Bachelor Thesis

August 22, 2019

Investigating Plugin Usage in Open Source Maven Projects

Marc Zwimpfer

of Kilchberg, Switzerland (16-713-885)

supervised by

Prof. Dr. Harald C. Gall

Dr.-Ing. Sebastian Proksch

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

Investigating Plugin Usage in Open Source Maven Projects

Marc Zwimpfer

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Marc Zwimpfer, marc.zwimpfer@uzh.ch

Project period: 13.03.2019 - 22.08.2019

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

First of all, I would like to thank Dr.-Ing. Sebastian Proksch for his support throughout my bachelor's thesis. I especially thank him for the regular meetings during my thesis in which he guided me whenever I faced difficulties and pointed out new ideas to further improve this thesis.

I would like to thank Prof. Dr. Harald Gall for allowing me to write my thesis and use the facilities at the Software Evolution and Architecture Lab at the University of Zurich.

Lastly, I thank my family and friends who supported, motivated and helped me writing this thesis and throughout my bachelor's studies.

Abstract

Continuous Integration (CI) has become a widely-used software-engineering practice, in which the automated software build plays a central role. Software build systems, like Maven, execute this task by automatically generating executable software from source code and hence, play a crucial role in the whole CI process. Studies have shown that the configuration of such systems grows with increasing age and size of the underlying project. However, little research was conducted on the actual content of their configurations.

In this thesis, we examine in-depth how Maven plugins are configured in practice by analysing configurations of Open-Source projects using Maven. In Maven, all functionalities are provided by plugins, and thus they form the core of a every project. Analysing how plugins are used in Maven is important to gain further understanding of the usage of Maven and build systems in general.

Despite the importance of plugins, we find that plugin management only makes up a small portion of the complete Maven configuration. However, we show that plugins and their configurations are strongly influenced by inheritance in Maven projects.

With this thesis, we provide further insight into how developers actually use build systems. We show that the standard configuration of Maven regarding plugins suffices in most cases and thus, the concept of Maven - "Convention over Configuration" - is also successfully realized in the plugin configuration of Maven. Moreover, we propose a method which encodes Maven configurations into vectors which can be used for various analysis without information loss.

Zusammenfassung

Continuous Integration (CI) wurde zu einem wichtigen Prozess in the Software-Entwicklung, wobei der automatische "Build" der Software aus dem Quellcode eine zentrale Aufgabe einnimmt. Build-Management-Werkzeuge, wie zum Beispiel Maven, übernehmen diese Aufgabe und sind daher eng verknüpft mit CI. Einige Untersuchungen zeigten bereits, dass die Konfiguration solcher Werkzeuge zusammen mit einem grösser- oder älterwerdenden Projekt wächst. Trotzdem wurden bisher der tatsächliche Inhalt solcher Konfigurationen noch nicht ausführlich untersucht.

In dieser Arbeit analysieren wir wie Maven Plugins eingesetzt und konfiguriert werden indem wir die Konfigurationen von Open-Source Projekten, welche Maven benutzen, untersuchen. In Maven spielen Plugins eine wichtige Rolle, da sie für alle Aufgaben während einem Build-Prozess zuständig sind. Obwohl Plugins eine solche zentrale Rolle einnehmen, nimmt die Plugin-Konfiguration nur einen kleinen Platz in der gesamten Maven Konfiguration ein. Zudem wird die Art des Gebrauchs von Plugins und deren Konfigurationen stark von der Vererbung in Maven Projekten beeinflusst.

Mit dieser Arbeit geben wir neue Einblicke in die Art, wie Software-Entwickler im echten Leben Maven brauchen. Wir zeigen, dass die Standard-Konfiguration von Maven meistens ausreichend ist. Zudem entwickeln wir eine Methode, welche es erlaubt, die Konfigurationen von Maven ohne Informationsverlust in Vektoren umzuwandeln, welche dann für viele verschiedene Analysen gebraucht werden können.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Software build tools are used for automatically generating executable software from source code. They perform a series of steps which are necessary for this transformation which range from the compilation of the source code over the execution of the tests up to the deployment of the deliverables. By taking all these steps of the shoulders of developers, they provide immense value to their users. For most major programming languages, a variety of different build tools exists [13]. Hence, it is almost impossible to develop software projects without coming into contact with such tools. Three famous build automation tools for *Java* are Apache Ant (ANT), Apache Maven (Maven) and Gradle [4, 15, 16].

In the last years, many software developers started using Continuous Integrations (CI) for their projects [5]. CI describes a method of software development in which changes are directly integrated into the main software in contrast to traditional processes where changes were only included periodically. The CI process is made easy for developers by CI systems which automate the needed procedures. Examples for such CI systems are *Travis CI*[1] or *Jenkins*[2]. The main aspect of these systems is the automatic execution of the project build which can be triggered by various events. Therefore, these CI tools are closely intervened with build automation tools.

In order that build automation tools work properly, they need to be configured correctly for each individual project. The continuous maintenance and configuration of such systems can result in a considerable overhead of up to 12% of development effort [8]. Furthermore, the complexity of build system even increases with an increasing source code size [1, 10]. Therefore, the configuration of these systems should continuously be checked and adapted to changes by developers.

Different studies have analysed how the configuration influences or is influenced by surrounding parameters of a project [2, 8, 10]. However, little research was conducted on how build automation systems are actually configured.

Maven is one of the most popular build automation systems for Java. Maven follows the principle of "Convention over Configuration" and offers a standardized build lifecycle [16]. Each Maven project contains a configuration file (POM) in which all action of the systems are defined. The POM forms the core of every Maven project. Configuring a POM correctly is crucial for a successful build of the executable software. Despite the importance of this file, no further research was conducted on the contents of such a POM.

In the scope of this thesis, we analyse the configuration of Maven by examining Open-Source Maven projects. In a POM, the goals of the build lifecycle are declared in the form of plugins, which execute them. Hence, plugins form the core of all actions of Maven and the configuration of plugins directly influences the results of Maven builds. Due to their importance, we decided

[1] `https://travis-ci.org/`
[2] `https://jenkins.io/`

to study way how plugins are used in more detail and provide first insights into their application in projects in practice.

In order to investigate the usage of Maven plugins, we first need to design a method to extract the needed information from POMs. The method should provide the configurations contained in a POM in such a way that it is possible to analyse them with standard statistical approaches. This leads to the first research question of this thesis:

> **RQ1** How can configurations in POMs be extracted to further analyse them?
>
> We propose a method which extracts the configurations contained in POMs. Using this method results in a one-dimensional vector which contains all values set in the POM. No information contained in the original POM is lost during this action.

Using the results from this method, we examine how Maven plugins are used and configured inside POM in Open-Source projects. By doing this, we address the following research question:

> **RQ2** How are Maven plugins used in practice in Open-Source projects?
>
> We found that the declaration of plugins only makes up a small part of the total Maven project configuration. Furthermore, plugins are only configured rarely; it seems as if the standard configuration provided by Maven suffices for many projects.

We answer these questions in two steps. First, we analyse the role plugins play in the configuration of Maven and how plugins are configured themselves. Afterwards, we examine how plugins are used in the context of aggregation and inheritance. These are core features of Maven enabling the implementation of multi-module projects. Since developers behind Maven themselves encourage using these practices and we later discover that they are part of many projects, we analyse how plugins are used and configured in projects using aggregation and inheritance [16].

Investigating how plugins are utilized in Maven is important to gain further understanding of the usage of Maven and build tools in general. By analyzing how a big number of Open-Source projects are configured, we provide insights into the current practices used by developers. The usage of the outcomes of this thesis is manifold. Build tool developers gain further insights into the actual usage of build tools by their users. Developers using these tools is given an overview of how Maven projects are configured in practice. We contribute new insights into the usage of build tools which can be useful for further research in this area. Finally, we provide a method to extract information from POMs or XML-files in general.

The remainder of this thesis is structured as follows: First of all, we study related work on the area of the topic of this thesis. Then, we summarize important aspects of Maven which are needed in order to explain how we developed a method to extract information contained in POM files. We need this method for the next part of the thesis in which we conduct data analysis on the configuration of Maven plugins structured in two parts: first, we examine the usage of plugins in POMs and afterwards, how inheritance and aggregation in Maven projects affect plugin configuration. Finally, we discuss our findings by stating their implications, future research ideas and also summarizing possible thread to the validity of the analysis. The developed Java tool, all other artifacts of this thesis as well as an appendix can be found online [21].

Chapter 2

Related Work

We present other research related to the topic of this thesis in this chapter. It is roughly divided into two sections according to the content of this thesis: research related to build systems, their failures, and their configuration and research on approaches to extract information from XML files.

In recent years, CI has become a major trend in software engineering [5]. Many studies have analysed why CI is failing, respectively why the automated builds result in failures. Tufano et al. [17] analysed the reasons for build failures of snapshots in Java projects of *The Apache Software Foundation*, which were using Maven as build tool. Over a certain timespan, they checked how many commits in the history of projects build successfully. They found that the majority of build failures (up to 58%) resulted from dependency errors. Similar results were found by Sulir et al. [14], which also identified dependency issues as one of the main reasons for build failures; in Maven they account for 39.11%. Furthermore, they analysed in-depth the reasons for build failures in Maven projects. They found that, additionally to the dependency related failures, the Maven compiler- and javadoc-plugin also are responsible for almost 27% of the broken builds. Because dependency-related build failures happen very frequently, Macho et al. [9] propose a technique to automatically fix broken builds introduced by dependency issues which is possible in 54% of the tested build failures.

Vassallo et al. [18] created a taxonomy for reasons of build failures in Maven projects. On the contrary to many other studies, they also included software repositories from companies rather than only relying on Open-Source-Software, which showed that company-owned projects do not have the same distribution of failures than Open-Source projects. Concerning this thesis, it especially interesting that their build failure catalogue is originally based on the Maven build lifecycle.

