mind.

6.1 Utility value

This section will explain why our proposed extension to Continuum is useful.

6.1.1 Help developers create backward compatible and stable software

The primary functionality is to add a new, unreleased version of a software project as a derived dependency to Continuum. The developers of this software can then utilize continuous integration to test whether their new version is compatible with projects that depend on an old version of the service. This will help developers to follow their chosen versioning scheme and help keep services backward compatible. Second, the build output from Maven is available from Continuum, making it easy to determine its origin and in which phase the build error occurred. This feature should be useful for developers working on a new version of an existing service, since the developers can trust that the projects they have in their instance of Continuum that utilize this service, will be compatible with the new version.

A history of good backward compatibility and an orderly release history indicate a *stable* service. This should make it more enticing to utilize the service. An indirect benefit is thus improved chances of *reuse*, since clients of the service can be confident that new versions are backward compatible.

Another indirect benefit is better collaboration with the development teams responsible for the projects that utilize their service. The rationale is that with automation there is no doubt as to whether the project builds successfully with the new version of the service or not. Continuous integration ensures that the status of the build is always known. A failed build entails that the new version of the service is not backward compatible with the old version of the service and that the development team responsible for the service is to blame. However, a successful build does not necessarily guarantee backward compatibility. If a project turns out to be incompatible with the new version of the service, despite a successful build, then the problem is that the tests in the utilizing project are inadequate. The development team responsible for the utilizing project is then in error, the developers responsible for the service cannot be blamed. Clarifying responsibilities should have a positive effect on collaboration between the development teams.

It is necessary to have projects that depend on the service in question to take advantage of our proposed extension. This is not considered a problem, since a service-oriented architeture does not make sense unless there are consumers interested in the services offered. A typical scenario is that multiple projects depend on some common core functionality. Best practice, according to object-oriented principles, is then to write the functionality in such a way that it can easily be reused. The obvious choice is a separate component with a well-defined API. This scenario is illustrated by the following example.

Example

The Apache MyFaces [1] project use Continuum for continuous integration. In terms of projects, it is not a huge organisation, but it is big enough that multiple projects depend on a core service. The summary page [2] of MyFaces' Continuum server show that their thirteen project groups result in a total of 118 modules/sub-projects. The three projects Trinidad, Tobago and Tomahawk all depend on the service *MyFaces Core - JSF 1.1*. While not all of their 74 modules depend on Core, it is reasonable to assume that the total API coverage is good, since the Core module holds functionality common to the other projects.

Using our proposed extension the development team responsible for Core can build all projects that depend on version 1.1 with the new 1.2-snapshot. This ensures that they will get feedback whenever the 1.2-snapshot no longer is compatible with version 1.1 and will make life much easier for developers on all four projects.

The Core service should also be much more enticing to other projects, because they can check the project summary page for MyFaces and see the status of all the builds. Developers evaluating Core can be confident that all functionality utilized by Trinidad, Tobago or Tomahawk will work in version 1.2, if it worked in version 1.1. While no API coverage is guaranteed, this is much better than having no guarantees at all.

6.1.2 Maven version ranges

Versioning ranges, explained in [4, chap. 3.6], is a way of specifying that multiple versions of a dependency is acceptable for a project. This can be used to resolve version conflicts in big projects. E.g., two dependencies use the Springframework as a transitive dependency. One require $[1.1, 1.3]$[3], while the other accepts $[1.2,)$[4]. Any version in the intersection between the two ranges, from 1.2.x to 1.3.y, can be used to solve the version conflict.

A prerequisite for exploiting version ranges is that releases can be ordered and sorted into ranges. However, this is not enough. To define an acceptable range, the developer must know which versions that are compatible. Testing all releases as they become available is not an option, so the developer must trust that releases adhere to the versioning scheme employed. Maven version ranges will only be useful when this trust is in place.

According to these premises, we can deduct that our extension can make version ranges more useful, since it will have beneficial effects on the backward compatibility and stability of services and help developers adhere to their chosen versioning scheme.

[1] `http://myfaces.apache.org` Last visited: 2007.05.21

[2] `http://myfaces.zones.apache.org:8081/continuum` Last visited: 2007.05.21.

