Automating Open Source Software License Information Generation in Software Projects

Sergius DYCK, Daniel HAFERKORN, Christian KERTH and André SCHOEBEL
Fraunhofer IOSB, Karlsruhe, Germany

ABSTRACT

This publication deals with Open Source Software (OSS) compliance. In a previous publication [1], we presented an organizational-technical concept for ensuring basic OSS compliance. Based on this concept, we now address further aspects that are essential to OSS compliance. Our focus is on methods for avoiding license infringements by automated generation of OSS notice lists.
We describe means to manage OSS license (OSSL) information of directly and indirectly used OSS. We use methods for establishing a common domain language based on a Domain-Driven Design (DDD) approach that leads to a better communication between experts from different fields, e.g., technical and domain experts, when discussing OSS compliance and developing our solutions. Furthermore, we present already existing Maven tools as well as self-developed Java tools, which make it possible to store the information that has been gained during the OSS compliance process in a structured way. With the aid of said tools, this information can then be used to create the lists of used OSS suitable for internal audits, external software deployments and software deliveries automatically to reduce manual effort and risk of errors.

Keywords: Open Source Software, Open Source Compliance, License Information Management, Notice Obligation, Document Generation, Software Engineering, Maven.

1. INTRODUCTION

Today's software projects usually make use of various third party libraries to reduce both considerable engineering effort as well as development time. Depending on the project, the software may also include many third party libraries released under possibly different OSSL. Using OSS without considering its license conditions can lead to various pitfalls. In particular, legal licensing conditions must be complied with when using OSS. Therefore, for each OSS that is newly introduced in software projects it has to be checked that its OSSL can be complied with before it can be used. License infringements are attributable to the organization responsible for the inclusion of OSS into their projects and may entail significant consequences in the event of a legal dispute, e.g., in terms of monetary cost or reputation. The burden of proof is seen on the side of said organization. Thus, it is crucial for organizations developing software to keep the risk of accidental license infringements as low as possible. This practice is often associated with the term compliance, or more specific, Open Source compliance. In this paper, we will use the term OSS compliance to address OSS specifically.
This paper outlines our work that resulted in an ubiquitous language, a Maven [2] plugin for automated creation of a list of used OSS and the drawn conclusions.

1.1 OSS Compliance

As [3] points out, OSS compliance can be furthered by incorporating "compliance process and policies, checkpoints and activities as part of existing software development processes". This includes activities to monitor what OSS is being used in which software projects. A basic and direct approach would be to perform these activities manually. However, manually tracking all used OSS, the accompanying OSSL as well as the results of e.g. performed license terms analysis is very time consuming and error-prone. In [1], we discussed OSS tools, such as FOSSology or Open Source License Checker, that can help tracking the details of used OSS, and presented a process for managing license texts when introducing new OSS to a software project in such a way that said texts can be prepared automatically when a software release build is compiled.
When a software product is delivered to a customer, it is usually required to include a complete list of all the OSS and OSSL relevant to the software. Manually creating and updating this list is also very time consuming and error-prone. Some OSSL demand that certain conditions are to be met when providing software that contains OSS licensed under them. This includes for instance providing a disclaimer along with the OSS, providing the source code of the OSS or a special acknowledgement text. Complying with these conditions without automated assistance can be seen as time consuming and error-prone as well.

1.2 Hurdles in Practice

Ensuring OSS compliance is not only relevant with regard to license conditions of directly used OSS, but also covers those OSS, which directly used OSS depends on. Such dependencies are called transitive dependencies. One basic problem here is to find out whether a directly used OSS itself uses other OSS and what OSS exactly is being used transitively. Depending on the structure of a software project, the toolchain used to build the software might already contain tools to help solving this problem. Direct dependencies and transitive dependencies may e.g. be visually represented as a tree with the help of the toolchain. In Maven-based (Java) projects, such a tree can be computed using a plugin that can process the dependency information of the project configuration, e.g., the so-called maven-dependency-plugin. The applicable OSSL have to be determined for each node in such a dependency tree. This might prove to be problematic in some cases. One such case is given when contradicting information about the license is given, scattered e.g. across