Besides research on builds and their failures, also several studies specifically about build system exist. McIntosh et al. [10] analysed whether a growing source code also implies an increase of the complexity of Java build systems. Firstly, they found that the project configuration of a build systems (ANT and Maven) grows over time. Additionally, they showed that the complexity of the build system correlates with the size of the source code. Regarding our thesis, another interesting finding is that the Maven build depth stays constant over the lifetime of a Maven project. A similar study was performed by Adams et al. [1] in which they found that the Linux build system evolves too. In contrary to McIntosh et al. [10], their findings are based on "make", which is a more low-level build system than Maven. Hence, the correlation between source code and configuration growth can be found in all types of build systems.

Another field of research focuses on the effort spent on maintenance of build systems. McIntosh et al. [11] conducted a large-scale empirical study to analyse the relationship between build

technology and maintenance spent on these systems. They found that modern framework-driven build technologies tend to need more maintenance than low-level build tools. Moreover, they discovered that Maven needs the most maintenance of the compared build systems in this paper and that high-level build tools (as Maven) tend to be coupled more tightly with the source code.

By surveying software developers, which are using build systems in their projects, another study [8] showed that a substantial part of the developing time is spent on build tool maintenance. On average their participants stated that they spent 12% of their working time on build system maintenance with some cases even ranging up to 30%. However, these values are based on the perceived time stated by the questioned developers and not direct measurements of activity. Désarmeaux et al. [2] tried to specify further where maintenance effort is needed in build system by analysing how this effort is dispersed over the lifecycle phases of Maven. They have found that overall most maintenance effort is spent on the compile-phase in the Maven-lifecycle. However, the maintenance effort tends to shift between the different lifecycle phases of Maven.

As a part of this thesis deals with the extraction of information of XML-files and their translation into a one-dimensional vector, we present other research on this topic. Information retrieval and encoding the results in a vector space are topics of many studies. The major difference to classic information retrieval is that XML-documents are semi-structured, meaning that the structure may also have an impact on the contained information [20]. A study on different information retrieval methods on XML-documents has shown that using a document's structure in addition to only using its content performs better [20]. Kakade et al. [6] proposes an approach for representing XML documents in a vector space which accounts for the property that XML is semi-structured. Their methods allow comparing XML documents as well as performing XML queries on these. However, they also find that there are differences in the importance of the structure in XML-documents which affects the performance of their method.

Chapter 3

Configuration Extraction from Maven Projects

In this chapter, we introduce Apache Maven (Maven) by explaining its core functionality to create a shared understanding of the problem domain for our encoding method. We especially focus on the way Maven builds runnable Java projects and its configuration. Furthermore, we provide a method to encode Maven configuration files (POM) into vectors in order to analyse these configurations more easily. Lastly, we introduce a tool which can execute this encoding method on POMs and prepare the resulting data for further analyses.

3.1 Apache Maven

Maven, meaning *accumulator of knowledge* in Yiddish, is an open-source build automation tool for Java projects. The first version of Maven was released in 2004 and it is currently one of the most widely used build automation tools for Java projects. Maven is licensed under the *Apache License 2.0* and developed by both public contributors and the Apache Software Foundation. Since 2014, the only actively supported Maven version is *Maven 3* [16].

Maven has the goal of simplifying the build procedure of Java projects by providing a uniform build procedure, autonomous handling of project dependencies, project deployment and documentation publishing. The main advantage of Maven is hereby that it follows the principle of *convention over configuration* by having standardized project layouts and build mechanisms. This eases the start for new Maven users, saves time when changing between different projects and allows to adopt best practices without much effort. A trade-off of the uniform build mechanism is that it may not be suitable for all projects [16].

3.1.1 Lifecycle and Goals

Lifecycle

Maven uses clearly defined lifecycles to build its projects. Three main uniform lifecycles exist in Maven:

default handles the build and deployment of the project

clean handles the cleaning of the project by removing artifacts and files generated by previous builds

site handles the deployment of the project's documentation to a specified site

Each of these lifecycles consists of different phases which are executed sequentially to achieve its final result. As an example, we take a closer look at the phases of the default lifecycle. It consists of eight predefined main phases:

1. *validate*: validates whether the project is correct and all needed information is present
2. *compile*: compiles the source code of the project
3. *test*: tests the source code with provided unit tests
4. *package*: packages the compiled code into a distributable format (e.g. JAR or WAR)
5. *verify*: runs tests on created packages
6. *install*: installs the packaged projects locally for usage in other local projects
7. *deploy*: copies the final package to a remote repository

The default build lifecycle consists of further phases; however, these produce results which are only of value for the previous or following phase and not for the final build. The above-stated phases can be invoked via command line: for example with the command *mvn install* all phases up-to and including *install* are executed. In multi-module projects, Maven executes the phases for each module sequentially [16].

Goals

Each phase further consists of goals, which are the actual processes that are executed during a lifecycle. A certain phase can be made up of zero or more goals, where phases with zero goals are skipped during the build process. Maven goals are executed in the order that they are specified by the default configuration or in the POM file. Two ways exist on how to specify the goals which should be executed during a phase - via the selected packaging or plugin configurations in the POM [16].

Goals determined by packaging All packaging options (JAR, WAR, EAR, POM) for Maven projects bring some default goal binding with them. As an example, the JAR-packaging binds at least one goal to every phase to build a deployable JAR-project whereas POM-packaging only binds one goal to the install and deploy phase each.

Goals determined by plugins The second possibility to bind goals to phases is via manual configuration of the plugins in the POM. Plugins are components of Maven that provide certain goals. Goals defined in this way will be added to the list of already defined goals of the according phases. An important remark is that the same goal of a plugin can be bound to multiple phases of a project. Often plugin goals are bound to a default phase of the build process in which case it is not necessary to specifically define the phase.

3.2 Maven Configuration - The POM File

All actions of Maven are based upon the configurations in the Project Object Model (POM) contained in the project. It is an XML file, which contains in a declarative manner all information Maven needs in order to successfully build projects. The POM is the core of every Maven project.

The main parts which a POM may contain, are the project dependencies, the build configuration or the project metadata [16].

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>sample-project</artifactId>
  <groupId>org.sample.group</groupId>
  <version>1.0</version>
</project>
```

Listing 3.1: Basic pom.xml of a Maven project

Listing 3.1 shows the most basic version of a POM. It only consists of the Maven coordinates, which are the minimal information a POM must contain. Maven coordinates are the *artifactId*, *groupId* and *version*. This is required by Maven in order to uniquely identify the project amongst others. Furthermore, the <*packaging*/> tag also must be specified, but if no packaging is declared, Maven assumes a JAR project [16].

Normally POMs do not stay in their most basic version, but rather contain other sections which contain the actual configuration of the project. The three main sections of a POM concerning the actual build or site generation are:

build contains configuration for the build - e.g. used plugins and directories

dependencies contains the dependency management of the project

reporting contains configurations which are used during the *site* phase

There are many other tags which can be set in a POM, but these do not have a direct influence on the project build itself but are rather metadata of the project (contributors, organizations or licenses for example).

Additionally, a POM may contain multiple profiles which themselves can contain the sections mentioned above. These profiles provide the possibility to declare different configurations for different build procedures.

3.2.1 POM Relationships

An essential aspect of Maven is that it handles the relations between different POMs automatically. Besides the dependencies, two other kinds of relations exist: inheritance and aggregation. For both types, the parent or the aggregator POM are required to be of *POM* packaging type. It is important to differentiate between inheritance and aggregation. However, the same POM acts often as a parent and as an aggregator in one. In inheritance, the parent POM does not know its children, but the children know their parent whereas in aggregation, the aggregator knows its submodules but the submodules do not know the aggregator [16].

Inheritance

Similar to Object-Oriented programming, Maven supports inheritance between POMs. In order to do so, the child POMs must contain a section <*parent*/>, which contains the Maven coordinates

of the parent POM. All POMs originally inherit implicitly from the *Super POM*, which contains the default configurations. An example of a parent definition can be observed in Listing 3.2, which is an extract from a POM which inherits from the project *parent:parentGroup:1*.

Inheritance allows to define the configuration in multi-module projects once and let the sub-modules inherit the configuration; hence, duplication is reduced. If not declared otherwise, most of the available configurations are passed to the children POMs automatically. However, the child is allowed to override or extend certain values of the configuration.

```
<parent>
  <artifactId>parent</artifactId>
  <groupId>parentGroup</groupId>
  <version>1</version>
</parent>
```

Listing 3.2: Example of a parent declaration in a POM

Aggregation

An aggregator or multi-module project is a Maven project that consists of different subprojects. When the aggregator is built, it triggers automatically the build of its submodules. The order in which the submodules are built is determined automatically by Maven which checks for inter-module dependencies [16]. Listing 3.3 displays a sample declaration in an aggregator which contains two submodules. The <*module/*> tags contain the relative paths to the POMs of the sub-modules.