[3] versions from 1.1. to 1.3, inclusive

[4] version 1.2 or newer

6.1.3 Tool support for complex dependency upgrades

The proposed solution makes it possible to use Continuum to investigate which combinations of dependencies that build successfully, which might be useful when upgrading multiple dependencies. In the rare case that multiple dependencies are incompatible with each other, it might save time to use our extension to generate projects with the different combinations. The purpose is then not to use continuous integration to detect incompatibility, but to discover which combinations that build successfully.

However, in most cases, experimenting with different versions could just as well be done manually. This kind of usage is considered a curiosity, because we are not familiar with any practical use cases. We have nevertheless chosen to describe it, since users often come up with scenarios that the developers never thought of.

6.1.4 Inspire innovation

The metadata needed for building a Maven project can be found in `pom.xml`[5]. This encourage convention over configuration, since defaults can be loaded from `pom.xml`. This is an important driver for Continuum's architecture and facilitates simpler configuration than traditional approaches like, e.g., CruiseControl.

Our extension combine the metadata from Maven with test data (the projects) from Continuum and utilize the concept of continuous integration. I.e., we take existing concepts and technology and combine them in a new way. This might inspire further innovation.

[5]See section 2.4 for an introduction to the Maven's Project Object Model (POM) and `pom.xml`.

6.2 Theoretical advances

Continuous integration is an old concept that has become increasingly more popular with the growing popularity of Test-Driven Development (TDD) and agile development methodologies. While the new work patterns demand more frequent builds than before, the emphasis is still on building a single service in isolation. We will here discuss a new continuous integration technique.

6.2.1 Prerequisite

Metadata is needed to realise the new continuous integration technique. We need to know the dependencies of a project, and we need a continuous integration server that allows us to modify this set of dependencies and build the project with these modified dependencies. Maven's Project Object Model (POM) holds the information we require, and Continuum satisfies the requirements for the build server. While there are other ways to implement this concept than using Continuum and Maven, the idea was conceived in this context. Our research is thus based on the advances made by these two Open Source Software (OSS) projects.

6.2.2 A new technique

Standard continuous integration builds one service and its dependencies. We propose to turn this around and build all projects that *depend* on a service. If a project builds successfully with the previous version of the service, but not with the new version, then a compatibility issue has been identified. Successful builds means that the developers can be confident that all available clients[6] still work with the new version of the service. When testing more than one service, the combinations of new and old versions of services are also interesting. This means that for projects that depend on more than one service the different combinations must also be built.

We suggest using a continuous integration server for running builds whenever a change is detected by the Version Control System (VCS), to obtain the benefits of continuous integration as described in section 2.2.1.

6.2.3 Contribution and benefits

This is a new angle to continuous integration, which adds a new level of testing. We exploit projects already built by Continuum as test data and use them to test the API of a service. The focus is not on checking whether method A and method B has been implemented, or if they take the correct parameters, but on checking that the functionality utilized by other projects still work. I.e, we concentrate on testing the functionality that is actually *used*. Pure API compliance can be tested by other tools, see [33, 34].

[6]A client is a project that depend on the service.

With this technique we achieve better control over changes to the service and can thus improve backward compatibility. Additionally, better control over changes should mean that no changes are inadvertently introduced, which should result in more *stable* services. Another benefit is that the general quality will increase as the software is put through not only *more* testing, but also a *different kind* of testing. The relevancy of this testing is high, since users of an API might find ways to utilize the API that the developers did not think of when writing and testing it.

Stable services with a tidy release history and good backward compatibility indicate a sound development process. This instills confidence in the service and encourages reuse. Tool support, as we have described, can thus improve backward compatibility, stability and service quality and can make a service more interesting to reuse.

6.2.4 Test coverage

This technique exploits projects that depend on the service as test data. The effectiveness of this technique is thus determined by the number of projects that use the service and how diverse their usage is. When testing *one* new version of service, one additional project must be built for each project that depend on this service. If another service is added for testing, there are four combinations that must be built. The complexity follows formula 3.1, and as explained in section 3.1.4, the number of combinations grows quickly.