different parts of the OSS documentation. For instance, the deployable binary of an OSS might contain a reference to a license ‘A’, but the project website for the OSS lists another license ‘B’. A lot of investigational effort might be necessary to resolve such contradictions. For such cases, in [1] we devised an approach based on a prioritized list of possible sources to check for license information: the binary of the OSS, the project website, the source code, the project file (i.e., the project configuration file for Maven projects). In some cases, it might even be impossible to resolve this contradiction at all, e.g., for OSS that is not actively maintained anymore. Additionally, if no particular license information can be found in any of these sources, our approach prohibits using the OSS at all, as the risk of an accidental license infringement is considered as being too high.

Within our own software projects, we identified several libraries where it turned out to be rather difficult to determine their actual license in the way described above. E.g., in one case, not only had the maintainer changed due to the original maintaining company having been bought by another company, but also was the OSS not actively maintained anymore and several of its components were now scattered across various source code repositories. In addition, the state of these repositories was not well documented, so the actual version of the source code in each repository was unclear. For cases such as this, a substantial degree of analysis might be required to unravel this issue.

This remainder of this paper is structured as follows. Sec. 2 addresses already existing approaches for open source compliance. In Sec. 3, we describe our approach using a ubiquitous domain language and how we intend to automate the generation of OSS lists. In Sec. 4, we describe how such an implementation can be realized in practice using the OSS tools Maven and Nexus [4] Repository. Finally, in Sec. 5, we give a conclusion and indicate topics for further work.

2. STATE OF THE ART

There are various approaches towards a best practice for reaching OSS compliance. One example is the guideline that “Germany's digital association” Bitkom released [5]. It gives an overview of several aspects of OSS and OSS compliance with a focus on license management and traceability of license interpretation. This guideline lacks both the comprehensive overview and detailed steps; therefore, we estimate it to be cumbersome to utilize.

On the other hand, [3] describes an end-to-end compliance process that components containing OSS have to undergo before receiving approval for distribution, using the term “compliance due diligence process”. This end-to-end compliance process consists of ten steps and according to the document itself, “focuses on practical aspects of creating and maintaining an open source compliance program”. The first step addresses the identification of OSS. This can be done by several methods, e.g., engineering staff informs the OSS compliance team of the intent to use specific OSS in a specific product or a product as a whole is audited to establish a compliance baseline. The second step consists of scanning the source code to discover matches with known OSS projects by using automated analysis tools. The main outcome of this step is a report identifying the origins and licenses of the OSS source code. Any issues identified during the audit have to be resolved in the third step by the appropriate engineering team. In the fourth step, the interactions between the OSS and proprietary code is analyzed to find out whether certain licensing obligations extend from the OSS components to the proprietary product. When all reviews have been completed, usage of the specific component can be approved (step 5) by the appropriate entity. After a software component has been approved for usage in a product, the component is added to the software inventory (step 6) that tracks OSS. Step 7 describes one of the key obligations of the compliance process, the documentation obligation. According to [3], that means that a product using OSS when distributed has to be accompanied by:

- Information for the end user about how a copy of the OSS source code can be obtained
- Acknowledgement statements about the used OSS
- All license agreements for the OSS included in the product

Steps 8, 9 and 10 are dealing with distribution strategy of OSS source code and proper implementation of the strategy.

The briefly described compliance process [3] is primarily aimed at enterprises where larger teams and multiple software components need to be managed. For this reason, the individual steps are quite strongly formalized. For smaller teams such as in our own environment, these steps need to be adapted and refined. In our previous paper [1], we described how we implemented a similar process after adapting it to the scope of a scientific research institution with small development teams. In this paper, we focus on management of OSSL information and automated generation of OSS notices.