```
<modules>
  <module>submodule-1</module>
  <module>submodule-2</module>
</modules>
```

Listing 3.3: Example of a module declaration in a POM

3.2.2 Maven Plugins

As mentioned in section 3.1.1, the actions of a Maven lifecycle are determined by the declared goals which themselves correspond to goals of Maven plugins. Plugins provide the core functionality of Maven or as the Apache Software Foundation itself describes it: "Maven is - at its heart - a plugin execution framework; all work is done by plugins" [16]. Each plugin can have one or more goals which are attachable to lifecycle phases of Maven. Maven differentiates between two plugin types: build and reporting plugins. However, these types are not exclusive; a plugin may be a build as well as a reporting plugin (e.g. the *maven-javadoc-plugin*).

Build plugins are executed during the build of the project and should be configured in the <*build/*> section in the POM

Reporting plugins are executed during the site generation of the project and should be configured in the <*reporting/*> section in the POM

Usage of PluginManagement Moreover, build plugins can be declared either directly following the *<build/>* tag or be part of a *<pluginManagement/>* section. When using inheritance, the parent POM may contain such a *<pluginManagement/>* section, in which plugins are configured. The difference to directly declaring the plugins in the *<build/>* section is that those plugins are only declared in this POM, but are not part of the build. This allows the central configuration of plugins for all children. Additionally, the plugins are only part of the children's' builds if the configured plugin is declared in the child POM. On the other hand, plugins declared in the *<build/>* section of the parent are automatically part of the build in child projects.

Plugin Configurations

In many cases, the default configuration of build plugins does not suffice. In this case, developers may need to adjust the plugin to fit the project's needs. A *<configuration/>* sections containing general configurations for a plugin can be added to both build and reporting plugins. These configurations map directly to fields or setters of the plugin's implementation. Listing 3.4 shows an example configuration of the *maven-compiler-plugin* in which two configurations have been set to specify the Java versions for the compiler. If they were to be omitted, the Java 1.6 compiler would be used by default [16].

```
1  <plugin xmlns="http://maven.apache.org/POM/4.0.0">
2    <artifactId>maven-compiler-plugin</artifactId>
3    <groupId>org.apache.maven.plugins</groupId>
4    <configuration>
5      <source>1.8</source>
6      <target>1.8</target>
7    </configuration>
8    <executions>
9     <execution>...</execution>
10   </executions>
11   <dependencies>
12    <dependency>...</dependency>
13   </dependencies>
14   <inherited>false</inherited>
15 </plugin>
```

Listing 3.4: Example configuration of *maven-compiler-plugin*

An additional configuration, both build and reporting, plugins accept, is the *inherited* option. By default plugins and their configurations are inherited by the children of a project. Setting the value of *<inherited/>* to *false* will stop the inheritance.

Configuration of build plugins Additionally to the generic configuration, build plugins allow two further configuration tags: *<executions/> and <dependencies/>*.

executions if plugins should be executed in multiple phases of the build lifecycle, one may use this section to configure each execution separately. A single execution must at least contain a phase to bind to and a goal to execute. Executions will be inherited to children.

dependencies if plugins depend on other Maven projects, they should be declared in this section.

Configuration of reporting plugins Reporting plugins can likewise be further adjusted within its *<reportSets/>* section.

reportSets In this section, the configuration, about which reports should be generated during the site generation of a Maven project, is located.

3.3 POM Encoding Into Vector Space

In this section, we propose a technique to encode POMs into a one-dimensional vector, which can be used for further analyses.

3.3.1 Goal of the Method

The Maven POM file contains all the configurations Maven needs in order to successfully build a project. POMs are written in XML. XML is a markup language which is widely used to encode hierarchical information in a human- and machine-readable format [19]. The XML standard is managed by the World Wide Web Consortium (W3C)[1].

Even though XML is standardized, analyzing XML documents is not straight forward. As XML documents are built in a hierarchical manner, the depth of a document can grow indefinitely. As a result, we decided to develop a method which transforms the tree-like structure of XML document into a one-dimensional vector. This vector allows the usage of common analysis techniques and nonetheless is still human-readable. Additionally, the encoded vector can be converted back to the original XML document.

Firstly, we designed an encoding which can be applied to all XML documents. Based on this version, we further developed a second encoding, which is only applicable to POM. This version was developed as it simplifies specifically the analysis of POMs.

3.3.2 XML and POM Properties

In order to develop the encoding, the specific properties of XML and POM must be defined previously.

XML Format Properties

XML itself is a widely used language which can hierarchically display information of any kind in a document. However, in most cases, the official standard of W3C is followed, which contains a set of rules when creating an XML document. We consider this standard as it is used by Maven.

General structure An XML document consists of multiple elements which themselves may be composed of other elements. A document must have one root element which contains all other elements as children. Each element is defined by a starting and an ending tag and a name declared in the starting tag. Every element or value between the starting and ending tag of an element is considered as the content of this element. The content may consist of other child elements, a character sequence or both of these together [19]. Multiple children elements with the same name are allowed.

[1] `https://www.w3.org/`

The XML document in Listing 3.5 has *root* as its root element and *element1* and *element2* as children of the root element. Additionally, an element may have an number of attributes defined in the starting tag (*element1* has the attribute *attribute1*).

```
<root>
  <element1 attribute1="">
    value1
  </element1>
  <element2>
    value2
  </element2>
</root>
```

Listing 3.5: Example XML file

Naming conventions of elements The names of the elements must follow naming conventions. Following rules apply in the W3C standard and are of importance for this thesis [19]:

Starting characters Element names must not start with a digit or with the string "xml" in any form

Allowed characters Element names should only contain alphabetical characters, digits, colons, hyphens, underscores, periods or middle dots. Other Unicode symbols are per definition allowed, but W3C discouraged the use of them.

POM Format Properties

Generally, POMs follow strictly the W3C's XML standard. However, there are some peculiarities of POMs that further affect the encoding.

Order of Elements In most cases, the order of the elements in POMs does not influence the information contained in it. For example, if the <*build/*> section and the <*reporting/*> section are interchanged, the build process stays the same. Nevertheless, the *Apache Software Foundation* proposes a standard layout, which does not have an influence on the build but eases switching between different POMs. However, the order in certain subsections of a POM does have an impact on the build behavior of Maven.

Order of Dependencies Changing the order of the declarations of dependencies in the <*dependencies/*> section of the POM may influence the dependency resolution of Maven. This results from the procedure Maven uses to resolve transitive dependencies and its dependency mediation works [16]. Maven generates a tree of dependencies for a project and in case of conflict uses the dependency closest to the currently building project in this tree. However, if two dependencies are encountered on the same level of the tree, Maven picks strictly the first declaration of the dependency.

Order of Plugins The order of the declaration of build plugins may affect the build, but only if executions are defined for the plugins. Maven sorts different plugin executions which are attached to same lifecycle phase according to the order of their declaration in the POM [16].

Order of Modules The order of the modules is not as critical as the before mentioned ones. In a multi-module project, Maven builds the submodules not blindly according to the order of declaration in the POM. It first checks for possible inter-module dependencies of any kind and determines in this way the build order [16]. Only if no other rules apply, Maven uses the order defined in the POM.

Further Properties Whereas basic XML allows element attributes, standard elements in Maven POMs do not contain attributes. For this reason, our proposed encoding will ignore element attributes if encountered. Moreover, the content of POM elements can only be of two types: either the content is a list of further elements or a character sequence. A mixture of the types, as it is possible in standard XML, is not found in Maven POMs. Elements which only contain a character sequence will be referenced as *Leaf* elements in the following sections.

3.3.3 Method to Encode POMs into Vector Space

Taking the listed properties from section 3.3.2 into consideration, we developed a method to encode POMs into one-dimensional vectors. Based on the general encoding method, we developed a second encoding method which especially focuses on the plugin sections of POMs.

Encoding Algorithm

The general idea of the encoding is that it breaks down the tree-like structure of an XML document into an easy analyzable, one-dimensional vector. Each entry in the resulting vector corresponds to a value of a *Leaf* element of the POM. In other words, this means that if n values are set in a POM, the resulting vector only contains n values too. In order to generate the keys for the vector, the node names of all ancestor elements of the *Leaf* element are concatenated together. In this way, it is possible to exactly determine the original location of the encoded value inside the POM.

```
6 <element1>
7   <element2>value1</element2>
8   <element3>value2</element3>
9 </element1>
```

Listing 3.6: Input XML

Key	Value
element1#element2	value1
element1#element3	value2

Table 3.1: Encoded values

In order to illustrate the encoding method Listing 3.6 represents a small example of an input XML document, which equals the encoded vector in table 3.1. Important to notice is that *<element1/>* does not receive a separate key in the encoding as it is no leaf element. Another approach would have been to add a separate key for *<element1/>* which would point to a *null* value. However, since this does not provide any added value, we decided not to include these keys.

Selection of a Delimiter

In order to create the keys for the encoding, element names must be separated by an unambiguous delimiter. Otherwise, reconstruction of the encoded values would be impossible since the keys contain the necessary information. If the keys are split into wrong element names, the reconstruction is not possible. Considering the rules which apply to the naming of elements from section

Table 3.2: Encoded values of POM in Listing 3.7 with basic method

Key	Value
project#build#plugins#plugin#1#artifactId	maven-surefire-plugin
project#build#plugins#plugin#1#groupId	org.apache.maven.plugins
project#build#plugins#plugin#1#version	3.0.0-M3
project#build#plugins#plugin#1#configuration#excludes#exclude#1	**/Test1.java
project#build#plugins#plugin#1#configuration#excludes#exclude#2	**/Test2.java
project#build#plugins#plugin#2#artifactId	maven-compiler-plugin
project#build#plugins#plugin#2#groupId	org.apache.maven.plugins
project#build#plugins#plugin#2#version	3.8.1
project#build#plugins#plugin#2#configuration#source	1.8
project#build#plugins#plugin#2#configuration#target	1.8

3.3.2, certain characters could be used as delimiters which are not allowed in element names. For this reason, special characters like "#", "%" or "/" are used as delimiters, since they are not allowed in element names. Additionally, the readability of the keys is improved because special characters are easily differentiable from all allowed characters. Throughout this thesis, the # symbol is used as the delimiter in any encodings.

Handling of Duplicate Values

The method of concatenating element names to form the correct key is not always applicable as such. Many times different elements with the same name are declared at the same level of the POM. In the POM in Listing 3.7, the <*plugins*/> sections contains two elements which both are named *plugin*. This would lead to duplicate keys in the encoding: The encoding of the example POM would contain the key *project#build#plugins#plugin#artifactId* for example twice. Hence, the information contained in the encoding would become ambiguous. Therefore, an enumerator is inserted after the duplicate element name in a key. This allows encoding all configurations of the POM and that the correct order of these elements is preserved in the encoding. Normal digits can be used for this enumeration since XML element names cannot consist solely of digits as mentioned in section 3.3.2. If an element name consisted only out of digits, the property that element names are not allowed to start with a digit would be hurt.

Basic Encoding Method

Combining the above-explained rules for the encoding, we created a basic encoding method. Table 3.2 shows the encoded values of the example POM in Listing 3.7. A # symbol was used as the delimiter. At two locations, the same element occurs more than once: the <*plugin*/> element and the <*exclude*/> element. Both times, the element names are followed by an enumerator.

Plugin Name Encoding Method

The second method on how to encode a POM differs only slightly from the basic encoding method. It was developed as it allows the analysis of plugins more easily. Instead of adding the element name of a <*plugin*/> element to the key, the artifactId of this plugin is added. The according encoding of the POM in Listing 3.7 is shown in Table 3.3. This method is possible because, in contrast to Maven dependencies, a plugin is only declared once per <*plugins*/> section in a POM. In case Maven should execute a plugin multiple times, then multiple executions should