To avoid congestion it is important that the continuous integration server is able to finish building projects before the next build is initiated. I.e., there must be balance between the number of projects to build, the time interval between builds and available hardware resources. Assuming a fixed hardware budget and a fixed time interval, this gives that it might not be possible to build all possible combinations. Test coverage is thus not only limited by the number of available projects that utilize a service, but also by how many projects the continuous integration server can support with the chosen time interval. This indicate that the user may have to prioritise which projects to build. Suggestions for future work that handle this potential problem can be found in section 7.2.1.

6.2.5 Real-life example

We will now present a use case from the NAUT [7] project where Erik Drolshammer is developer. This example show why backward compatibility is important and demonstrate the problem that we claim to reduce.

The Acegi Security System [8] depends on the Spring framework. This dependency is documented in the `pom.xml` files published at Maven's central repository [9] . The different versions of Spring that Acegi depends on is shown below.

[7] `http://naut.abakus.no` Last visited: 2007.05.19
[8] `http://www.acegisecurity.org` Last visited: 2007.05.19
[9] `http://repo1.maven.org/maven2` Last visited: 2007.05.19

Acegi version		**Spring version**
1.0.0	→	2.0-m2
1.0.1/1.0.2	→	1.2.7
1.0.3	→	1.2.8

Acegi use a versioning scheme based on *major.minor.patch*. The patch version is defined as: *"To retain perfect source and binary compatibility, a patch release can only change function implementations. Changes to the API, to the signatures of public functions, or to the interpretation of function parameters is not allowed. Effectively, these releases are pure bug fix releases."* [35]. Developers should thus expect that releases that differ only in patch versions are completely backward compatible.

However, this is not necessarily true. Projects that already have a dependency on Spring, in addition to the transitive dependency from Acegi, may experience version conflicts when trying to upgrade Acegi. This is due to the fact that two versions of the same dependency cannot coexist[10].

There are no version conflicts in a project that depends on Spring 2.0-m2 and Acegi 1.0.0, but upgrading to Acegi 1.0.1 will force Maven to make a choice between 1.2.7 and 2.0.x. If 2.0.x is chosen, and Spring is backward compatible, as they claim to be[11], then there is no problem. However, if 1.2.7 is chosen, and the project utilize functionality found only in 2.0.x, trouble is inevitable.

Poor backward compatibility does not instill confidence. The kind of behaviour described above may thus make developers more sceptical to Acegi, affecting their willingness to utilize the service. Using our extension, with relevant test projects, the Acegi developers could have detected and fixed this problem prior to release. The problem of backward compatibility can thus be reduced to *"How to ensure that we have enough relevant test projects?"*. Suggestions for how to deal with this new problem can be found in section 7.2.2.

We do not doubt that the Acegi team has lots of tests to check backward compatibility, but it is unreasonable to assume that they can cover all possibilities. This example shows that using actual projects as test data can unravel problems that other tests missed.

[10]See [4, section 3.6] for details on how Maven resolves version conflicts.

[11]Rod Johnson claims backward compatibility in [36].

Chapter 7

Conclusion and further work

This chapter contains the conclusion and a list of suggestions for future work.

7.1 Conclusion

We have described a new, advanced continuous integration technique. The concept is to automatically build projects that utilize a service to test a new version of this service. If not all projects build successfully with the new version, then a compatibility issue has been identified. A continuous integration server ensures that the projects are built whenever the service is changed and alerts the developers when a build error occurs.

The purpose of the new technique is to help developers ensure that their services are backward compatible, not to test simple API compliance. The new technique is based on the assumptions that more thorough testing is necessary to ensure good backward compatibility, and that it is most important to thoroughly test functionality that is actually *used*. I.e., if we test the new version of the service with all available projects that utilize the old version of the service, then we know, for all practical purposes, that the new version is backward compatible.

Developers that utilize a service will exploit the service in any way possible to achieve their business goals. This means that the service is tested with combinations of parameters and environments that the regular functional tests might not cover. This is a beneficial secondary effect that will improve the overall quality and stability of the service.