3. OUR APPROACH

The proposed approach in the following consists of establishing a common language by defining a domain model and an automatic process based on the domain model.

3.1 Establishing a Common Language

As we delved into the details of OSS compliance, we found that there are often misunderstandings between technical and domain experts, as certain terms have different meanings in different fields. We observed that experts were addressing the same aspects using different terms, such as “third party library”, “open source library” and “dependency”. Furthermore, in several instances the same term was used with different meanings by involved experts. For example, one expert used the term “license” to designate the license text accompanying a third party library, whereas another expert used the term to speak about a certain type of license (e.g., “Apache software license 2.0” or “GNU Public License 2.0”). Such ambiguity caused communication and common documentation to be confusing and/or misleading. To avoid these linguistic tripping hazards and to decrease their potential risk of license infringements, it was necessary to develop a common, ubiquitous language.

We used a Domain-Driven Design (DDD) approach as described in [6] in order to both collect and structure the collected or devised information. Starting with a “knowledge

crunching" session with representatives of the development team, participants having a focus on a legal perspective as well as participants having a focus on the technical development expressed their points of view to the other participants. From this session, the participants were able to derive common concepts, a common terminology, as well as the boundaries and connections between their points of view. These findings were documented and presented to the entire team, and it was agreed to use them as the common, ubiquitous language for future work. The domain model that resulted from this work is described in the following section.

3.2 The Domain Model

The domain model that describes the ubiquitous language consists of several so-called bounded contexts: the "third party library" context, the "license information" context and the "license type" context. Combining these contexts defines our entire problem scope and gives a consistent description of the applicable scopes.

3.2.1 Overview: The main elements of the domain model are the "third party libraries" element, "license types" element, "license information" element and the "aggregated license text" element.

The "third party library" element describes the actual software binary that is usually accompanied by its source and documentation when distributed by the original vendor.

The "license type" element describes the type of OSSL, e.g., "Apache Software License 2.0", under which the vendor provides his library. As a vendor can provide various parts of the library under diverging license types, a 1:1 mapping between the "third party library" element and a single "license type" element in general does not suffice. Instead, there needs to be a 1:N mapping.

To link the "third party library" element with the "license type" elements it is provided under, the "license information" element is used. The "license information" element contains further information such as traceability rationale, internal remarks and the references to the original location or source that indicated the license type.

The aggregated license text is a technically required artifact and represents the license texts of all the license types under which a third party library is provided.

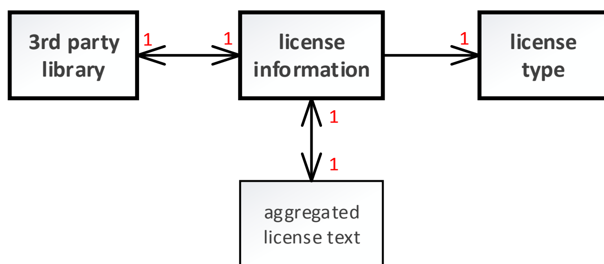

Figure 1 – license information overview

3.2.2 License Type: The license type acts as an identifier and embodies the requirements imposed by the vendor when for using a library, as well as the obligations that the software developers have to comply with when using the library in a software project.

Examples for obligations include:

- Providing a disclaimer
- Providing the source code of the library
- Withholding the source code of the library
- Performing no modification to the library

3.2.3 License Information: A variety of information is kept in the "license information" element. One part is the linkage between the "third party library" element and its "license types" element. Other parts are the traceability rationale that allows other team members to verify the information in the "license information" element and "internal remarks" element manually.