```
<project>
...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <groupId>org.apache.maven.plugins</groupId>
        <version>3.0.0-M3</version>
        <configuration>
          <excludes>
            <exclude>**/Test1.java</exclude>
            <exclude>**/Test2.java</exclude>
          </excludes>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <groupId>org.apache.plugins</groupId>
        <version>3.8.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Listing 3.7: Example POM with two declared build plugins

be declared instead of multiple instances of the same plugin [16]. The same procedure is executed if a plugin execution is encountered while encoding a POM. However, even though execution should contain a unique id, this is not required. Hence, if an execution is encountered it must be checked first if an id exists. In case an id exists, the element name can be replaced by it, if not, the element name stays in the encoding.

The idea behind this encoding method can also be extended to different sections of the POM instead of only to the plugins and execution sections. However, it should be noted that the value which replaces the element name should be (at least in most cases) unique, as otherwise this information will be lost. A disadvantage of this method is that the ordering of the replaced elements is lost. It is for example no longer possible to determine whether the *maven-compiler-plugin* or the *maven-surefire-plugin* was declared first in the original POM. The same applies to the execution section in the plugins.

Table 3.3: Encoded values of POM in Listing 3.7 with using plugin name method

Key	Value
project#build#plugins#maven-surefire-plugin#artifactId	maven-surefire-plugin
project#build#plugins#maven-surefire-plugin#groupId	org.apache.maven.plugins
project#build#plugins#maven-surefire-plugin#version	3.0.0-M3
project#build#plugins#maven-surefire-plugin#configuration#excludes#exclude#1	**/Test1.java
project#build#plugins#maven-surefire-plugin#configuration#excludes#exclude#2	**/Test2.java
project#build#plugins#maven-compiler-plugin#artifactId	maven-compiler-plugin
project#build#plugins#maven-compiler-plugin#groupId	org.apache.maven.plugins
project#build#plugins#maven-compiler-plugin#version	3.8.1
project#build#plugins#maven-compiler-plugin#configuration#source	1.8
project#build#plugins#maven-compiler-plugin#configuration#target	1.8

3.3.4 Consolidating and Filtering Encoded Vectors

In order to analyse multiple POMs, it is mostly not enough to receive a plain list of vectors but they should be prepared in some way. For this reason, we introduce an addition to our methods which prepares the encoded vectors for their future usage.

Consolidation of Vectors For analysing a single POM, encoding this POM in a vector suffices. However, in most cases, multiple POMs will be analysed together. As of this reason, a method is needed which consolidates vectors of all encoded POMs.

Each different vector contains most likely different keys, as these keys represent the structure of the underlying POM. Hence, these vectors also have different dimensions and due to this, analysing them would become more difficult or even impossible. To bypass this, it is possible to consolidate multiple vectors.

We achieve this in two steps: (1) we need to extract all keys from the vectors of all encoded POMs and form a "master" vector with the unique keys and (2) transform all vectors from the underlying POMs to the same dimension as the "master" vector. If a key of the "master" vector is present in a POM vector as well, we keep this value, whereas if this is not the case, this vector element is left empty. The result of this transformation is a set of different vectors with the same dimensions and keys.

Filtering Keys of Encoded Vectors When encoding a POM with our method, all information in the POM is encoded into a vector. However, sometimes not all information of a POM is needed, for example when only a specific section of the POMs is analysed.

In order to illustrate this, imagine a case in which only the plugins of a POM should be analysed. This means that we only are interested in keys and values of the encoding, which contain information about the plugins in the POM. The keys needed for this share some characteristics. For example, all of them are going to contain the term "plugins" somewhere, as all plugins in POMs are children of a *<plugins/>* element.

For this reason, we developed a range of filters, which can be applied to the encoded vectors to adjust them in a way suitable for the intended usage of them afterwards. Following filters are the most used ones in the course of this thesis:

Contains Element Filters out all keys that do not contain the name of the provided element at least once

Exclude Element Filters out all keys that contain the name of the provided element

Start/End with Element Filters out all keys that do not start/end with the provided element

Table 3.4: Example for trimming keys of encoded vector

Before Trimming	After Trimming
project#build#plugins#maven-compiler-plugin#artifactId	maven-compiler-plugin#artifactId
project#build#plugins#maven-compiler-plugin#groupId	maven-compiler-plugin#groupId
project#build#plugins#maven-compiler-plugin#configuration#target	maven-compiler-plugin#configuration#target
project#build#pluginManagement#plugins#maven-compiler-plugin#artifactId	maven-compiler-plugin#configuration#source
project#build#pluginManagement#plugins#maven-compiler-plugin#groupId	
project#build#pluginManagement#plugins#maven-compiler-plugin#configuration#source	

Compare "nth" Element Checks each key if the element at the "nth" position of the key matches a provided element name and filters out all other keys

These filters can be combined or applied sequentially in order to only keep the keys that are needed for the wanted analyses. In addition to these basic filters, we developed an additional, more complicated, method to adjust the encoded vectors, which we use for our data analysis.

We created a method to trim the keys of the encoded vector. Referring back to the example in which we are only interested in the plugins of the POMs, it is possible that only the part containing plugins is important at the moment. This means that all values which precede the plugin name in the encoded keys do not matter and can be cut off. Trimming the keys before using the encodings for an analysis has several advantages. Firstly, it improves the readability of the keys for the vectors. Secondly, the part of the key preceding the actual element name may produce noise in the analysis since the full path to this element is contained in the encoding. In addition, in case the same element (e.g. a plugin) is contained twice in a POM and we are not interested in the number and locations of the occurrences of the plugin but only in the union of the values part of the plugins, trimming the keys may be useful. By removing the paths to the different declarations of the plugins, the values of different declarations are merged.

Table 3.4 shows this procedure in a simplified way. The "maven-compiler-plugin" is declared in two sections in this example: one time directly in the "build" section and one time in a "pluginManagement" section. However, let us assume that in this case we are only interested in the number of different values which are set in the "maven-compiler-plugin" without carrying how many times these values are set. Therefore, we used the trimming term "maven-compiler-plugin" and are only keeping the part which follows this term. By removing duplicate keys in the result, we obtain the total set of unique values which are set inside all declarations of the "maven-compiler-plugin".

3.3.5 Tool

In order to process a large amount of POMs, we developed a Java library which is able to process POMs to encoded vectors. The core function of the library is the encoding of the given POMs with the encoding methods introduced above. Furthermore, the library assists in searching and preprocessing POMs from single and multi-module Maven projects. It allows the application of different filters, which were introduced in section 3.3.4. Additionally, the design allows to easily create new filters in case they are needed for specific data preparation. Eventually, the encodings of all provided POMs can be consolidated and exported for further analyses. The library can be found online at [21]. All encodings which are used in further sections are generated with the help of this library.

Chapter 4

Data Analysis

To answer our second research question - how Maven plugins are used in practice - we conduct a series of experiments on Open-Source Maven projects. However, we have to establish a relevant data sample and validate our proposed encoding methods to produce significant results beforehand. The analysis itself is roughly structured into two parts: We begin with generally investigating the use of plugins in Maven and then switch to examine the implications of aggregation and inheritance plugin configurations in Maven projects.

4.1 Data Collection

To analyse Maven configuration and especially the usage of plugin, we need data from real Maven projects. The goal is that the dataset consists of Maven projects which are actively developed to present a representative analysis. In this section, we explain how this dataset is gathered.

Selection of GitHub Repositories We collected the data from *Github*[1] as it is currently with over 100 million repositories and over 36 million developers the largest host for code in the world. *Github* itself provides an API[2] which allows querying public repositories according to certain parameters.

Since Maven is a Java build tool, we restrict the search to repositories that have defined Java as the main language. Furthermore, we sort these repositories according to their stargazer count. Every repository hosted on *Github* possesses a stargazer count [12]. This count indicates how popular a repository on *Github* is because many developers give a "star" to a repository to indicate that they like the repository. This should limit the result to repositories that contain real projects because otherwise, these repositories most likely would not achieve a high stargazer count. Thus, we retrieved a list of *3132* Java projects which have a stargazer count of 100 or higher.

In the next step, the repositories not containing Maven project were removed. Every Maven project must contain one POM at least and hence the repositories must contain a *pom.xml* file somewhere. Using the "search"-query provided by the *Github* API, all projects not containing a *pom.xml* were removed from the list. *662* repositories remained on the list.