A prototype based on the continuous integration server Continuum has been implemented as proof of concept. We have specified six rules that govern how the prototype should react to different user actions. Two rules, add new project and remove project, are essential to demonstrate the concept and are correctly implemented in the prototype. The four other rules depict updates to existing projects. These are not essential to explain the concept and are not supported by the prototype. However, it should be a straightforward matter to add support later, since the prototype was structured to support this extension. See section 7.2.1 for further details.

The purpose of the prototype was to create a proof of concept. Quality attributes have therefore not been evaluated. However, continuous integration is resource intensive, and performance might be an issue in an actual implementation. Worst case complexity calculations show that the number of additional projects to build grows quickly. Tactics to improve performance should thus be evaluated when refining the prototype further. Ideas that came up during prototyping can be found in section 7.2.1.

We have shown that continuous integration can be used to test backward compatibility of a service. The new technique should also improve quality and increase stability of the service, since the service is put through additional functional tests. The prototype demonstrates how the concept works in practice and that an implementation is feasible with technology available today.

7.2 Future work

This section describes possible steps for future development on the prototype and several topics suitable for future research. Section 7.2.1 describes possible extensions identified during prototyping. Section 7.2.2 contains suggestions on how to further extend the prototype by combining elements from test coverage tools and tools for API compliance testing. Section 7.2.3 contains a list of questions that can be used as basis for empirical studies.

7.2.1 Prototype

The purpose of the prototype was to demonstrate a concept. In this section we have identified steps necessary to transform the proof of concept implementation into fully functional software. We suggest implementing the features in the order they are described.

Port the prototype to trunk

Continuum 1.1 Alpha 1 was released April 23. 2007, a year after the release of Continuum 1.0.3. A report from their issue tracking system Jira [1] is the only documentation currently available that lists the changes. This report [37] shows that there are 343 closed issues. In other words, 1.1-alpha1 has undergone massive changes compared to version 1.0.3. Since our prototype is based on version 1.0.3, the natural first step is to port it to trunk[2] or at least to the continuum-1.1-alpha-1 tag.

The prototype utilize Java 1.5 syntax, and both version 1.1-alpha and and trunk currently use Java 1.4. However, as of time of writing there is an ongoing vote on the developer's email list [3] as to whether trunk will be upgraded to Java 1.5. It is thus likely that it is unnecessary to convert the prototype back to Java 1.4 syntax.

Technical improvements

The focus of the prototype was proof of concept. The code quality is thus not comparable to production code and measures must be taken to improve the quality. More extensive tests, at least unit tests and integration tests, should be written first. This is necessary to facilitate refactoring and detect bugs. Code cleanup and bug fixing are implied in this process.

The next step is to examine JPOX calls. Several suboptimal solutions were chosen in the prototype and can be optimized. The most serious were due to technical difficulties related to wiring in the continuum-store component. The work-arounds introduced are acceptable for the prototype, but not for a finished product and should be fixed.

[1] `http://www.atlassian.com/software/jira` Last visited: 2007.05.26

[2] The term trunk denotes the main development branch in a VCS.

[3] `http://mail-archives.apache.org/mod_mbox/maven-continuum-dev/200706.mbox/date` Last visited: 2007.06.05

Support for updates

The set of derived projects must be updated whenever the input list or the dependencies of the original project are changed. Otherwise the projects would have to be deleted and re-added to Continuum whenever any of these changes occurred.

We anticipated the need for updates, so the *updateDerivedProjects()* method in *Default-DerivedProjectManager* already support adding new derived projects according to the list of dependencies that is sent as parameter. *updateDerivedProjects()* should also remove derived projects that are no longer relevant. This method can then be used to update the project list whenever the input list is changed. The steps needed are thus;

- Extend *updateDerivedProjects()* to remove derived projects that no longer are relevant.
- Call *updateDerivedProjects()* every time the input list is changed.

Improve usability

Usability is not critical for a prototype, but it is important for the success of an actual product. We suggest the following improvements to make it easier to cope with derived projects.