Selecting Actively Engineered Projects Since GitHub is free of use and anyone can create code repositories, not every repository is necessarily an engineered software project as these repositories can be used for different purposes [7, 12]. Many repositories are used as only storage or as a small sample project. However, we want to conduct our data analysis on significant data

[1] `https://github.com/features`
[2] `https://developer.github.com/v3/`

from projects which are actively engineered. Although we already narrowed down our results by only considering repositories with a high stargazer count, we further want to exclude projects which are not actively engineered. As the stargazer count is based on how many developers liked the project, it is an indication but not a proof whether or not the repository contains an engineered project. Due to this, we have chosen to further test our repositories.

reaper is a tool that can be used to determine whether or not a repository is an actively engineered project [12]. Eight dimensions of a repository are analysed to evaluate and create a score for the repositories. A score greater or equal to 30 points is the threshold for an engineered project.

In order to use *reaper*, we created an *mySQL* database with the database dump of *GHTorrent* (1th June 2019). *GHTorrent* is a project which tries to mirror the events which happen over the public *Github* API [3]. In comparison to the official *Github* API, *GHTorrent* does not limit the search results but provides all events over the last years.

After the evaluation with *reaper*, *559* repositories with a score over 30 remained in the list.

Manual Check of Repositories After the first few analysis, the POMs of some repositories seemed different from the others. For this reason, we decided to check the repositories manually if they are indeed actual Maven projects. *44* repositories were excluded from the dataset because they turned out not to be Maven projects. The majority of these project use *Gradle* or *Apache Ant* as build system and just contained a *pom.xml* somewhere in the directory without actually using Maven.

Final Dataset Eventually, the dataset contained *515* repositories which use Maven as the main build automation tool. Furthermore, these repositories contained *14184* POMs. These POMs form the dataset which will be used in the following sections.

4.2 Validation of Encoding Methods

In the remaining analyses of this chapter, we use our proposed encoding method and rely on its correctness. Hence, we first need to validate our methods to be sure they work correctly and produce the right results.

Reconstruction of POM from Encoded Vector To validate the proposed encoding methods, we encode a POM into a vector and then try to reconstruct an XML-version of the POM from information contained in the vector. Then we compare the reconstructed POM with the original POM and check whether the information in both POM is equal. For this reason, we first define the criteria for deciding if two POMs are equal.

The reconstruction back from the encoded POM to the original POM in XML format is possible with almost no information loss. The only difference originates from the loss of the order between non-duplicate elements. Taking the encoded values from Table 3.2 as an example, it is not possible to determine whether the *groupId* or the *artifactId* of a plugin was declared first when taking only the encoded values into consideration. This bases on the assumption that the order of the encoded values is arbitrary.

Although the order of the elements in a reconstructed POM cannot be fully maintained, the POM is nevertheless valid, fully functional and contains the same configurations as the original POM. The only elements in a POM of which the order of declaration has an influence on the build, are described in section 3.3.2. As dependencies and modules are enumerated by both of the above-introduced methods, Maven interprets the POM in the same way as the original one. While the first encoding method preserves the order of plugins and thus the total order of the executions, the second method loses this information due to its design.

Validating the Encoding Methods Keeping these criteria in mind, we created a test that compares the structure of two XML-versions of POMs. Starting from the root node of the XML-versions, the test checks recursively whether all child nodes and their subtrees are equal. In case, the child element forms a section of a POM in which the order of the elements matter, the test additionally checks whether the children of the two nodes are in the same order. In other cases, the children are compared without taking their order into consideration.

In this way, we encoded the *14184* POMs from our dataset into vectors, reconstructed XML-versions from these vectors and compared them pairwise with each other. We did this for both encoding methods proposed in sections 3.3.3 and 3.3.3. With the basic encoding method, we were able to reconstruct all except for *two* POMs without loss of information. After manual inspection of these two POMs, we found that the error was caused by our test rather than the method itself. As soon as an element with the name "dependencies" was encountered, we checked for the correct order of the children. However, in these two cases, the child elements had different names, which is why the order could not be maintained during the encoding and the test failed.

The plugin name encoding method was able to correctly reconstruct all except *23* POMs. These failures resulted from POMs which have declared the same plugin several times in the same section. As plugins which appear multiple times, are dropped by our method, it was not possible to reconstruct these POMs. *The Apache Software Foundation* states that in case a plugin should be executed multiple times, multiple executions should be added to one declaration of the plugin [16].

Hence, in over *99%* of the cases, the methods succeeds to reconstruct the encoded POM without loss of information.

4.3 Data Analysis of Plugin Usage in Maven

Having established a dataset and verified that our proposed encoding methods works properly, we analyse how plugins are actually used in Maven projects. For this reason, we conduct a series of analyses with the Maven projects in our dataset in this section.

4.3.1 POM Distribution in Repositories

The gathered dataset contains *14184* POMs distributed over *515* repositories. However, there are some major differences in the number of POMs per repositories.

Table 4.1: Distribution of POMs across repositories

	# of Repositories	Mean	STD	Min	25%	50%	75%	Max
All	515	27.54	66.97	1.0	1.0	7.0	21.5	871.0
Without Outliers	453	9.66	11.54	1.0	1.0	5.0	14.0	49.0

Table 4.1 shows the big differences in the number of POMs across all repositories. We found that in average *27.5* POMS are contained in a repository; however as the median lies only *7* POMs, the mean seems strongly influenced by some repositories with a very high number of POMs. For this reason, we removed all repositories of which the POM count exceeds the third quartile plus 1.5 times the interquartile range. This leads to a more reasonable average of *9.66* POMs in a repository.

Table 4.2: Proportion of Different Parts in POMs

	Percentage of Dependency Values	Percentage of Plugin Values	Percentage of Remainder Values
Mean	38.8%	17.3%	43.9%
STD	27.3%	21.1%	28.8%
Min	0.0%	0.0%	0.3%
25%	14.3%	0.0%	22.1%
50%	40.0%	8.0%	35.7%
75%	60.4%	31.4%	58.8%
Max	99.7%	97.9%	100.0%

Furthermore, we found *158* repositories, which only contain one POM. This is as much as *31%* of all repositories. Thus, these repositories only contain a single Maven Project.

4.3.2 Distribution of Different Content Sections in POMs

Before starting the analyses on Maven plugins themselves, we want to understand how much of Maven's configuration is used actually to configure plugins. In section 3.2, we described the different sections a POM may contain. A POM may consist of several different sections, but for now we abstract into three parts:

Dependencies This part covers all sections in a POM that deal with dependency management.

Plugins This part covers all sections in a POM that deal with plugin management.

Remainder This part covers all the remaining parts of a POM, for instance, the Maven coordinates, the distribution management, and the project management.

Our proposed encoding method encodes a POM into a vector where each entry of the vector stands for a value set in in the configurations. Hence, it is possible to count the values set for each of the previously mentioned parts of a POM. The number of values indicates which parts of a POM are most extensively configured.

To extract all values for a given part, the keys of the encoded POMs were filtered. Every dependency in a POM is contained in an *<dependency/>* element, which itself is part of a *<dependencies/>* element. Hence, any key that contains the substring "dependencies#dependency" points to a value that is used for dependency management. The plugin declarations are filtered out in the same way. From the remaining values after the extraction of the dependency, all keys which contain the substring "plugins#plugin" are selected. Finally, all remaining keys form the third part.

Table 4.2 shows the results for our dataset with *14184* POMs. There are major differences between the number of values used to declare dependencies and plugins. Whereas the dependency declaration makes up *38.8%* of a POM in average, only *17.3%* of the values are used to declare plugins. With *43.9%* of the values, the group with the remaining values forms the biggest part of a POM on average. However, this is not surprising as it accumulates the biggest number of different POM sections.

Furthermore, in 50% of the POMs, only *8%* or less of the values are used for plugin management. On the other hand, the median of the values for dependencies is at *40%* of the values of

Table 4.3: Total number of POMs and number of different plugin declarations per POM

	All plugin declarations	Different plugin declarations
Total Number of Poms	14,184	14,184
POMs with no Plugins	6,460	6,460
POMs with one Plugin	2,861	3,000
POMs with more than one Plugin	4,863	4,724

a POM which is even higher than the percentage of the remaining values. Hence, in most of the POMs, dependency management makes up the majority of the POMs.

4.3.3 Plugin Usage in POMs

Moving away from the analyses of the whole content of POMs, we now focus explicitly on the usage of Maven plugins. In this section, we want to answer the question *how frequently, and which, plugins are declared in POMs throughout Maven projects*.

For the first analysis, we used the *Plugin Name Encoding Method* to encode the POMs of all repositories into vectors. Then, we counted the frequency for each plugin in each POM of our dataset. In a POM, each plugin is declared inside a <*plugins*/> section. Furthermore, each plugin contains a <*artifactId*/> element, which contains the id of the plugin. To extract the plugins, we applied a series of filters on the keys of each vector. First, we removed all keys that do not contain a *plugins* node or do not end on *artifactId*. Since it is possible that somewhere else in the plugin declaration another element with the name *artifactId* exists, we removed all keys that contained a node named either *executions, configuration* or *dependencies*. Hence, for each plugin declaration, only a single value containing its id stayed in the encoded POM. Then, we consolidated the vectors of the encoded POMs and counted the frequency of each plugin in each POM.

Table 4.3 shows how many POMs exists which only contain zero or one plugin. A large group of POMs, which makes up *45.5%* of all POMs, does not declare any plugins at all. Another *20.2%* of the POMs only declares a single plugin. When checking for declarations of different plugins inside POMs, this group becomes even larger and contains *3000* POMs, which form *21.2%* of all POMs. Moreover, on average the POMs in our dataset only contain *1.8* plugin declaration. When removing POMs which do not contain any plugins, the average rises to *3.31* plugin declaration per POM.

Furthermore, the analysis showed that major differences in the number of occurrences between the different plugins exist. In total, the POMs in our dataset contained *562* plugins with different artifactIds. However, only *458* of plugins have an artifactId which contains the string "plugin", which should be contained in a plugin artifactId according to the Maven guidelines [16]. For this reason, we decided to exclude those plugins from our analysis for this and all following analyses.

On average, a single plugin is declared *49.3* times in our dataset with a standard deviation of *189.76*. However, the median is only *3* declarations per plugin. Figure 4.1 shows the density plot of the declaration counts of the single plugins. The plugins are sorted according the number of occurrences in descending order. It is very heavily skewed to the right, as only a small number of plugins are declared many times. The majority of plugins are declared only a few times. The number of occurrences of plugins ranged from *one* up to *2232* and all of ten most declared plugins are official Maven plugins [16]. Moreover, of the *35* plugins which are declared more than a hundred times, *22* are official Maven plugins.

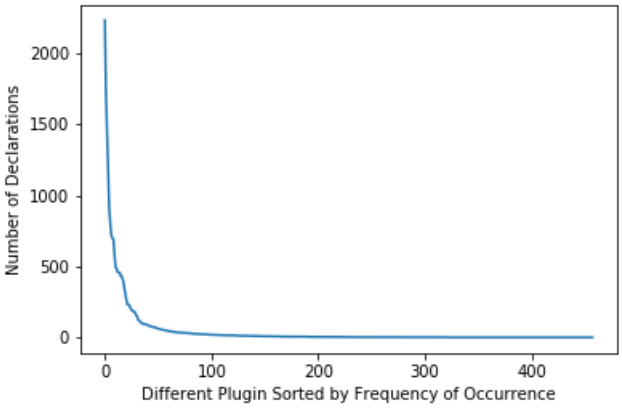

Figure 4.1: Number of declarations per plugin. Each plugin is given an index (0-457) according to its frequency of occurence

Configurations of Maven Plugins One advantage of Maven plugins is that they provide the possibility to adjust their behavior by configuring them. We want to find out *how often plugins are actually configured in POMs and, hence, overwrite the standard configuration provided by Maven*.

Each plugin may contain four main configuration sections: <*configuration/*>, <*executions/*>, <*dependencies/*> and <*inherited/*>. For each of these sections and each plugin separately, we extracted the frequency of occurrence from their encoded vectors.

The procedure to extract the frequency of the different configuration sections was equivalent for each section why we only describe it for the <*configuration/*> section. We first determined all plugins which are used in the POMs in the same manner as the last analysis. Then, we encoded all POMs, removed all keys which do not contain the node "plugins" and eventually trimmed all remaining keys in order that they start directly with the plugin artifactId. We further removed all keys in which the second node was not equal to "configuration". Hence, only keys persisted which point to a value of the configuration section of a plugin. Eventually, we grouped the keys by their first node name (the plugin's artifactId) and counted the configurations for each plugin in this POM. Table 4.4 summarized the results for all sections.

In total, the POMs in our dataset contained *22756* plugin declarations. *77%* of the plugin declarations contain any type of configuration section. *49.1%* of the plugin declarations contain a "configuration" section. In the declarations, which contain this section, the configuration consists of *3.88* values on average when weighing the averages of the different plugins in regards to their frequency count. We further checked whether a correlation between the frequency of occurrence and the average number of configurations in plugins exists. This may indicate whether plugins that need fewer configurations are used more often. However, with a Pearson correlation coefficient of *-0.041*, no linear correlation exists.

We conducted the same analysis on the other three configuration sections of Maven plugins. *39.8%* of the declarations contain an execution section with an average of *6.7* values in them. With a Pearson correlation coefficient of *0.008*, no correlation between the frequency of a plugin and the number of execution values exists in this configuration section too.

With *3.2%* and only *1.9%* of the plugin declarations containing these sections, the "dependencies" and "inherited" section occur very rarely. On average, a dependency section contains *4.84* values and no correlation exists between frequency and number of dependency values as before with the other sections. The inherited contains in average exactly 1 value which is trivial as this

Table 4.4: Frequency of occurrence of different plugin configuration sections

	Configuration Section	Executions Section	Dependency Section	Inherited Section
Percentage of Plugin Declaration Containing this Section	49.10%	39.80%	3.20%	1.90%
Average Number of Values in this Section	3.88	6.7	4.84	1

section consists only of one XML-element in a POM.

4.3.4 Aggregation and Inheritance in Maven

For the second part of our analysis how Maven plugins are used in practice, we examine how plugins are configured in the context of aggregation and inheritance. However, before starting analyzing how inheritance in Maven projects influences plugin configurations, we investigate the importance of aggregation and inheritance in Maven. By doing this we want to answer three questions: (1) *How frequently is aggregation and inheritance used in Maven projects?*, (2) *What is the average level of abstraction?* and (3) *How often aggregation and inheritance are used in combination?*.

Inheritance in Maven In Maven, every POM inherits implicitly from a Super POM which contains all standard configurations. However, we did not include this relation as it would not provide any additional value to the analysis. Furthermore, we only included parent POMs in case they were also part of the same repository. If a parent POM of a POM inside the repository is not included in the same repository, the relation is not included in the analysis.

Table 4.5 shows the statistics about the highest found inheritance level in the repositories and the percentage of the POMs inside a repository which are part of an inheritance relation. In Maven, inheritance is a transitive process meaning that inheritance can appear over multiple levels. If the largest inheritance level equals zero, the repository does not have any inheritance relations between its POMs. If the level equals one, at least one POMs has a parent inside the same repository, and if it was two, the POM additionally has a grand-parent. We found that the repositories of our dataset on average had a maximal inheritance level of *1.08* and *61%* of the repositories have an inheritance level of one or more. When removing repositories that only contain a single POM (and hence do certainly not contain inheritance), the average inheritance level rises to *1.55* and *88.8%* of the repositories have an inheritance level of one or higher.

Furthermore, *57.2%* of the POMs inside a repository are part of an inheritance relation on average as table 4.5 states. Considering the median, which is less influenced by outliers, even *88.9%* of the POMs are part of inheritance relations in a repository.

Aggregation in Maven Similar to the analysis of the inheritance, we prepared the data for the aggregation analysis. We only included submodules which were themselves part of the repository. Furthermore, we only considered the modules of the aggregator which were declared in the main part of the POM. Submodules which are defined inside profiles in POMs are not part of the data.

Table 4.5 contains the values gathered from our dataset. The average highest aggregation throughout all repositories lies at *1.08* and the median at *1*. When we remove again the projects,

Table 4.5: Metrics of Inheritance and Aggregation Usage in Maven projects

	Highest Inheritance Level	Percentage of POMs in Inheritance	Highest Aggregation Level	Percentage of POMs in Aggregation
Mean	1.08	57.2%	1.08	59.3%
STD	1.22	46.8%	1.16	46.7%
25%	0	0%	0	0%
Median	1	88.9%	1	91.7%
75%	2	100%	2	100%
Max	8	100%	8	100%

which only contain a single POM file, the average highest aggregation level becomes *1.56* and *90.8%* of the repository have a level of one or higher.

Similarity of Inheritance and Aggregation The values obtained by the analysis of inheritance usage are very similar to the ones of the aggregations but not fully equal. Additionally, when considering figure 4.2, it seems as if the distributions of aggregation and inheritance levels across our dataset are almost equal.

This is surprising as inheritance does not automatically imply aggregation and vice versa [16]. We further checked how many inheritance relations are also aggregation relations. For this reason, we created tuples of parent-child and aggregator-submodule POMs for each repository. We then checked whether a child is also a submodule of the parent and whether a submodule of an aggregator is also a child.

Referring to table 4.6, we found that on average *92%* of the children are also submodules of the parents per repository. In more than half of the repositories, even all children are submodules. On the other hand, slightly fewer submodules are children of the aggregator on average.

4.3.5 Plugin Configuration in Inheritance

After analyzing the usage of inheritance and aggregation in Maven projects, we focus on the plugin usage with regards to inheritance in this section. In a first step, we analyse if more plugins are declared in a parent or a child POM. Afterwards, we particularly focus on the distribution of the configurations of plugins between parents and children.

Maven plugins (and their configurations) are inherited by child POMs and it is a good practice to declare plugins, when possible, in the parent rather than in the child. Hence, assuming developers follow these practices, there should be a *higher number of plugin declarations as well as configurations in parent than in child POMS*. In this section, we try to prove this hypothesis.

For all the following analyses, we only consider plugins and configurations which are declared in the main part of the POMs and not possible declarations in profiles. As before, we generated pairs of all parent-child POMs for each repository. To generate the necessary data for all analyses, we used the *Plugin Name Encoding Method* to generate encoded vectors of the POMs.

Distribution of Plugin Declarations In this part, we are interested in the differences in the number of plugin declarations between parent and child POMS. We omit plugins which are only part of a "pluginManagement" section and consider only plugins that are directly part of build or reporting sections. Only these plugins are actually executed during the build or site lifecycle.

To test our hypothesis, we created parent-child pairs of POM from our dataset, which let to *12076* individual pairs. Then, we used the same procedure to reduce the encoded vector to a vector, only containing a single value per plugin as previously explained in section 4.3.3. Next, we removed all pairs in which neither of the two POMs contained a plugin which reduced the

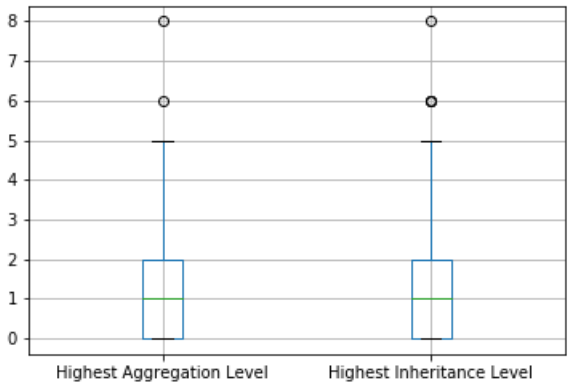

Figure 4.2: Box-and-Whisker plot of highest inheritance and aggregation levels across all repositories

Table 4.6: Similarity between aggregation and inheritance in Maven projects

	Percentage of Children that are Submodules	**Percentage of Submodules that are Children**
Mean	0.92	0.89
STD	0.21	0.25
25%	0.96	0.95
Median	1	1
75%	1	1
Max	1	1

number of pairs to *8965*. These pairs are located in *317* different repositories. Then, we compared the number of plugins declared in the parent and the number of plugins declared in the child that were not already declared in the parent. The union of these two sets of plugins equals all the plugins which are executed during the build of the child POM. We summarized the result in table 4.7.