- The input list should be located in a separate configuration file. The format for *dependency* described in Maven's project descriptor [32] seems appropriate. An empty file or a missing file should disable our suggested extension all together. It would also be convenient to have an option to disable all or some of the dependencies in the configuration file without actually removing the entry.
- It should be possible to choose whether Continuum should use all dependencies added to the input list as basis for the calculation of combinations or just the dependencies with newer versions than the corresponding original dependency. This setting can be placed in the same configuration file as mentioned above.
- Tactics to improve navigation and make the relationship between original project and its derived variants more explicit:
  - Use a tree structure on the project summary page, where each original project is a top-level node, with its derived projects are children.
  - Add a link to the original project in each of the derived projects and a list of links to the derived projects to the original project.
  - Add a link to the original dependency in the derived dependencies.

Improve performance

Bulding and integration are performance intensive activities. This indicate that it can be beneficial to evaluate tactics that make the implementation more efficient. An effective tactic is to reduce the number of derived projects that is actually added or built. We suggest to make it possible for the user to

- disable our suggested extension for certain projects.
- exclude or remove combinations of dependencies, to support that some combinations of dependencies may be irrelevant in the user's context.

Support for these scenarios allows the user to choose which projects to build and how extensive and how resource intensive the building will be. Available resources can thus be spent where there is most to gain.

7.2.2 Combine with other tools?

Our proposed new continuous integration technique is related to API compliance testing and test coverage. These relations seem to warrant further studies;

- Can existing test coverage tools like Clover [4] , Cobertura [5] or Emma [6] be used to obtain metrics on utilization coverage for our proposed extension?
- Supplementing our implementation with API compliance testing seems to be an appropriate extension. Can our proposed feature be combined with existing tools for testing API compliance?

7.2.3 Empirical studies

If the prototype is incorporated into an actual product, it is important to investigate if the expected benefits are realised and to what degree. The following questions should be elucidated:

- Have projects that utilize this new continuous integration technique better backward compatibility than other similar projects?
- Has collaboration between development teams improved?
- Has the new technique facilitated more frequent releases?
- Has the reputation of the service changed and how has this affected reuse?

[4] `http://maven.apache.org/plugins/maven-clover-plugin` Last visited: 2007.05.22
[5] `http://maven-plugins.sourceforge.net/maven-cobertura-pluginn` Last visited: 2007.05.22
[6] `http://emma.sourceforge.net/maven-emma-plugin` Last visited: 2007.05.22

Bibliography

[1] Robert C. Martin. *Agile Software Development*.
Prentice Hall, 2003.

[2] Venkat Subramaniam and Andy Hunt. *Practices of an Agile Developer: Working in the Real World (Pragmatic Programmers)*. Pragmatic Bookshelf, April 2006.

[3] Jeff Langr. *Agile Java, Crafting Code with Test-Driven Development*.
Prentice Hall, 2004.

[4] Vincent Massol and Jason van Zyl. *Better Builds with Maven*.
Mergere Library Press, October 2006.

[5] Wikipedia. Software versioning.
http://en.wikipedia.org/wiki/Version, January 2007.

[6] Denis Howe. *Free Online Dictionary of Computing*. http://www.foldoc.org/, May 2007.

[7] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[8] Martin Fowler. Continuous integration. *martinfowler.com*, May 2006.

[9] Steve Mcconnell. Daily build and smoke test.
IEEE Software, 13(4):144, 1996.
http://stevemcconnell.com/ieeesoftware/bp04.htm.

[10] Wikipedia. Chrysler Comprehensive Compensation System.
http://en.wikipedia.org/wiki/Chrysler_Comprehensive_Compensation_System, January 2007.

[11] Cruisecontrol - Project Details.
http://sourceforge.net/projects/cruisecontrol, January 2007.

[12] Anthill OS - A Bit of History.
http://anthillpro.com/html/products/anthillos, January 2007.

[13] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.

[14] James Newkirk and Robert C. Martin. *Extreme Programming in Practice*.
Adison-Wesley, 2001.

[15] Tijs van der Storm. Continuous release and upgrade of component-based software. In *SCM '05: Proceedings of the 12th international workshop on Software configuration management*, pages 43–57, New York, NY, USA, 2005. ACM Press.

[16] Continuous Integration Server Feature Matrix. `http://damagecontrol.codehaus.org/Continuous+Integration+Server+Feature+Matrix`, January 2007.