We found that in average *60%* of the plugins which are executed during the build of the child POM have originally been declared in the parent POM. With a median located at *80%*, the difference in the number of plugin declarations between parent and child is even more significant.

In addition, we grouped all pairs by their repositories and calculated the average percentage of plugins declared in the parent POMs firstly for each repository and only then for all repositories. This minimizes the influence of a single repository with an above-average number of pairs as only the average of all pairs per repository is considered. The results are found in table 4.7. Although the mean (*59%*) and the median (*70%*) decreased slightly in comparison with the previously obtained values, the first quartile increased and the variation became less. Hence, the average that about *60%* of the plugin declarations are located in the parent POM is stable overall repositories.

Distribution of Plugin Configurations Using the same procedure as before, we started by creating pairs of all parent-child POMs from our dataset. Then, we had to isolate the configurations of each plugin in the POMs. To achieve this, we used the same method we already used in section 4.3.3 so that only values of "configuration" sections of each plugin remained in the encoded vectors. However, since plugins can be declared directly in the build (or reporting) section and simultaneously in the "pluginManagement" section, we merged the configuration contained in the two sections for each plugin. In case, the same configuration was present in both sections, only the one directly declared in the build section was considered. Thus, the only remaining values inside the encodings of the POMs were the configurations of each plugin.

For each pair and plugin, we then divided the number of configurations in the parent by the total number of configurations that apply to the plugin in the child POM. The divisor represents, in this case, the union of all configurations which apply to the child plugin; either directly declared in the child or inherited from the parent. From this we obtain the percentage of the configurations which are inherited from the parent. We only considered pairs of POMs in which either the parent or the child contains at least one configuration, otherwise the pair is dropped.

Eventually, we consolidated the values obtained in each pair of POMs for each plugin. When omitting the frequency of occurrences of the different plugins, we found that in average *43.5%* of the configurations of a plugin in a child are inherited from the parent with a standard deviation of *43.75%*.

As the frequency of declarations of the plugins differs significantly - between *one* and *3050* occurrences in all pairs with a coefficient of variation as high as *3.14* - we further calculated the weighted mean of the percentage of the configurations that are declared in the parent. For this reason, we multiplied the mean percentage of every plugin by the number of occurrences of this plugin and divided it by the total number of occurrences of all plugins. Now, even *79.14%* of the configurations of a plugin in a child POM are declared in the parent on average.

Frequency of Overwritten Configurations Lastly, we examined how often configurations in a plugin, which are inherited from a parent POM, are overwritten in a child. In the same way as before, we extract the configurations of each plugin for all parent-child POM pairs. We then compared the configurations between the parent and the child POM for each plugin to gain following two measures:

- The number of configurations that are overwritten in the child POM. In this step, it does not

Table 4.7: Distribution of plugin declarations in parent and child POMs

	Overall Percentage of Plugin Declarations in Parent	**Grouped by repositories**
Mean	0.6	0.59
STD	0.43	0.35
25%	0	0.3
50%	0.8	0.7
75%	1	0.89

matter whether the value of this configuration is different or equal to the value in the parent POM.

- The number of configurations that are overwritten in the child POM unnecessarily. In other words, these are the configurations that are overwritten with the same value they already had in the parent POM.

We then divided the obtained values by the number of declared configurations in the child. The received measure equals the percentage of configurations in the child, which are overwriting configurations from the parent. Afterwards, in the same manner as before, we calculated the weighted mean regarding the frequency of concurrence of the different plugins in the parent-child POM pairs.

On average, *3.17%* of the configurations set in a plugin of a child POM overwrite configurations which are inherited from the parent. Furthermore, *2.06%* of the configurations in a plugin of a child POM overwrite an inherited configuration with the same value it already contained. Combining these two results, we find that almost *two-thirds* of the configurations overwriting inherited values are not necessary and could be left out in the child declaration of the POM.

Chapter 5

Discussion

After summarizing the necessary background information on Maven, proposing a method to encode POMs into vectors and conducting data analysis on real Maven projects, we join the findings from these parts to answer the research questions of this thesis. During this discussion, we propose ideas how to continue this research or use it in other research areas. Eventually, we introduce possible threads to the validity our findings.

5.1 Addressing the Research Questions

In the first chapter of this thesis, we proposed two research question, which this thesis should answer. In this section, we answer them one-by-one with findings and data from the previous chapters.

RQ1 How can configurations in POMs be extracted to further analyse them?

To answer this question, we introduced a method to encode Maven POM files into one-dimensional vectors in section 3.3. Additionally, we summarized the necessary background information on Maven, XML in general and especially Maven POM files to establish a shared understanding of the problem domain.

Our proposed method can encode the configurations which are contained in a POM into a one-dimensional vector. The basic method encodes the POM file without loss of information and without invalidating the original information contained in the POM. This is especially important when conducting content analyses on the received vectors afterwards.

Based on this basic encoding method, we introduced a second method which is particularly designed to encode Maven plugin sections in POMs. This is achieved by switching element names at specific locations with different values. In this method, when a <*plugin/*> or a <*execution/*> node is encountered, the artifactId, or the execution id respectively, replaces the element name.

When validating this method, we found that over *99%* of the encoded POMs could be reconstructed without losing information which the POM contained beforehand. The failed reconstructions resulted from errors in the POMs that were not accepted by our method.

One shortcoming of this method is that the attributes of XML elements are not included in the resulting vector. Another possible obstacle of using this method is that duplicate values are enumerated. Due to the importance of the order in some sections of a POM, we nevertheless decided to construct the method in this way. However, when analyzing the encoded vector it is important to remember this.

A further note is that this method is not only limited to encoding Maven POM files but can be used for all possible XML-documents. Additionally, the method allows reconstructing an XML

version of the values encoded, which we momentarily only use for the validation of the method. However, the possibility to reconstruct the encoded values may prove useful in other scenarios. This allows encoding an XML document, with the help of filters from section 3.3.4 extract all needed information and finally reconstruct an XML, which only contains wanted values. This provides the possibility to apply these encoding methods in different areas that have to analyse or edit the contents of XML-files.

Another application of this method could be to encode POMs of multiple projects into vectors and then examine whether it is possible to classify or group projects according to their configuration. By applying certain combination of filters, the classification could be reduced to only being dependent on certain subsections of a POM; for example only taking plugins, dependencies or other sections of a POM into consideration.

Answer Second Research Question Using this encoding method, we carried out a data analysis on the plugin usage in Maven POM files and answer following research question:

RQ2 How are Maven plugin used in practice in Open-Source projects?

To answer this research question, we conducted a wide range of different statistical tests on our dataset. In a first step, we compared the distribution of the different content sections of a POM. We found that on average only *17.3%* percent of the configuration set in a POM, are used to declare plugins. Surprisingly, the dependency management in POMs is responsible for *38.8%* of the values, which is very high considering that on average the remainder section makes up *43.9%* of the values. This signifies that the configuration of plugins is only a minor part of the Maven configuration. This distribution is even more noticeable when checking the medians. Whereas the medians of dependency and the remaining sections only slightly differ from their means, the median of the plugin section lies at only *8%*.

Together with our second analysis, which shows that only *4724* of *14184* the POMs *(33.3%)* in our dataset declare more than one different plugin, it is possible to state that the plugin management only makes up a small part of the total configuration of Maven. On the other hand, the dependency management forms a big part of the Maven configuration. Additionally, other studies [9, 17] have found that dependency errors cause a big percentage of build failures. Combing these findings, the dependency management forms the most dominant part of the configuration as well as the most critical one regarding build success.

Making a step away from analyzing a POM as a whole, we continued investigating Maven plugins more in detail without their surroundings. We found that the different plugins are very unequally distributed across the POMs in our repositories. We found *458* different plugins in all POMs, but only *35* of them were declared more than a hundred times. Furthermore, most of the plugins declared often are plugins that are officially developed by *The Apache Software Foundation* themselves. A reason for this might be that many plugins execute a very specific task which is only needed in a single project and are probably privately developed. This assumption is strengthened by the fact that the *Maven repository* hosts thousands of different Maven plugins as of the time of this writing [16]. It is likely that some of these plugins execute a similar - if not even an equal - task.

Afterwards, we showed how Maven plugins are configured and found that *77%* of the plugins are declared containing an additional configuration. We found that *49.1%* of the declared plugins contain a "configuration" and *39.8%* contain a "executions" subsections. With only *3.2%* and *1.9%* of the declared plugins containing a "dependencies" and "inherited" subsection, these two sections are very rarely found. Also, in case a configuration was present in a plugin declaration, only very few values were set on average. We assumed that the frequency of plugins might influence the number of configurations as more widely used plugins probably are more standardized

(hence need more individual adjustments) than plugins developed for a specific use case. However, we found no evidence that between the number of declarations and the average number of configurations of a plugin a correlation exists.

Finally, we examined how plugins are used in inheritance. For this reason, we first analysed how aggregation and inheritance are utilized in general to determine their importance in Maven projects.

We found that *61.6%* of the repositories in our dataset used inheritance and *62.9%* aggregation respectively. Furthermore, most of the repositories not using inheritance or aggregation, consisted only of a single project, which trivially explains why they do not use these types of relations. Moreover, considering only repositories with inheritance or aggregation, on average *92.9%* of the POMs in a repository are involved in an inheritance relationship in a project. With *94.2%*, the percentage of POMs which are part of aggregation in a repository is even slightly higher. Hence, we can conclude that the big majority of multi-module project in our data set uses inheritance, aggregation or both.

The majority of the repositories, which contain more than one POM, have both a maximal inheritance and aggregation level of *one*. This can be interpreted as such: They contain a parent (or aggregation) layer and a child (or submodule) layer. It shows that, although multi-level relations are possible, most of the repositories contain only one level of abstraction.

Even tough that aggregation does not imply inheritance and vice versa, we found strong evidence that in most cases these two features are used in combination. On average, *92%* of POMs that have a parent, are submodules of this parent too. In the other direction, *89%* of the submodules of an aggregator, have declared the aggregator as their parent.

Since inheritance is widely used, we further analysed how Maven plugins and their configuration are affected by inheritance relations. We found that in average *60%* (median at *80%*) of the plugins, which are executed during a build or site lifecycle of a child project, are actually declared in the parent project. When analyzing whether the configurations of an executing plugin are declared in the parent or child POM, we found that *79.14%* of the configurations are set in the parent and are inherited by the child. Both results suggest that inheritance is extensively used in the configuration of plugins. However, as not all configurations are inherited, we checked how much of the configuration of a plugin in a child is used to overwrite inherited configurations. We found that only a small part (*3.07%*) of the configurations in a child plugin overwrites inherited values. However, the major part of these configurations overwrites the inherited configuration with the same value, which, when unintentionally done, can cause problems. For example, when changing a value in the parent, which is overwritten by the child, it does not have any effect on the build of the child. Although in many cases, some configurations have to be set specifically in a child, it is wise to configure plugins, when possible, in the parent in order to group the configurations at a central point.

In conclusion, although Maven plugins play an important role in every part of the build, configuring plugins only makes up a small amount of all configurations of a project. It seems as if the standard configuration of plugins suffice in many cases and only small adaptations have to be made in order for them to work correctly. As one of the concepts of Maven is *Convention of Configuration*, this seems reasonable.

Our analysis is based on a static image of the configuration of Maven without considering the influence from or on external factors. Some studies [1,10] have already found that the complexity of the build system configurations growths in correlation with source code size. However, it would be interesting to analyse further whether certain types of configuration changes cause more build failures, longer build times or extensive adjustment in the source code.

With our analyses, we contribute various new insights into the usage of Java build tools and especially Maven. Firstly, we show that plugin management only makes up a small part of all

configurations, whereas the dependency management accounts for the majority of the configurations. Hence, when researching Maven and other build systems, the central role of dependency management should not be neglected.

Furthermore, we show that only few plugins are used frequently and additionally, only a small number of configurations is needed. This indicates that the standardized build approach of Maven is successful since the same (standard) configurations can be shared across many projects without many adjustments.

Lastly, we find that inheritance and aggregation are used in the majority of projects and have a significant influence on the configuration practices of plugins. When conducting research related to build tools, which allow inheritance, it should be considered that the majority of the configurations are inherited and not located in the executing POM itself.

5.2 Threads to Validity

Internal Validity For our theses we generated a dataset which contains a rather small sample size of *515* repositories. We only considered repositories with a high stargazer count on GitHub and further used the *reaper* tool in order to extract actively engineered projects, but it is still possible that there exists a certain bias in our repositories [12]. Moreover, as we used *reaper* to classify the repositories, it is possible that repositories were excluded which are in fact actual engineering project, however *reaper* does not recognize them as such.

Furthermore, we used all POMs that are located in a repository. However, by going through the repositories, we found that sometimes repositories contain POMs, which are not part of the project itself. Hence, these POMs may have introduced a certain bias to the obtained results.

Generalizability All analysis in this thesis are based on Maven, but how are these findings applicable to other build systems? Besides Maven, Apache Ant[1] and Gradle[2] are two other famous Java build tools. Even though all of those tools have the main goal of converting Java source code to executable software, there exist some major differences between them.

Apache Ant (ANT) is another build automation tool of the *Apache Software Foundation*. The biggest difference compared with Maven is that ANT does not follow the principle of *"Convention over Configuration"*. There are no standardized or uniform build procedures; every build step must be declared in the configuration file of ANT. This allows higher flexibility during the build process, but the configuration files may grow over-proportionally and become difficult to understand. Furthermore, the original version of ANT did not contain native dependency management. This was adjusted when later on *Apache Ivy*[3] was integrated, which offers a dependency management [15]. As there exists a major difference between ANT and Maven regarding the way of configuring the build and the range of functions, it is not guaranteed that our findings apply to this build system as well.

In contrast to ANT, Gradle is more similar to Maven. It is a build automation and dependency management tool which was released in 2007 [4]. It is based on concepts from Maven as well as ANT and follows the principle of *"Convention over Configuration"* too but nevertheless tries to offer the flexibility of ANT. As opposed to the previously introduced build tools, Gradle uses a domain-specific-language based on Groovy[4] instead of XML for its configuration. This leads to smaller and more compact configuration files. Even though Maven and Gradle are different tools,

[1] `https://ant.apache.org/`
[2] `https://gradle.org/`
[3] `https://ant.apache.org/ivy/`
[4] `https://groovy-lang.org/`

they are similar in their operating principles and scope of functions. Hence, it seems possible that our findings may apply to this tool too.

Chapter 6

Summary

In this thesis, we examine in-depth how Maven plugins are configured in practice by analysing configurations of Open-Source projects using Maven. During this thesis we address two research questions: (1) how can configurations in POMs be extracted to further analyze them and (2) how are Maven plugins used in practice in Open-Source projects.

Firstly, we propose a method which allows to encode POMs, the Maven configuration files, into the vector space. Our approach is focused on prevailing the informational integrity since every information change in the encoded vector may introduce invalid configurations.

By using this method, we conducted several analyses on the usage of plugins in Open-Source Maven projects. Despite of the importance of plugins, we find that plugin management only makes up a small portion of the complete Maven configuration. On the other hand, dependency management makes up the majority of the configuration of Maven. Additionally, we find that the standard plugin configuration provided by Maven suffices in most cases and only little adjustments on them are made.

We show that inheritance and aggregation is used in the majority of Maven projects and that plugins and their configurations are strongly influenced by inheritance in Maven projects. Most of the plugins and their configurations are inherited in child POMs. This should be taken into consideration when researching a topic that includes Maven configurations.

With this thesis, we provide further insight how build systems are actually used by developers. Moreover, we provide a new approach to encode configuration files based on XML. Since our proposed method can be used quiet flexibly, we believe that it also might have applications in other areas than only configuration analysis.

Bibliography

[1] B. Adams, K. de Schutter, H. Tromp, and W. de Meuter. The evolution of the linux build system: Electronic communications of the easst, volume 8: Ercim symposium on software evolution 2007. 2008.

[2] C. Désarmeaux, A. Pecatikov, and S. McIntosh. The dispersion of build maintenance activity across maven lifecycle phases. In A. S. I. G. o. S. Engineering, editor, *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 492–495, New York, New York, USA, 2016. ACM Press.

[3] G. Gousios. The ghtorent dataset and tool suite. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 233–236. IEEE, 18.05.2013 - 19.05.2013.

[4] Gradle Inc. Gradle documentation.

[5] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In D. Lo, S. Apel, and S. Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 426–437, New York, New York, USA, 2016. ACM Press.

[6] V. Kakade and P. Raghavan. Encoding xml in vector spaces. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, D. E. Losada, and J. M. Fernández-Luna, editors, *Advances in Information Retrieval*, volume 3408 of *Lecture Notes in Computer Science*, pages 96–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[7] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In P. Devanbu, S. Kim, and M. Pinzger, editors, *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 92–101, New York, New York, USA, 2014. ACM Press.

[8] G. Kumfert and T. Epperly. Software in the doe: The hidden overhead of"the build".

[9] C. Macho, S. McIntosh, and M. Pinzger. Automatically repairing dependency-related build breakage. In *25th IEEE International Conference on Software Analysis, Evolution, and Reengineering : SANER 2018 : Campobasso, Italy : proceedings*, pages 106–117, Piscataway, NJ, 2018. IEEE.

[10] S. McIntosh, B. Adams, and A. E. Hassan. The evolution of java build systems. *Empirical Software Engineering*, 17(4-5):578–608, 2012.

[11] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan. A large-scale empirical study of the relationship between build technology and build maintenance. *Empirical Software Engineering*, 20(6):1587–1633, 2015.

[12] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan. *Curating GitHub for engineered software projects*. 2016.

[13] P. Smith. *Software build systems: Principles and experience*. Addison-Wesley, Upper Saddle River, N.J., 2011.

[14] M. Sulír and J. Porubän. A quantitative study of java software buildability. In C. Anslow, editor, *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 17–25, New York, NY, 2016. ACM.

[15] The Apache Software Foundation. Ant documentation.

[16] The Apache Software Foundation. Maven documentation.

[17] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838, 2017.

[18] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, and S. Panichella. A tale of ci build failures: An open source and a financial organization perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution*, pages 183–193, Los Alamitos, California, 2017. IEEE Computer Society, Conference Publishing Services.

[19] World Wide Web Consortium. Extensible markup language (xml) 1.0 (fifth edition): W3c recommendation 26 november 2008.

[20] C. C. Yang and N. Liu. Measuring similarity of semi-structured documents with context weights. In E. N. Efthimiadis, S. Dumais, D. Hawking, and K. Järvelin, editors, *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '06*, page 719, New York, New York, USA, 2006. ACM Press.

[21] M. Zwimpfer. marzwi/MavenPluginExtractor: Finished Bachelor's Thesis. Zenodo. Aug. 2019. doi: 10.5281/zenodo.3374936