[17] Paul Duvall. Automation for the people: Choosing a Continuous Integration server. `http://www-128.ibm.com/developerworks/java/library/j-ap09056/index.html#N101CD`, September 2006.

[18] Cruisecontrol - Default Plugin Registry. `http://cruisecontrol.sourceforge.net/main/plugins.html#defaultregistry`, January 2007.

[19] Cruisecontrol - 3rd Party CC Stuff. `http://confluence.public.thoughtworks.org/display/CC/3rdPartyCCStuff`, January 2007.

[20] Continuous Integration Features. `http://anthillpro.com/html/products/anthillpro/features/continuous-integration.html`, January 2007.

[21] TeamCity Features' Comparison Matrix. `http://www.jetbrains.com/teamcity/documentation/featureMatrix.html`, January 2007.

[22] Jason van Zyl and Eirik Bjørsnøs. Maven: The Definitive Guide. `http://www.sonatype.com/book`, May 2007.

[23] Henning Jensen and Erik Drolshammer.
Best Practice within Java Web Application Development, 2006.

[24] Introduction to the Dependency Mechanism. `http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html`, June 2007.

[25] The XStream project. About Versioning. `http://xstream.codehaus.org/versioning.html`, January 2007.

[26] The Apache Software Foundation. Apache License, January 2004. `http://www.apache.org/licenses/LICENSE-2.0.txt`.

[27] Palash Ghosh. Java Component Development: A Conceptual Framework. *ONJava*, March 2005.

[28] George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[29] application.xml from GBuild. `http://cwiki.apache.org/gbuild/applicationxml.html`, February 2007.

[30] Plexus Component Descriptor.
http://plexus.codehaus.org/guides/developer-guide/configuration/component-descriptor.html, February 2007.

[31] JPOX. Java Persistent Objects - Fetch-Groups.
http://www.jpox.org/docs/1_1/fetchgroup.html, May 2007.

[32] The Apache Maven Project. Maven project descriptor, May 2007.
http://maven.apache.org/ref/current/maven-model/maven.html.

[33] The Java Compatibility Test Tools.
http://java.sun.com/developer/technicalArticles/JCPtools/, November 2006.

[34] Stuart Ballard. Japitools - Java API compatibility testing tools.
http://sab39.netreach.com/japi/, November 2006.

[35] APR's Version Numbering.
http://apr.apache.org/versioning.html, May 2007.

[36] Rod Johnson. Spring 2.0: What's New and Why it Matters.
http://www.infoq.com/articles/spring-2-intro, 2007.

[37] The Codehaus. Release Notes - Continuum - Version 1.1-alpha-1, May 2007.
http://jira.codehaus.org/secure/ReleaseNote.jspa?projectId=10540&version=12082&styleName=Html.

Appendices

Appendix A

Acronyms

API Application Programming Interface

CI Continuous Integration

CBSD Component-based software development

DAG Directed Acyclic Graph

DBMS Database Management System

GUI Graphical User Interface

IDI Department of Computer and Information Science

IoC Inversion of Control

IRC Internet Relay Chat

IT Information Technology

JDO Java Data Objects

JPA Java Persistence API

JPOX Java Persistent Objects

JSP JavaServer Pages

NTNU Norwegian University of Science and Technology

OCP Open-Closed Principle

ORM Object-Relational Mapping

OSS Open Source Software

POM Project Object Model

QA Quality Assurance

SOA Service-Oriented Architecture

TDD	Test-Driven Development
VCS	Version Control System
SCM	Source Code Management
XML	Extensible Mark-up Language
XP	eXtreme Programming

Appendix B

Web pages

Continuum development environment

Subversion console client
http://subversion.tigris.org/project_packages.html
CVS console client
http://www.nongnu.org/cvs/#downloading
Java Development Kit
http://java.sun.com/javase/downloads
Maven2
http://maven.apache.org/download.html
Maven1
http://maven.apache.org/maven-1.x/start/download.html
Ant
http://ant.apache.org/bindownload.cgi

Continuum technology

Plexus
http://plexus.codehaus.org
JDO
http://java.sun.com/products/jdo
JPOX
http://www.jpox.org
Modello
http://modello.codehaus.org
Apache Velocity Project
http://velocity.apache.org