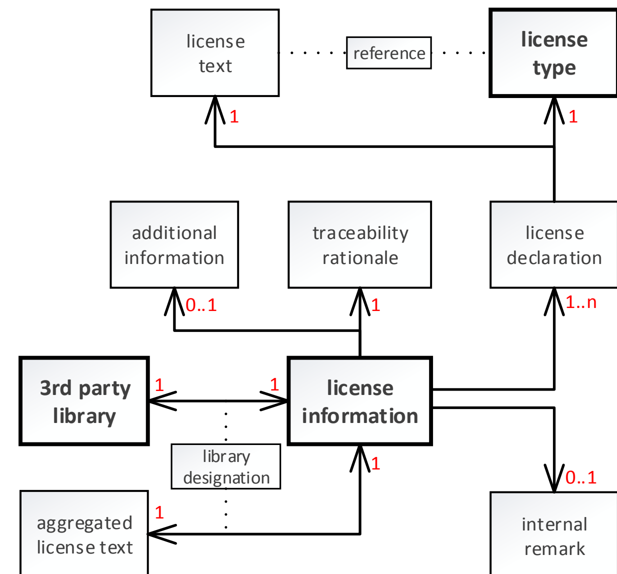

Figure 2 – license information details

3.3 Design of the Automation Process

Using the agreed-upon domain model and terminology enabled us to build upon the results described in our previous publication [1]. Furthermore it allowed us to develop a process for generating the desired list of used OSS that involves the previously established repository structure for storing license texts.

In our environment, we use mainly Java for software development. Our software development projects are managed by Maven. Artifacts such as third party dependencies and OSS license texts are stored in the Nexus repository. When releasing software, relevant artifacts such as OSS source code and license texts are pulled from the Nexus repository and provided together with our products. In our opinion, the following design should be transferable to other programming languages that use other artifact repositories.

With the data model as the basis and the OSS documentation as the goal, we more precisely specified the additional information about every used OSS that needs to be stored in order to generate documentation about the used OSS for customers as well as for internal audits. As we already store

the license texts in a repository, it became apparent to re-use the same structure and tools to store this additional information in the same way. For our software development environment, this means using a software build repository that can be accessed by Maven projects. In similar to the license texts themselves being stored in parallel to the OSS using the same Maven Group, Artifact and Version (GAV) description as the OSS and the suffix "license" as a so-called artifact classifier, we settled on using a separate JSON file with an additional classifier to hold the additional license information that we want to handle. Re-using the GAV information of the OSS allows associating the additional information, the OSS and the license file.

4. IMPLEMENTATION

Based on the approach described in the previous section we developed a Maven plugin named osslist-maven-plugin (OMP). The plugin takes relevant information from the Maven repository during the software build process and creates two lists of used OSS software: one for customer deployment and one for internal usage.

All the aspects described in section 3 are addressed by within the OMP. The list for internal usage contains all documented and stored OSS information about each OSS, whereas the list intended for the customer contains a filtered view, without internal notes, such as the source of the license text. The OMP itself uses the maven-dependency-plugin to resolve the dependency tree of a project. In order to dismiss internal and third party dependencies that have not been released under OSSL, a filter functionality following Maven best practices has been implemented.

The OMP makes use of the existing Maven project environment and infrastructure, such as Maven repositories [7], which we were already using for developing our software. The OMP was developed in accordance with the Maven plugin developer guide [8].

4.1 Preconditions

A Maven project is defined by a project object model (POM) [9]. To make use of the OMP and its functionality in a software project, the OMP has to be included into the POM of the project, enabling it to either be called as a separate goal or integrated into the so-called lifecycle phase of Maven projects using the execution-tag. The OMP may be configured in more detail in the POM. In addition, it provides a default configuration so that it can also be executed without needing any explicit configuration. The available configuration options are described below.

4.2 Functionality

In this sub-section, the three main functionalities of the OMP are described. The first functionality is to resolve the transitive dependencies of the Maven project in which the OMP was executed and to collect necessary artifacts of the retrieved dependencies from the repository. The second functionality is to check and ensure that certain licensing constraints given by the OSSL of a direct or a transitive OSS are being complied with. The third functionality is to create the described lists of used OSS. In the following, these three functionalities are explained in more detail.

4.2.1 Handling license information: Initially the OMP resolves all the transitive dependencies given by the POM of the Maven project in question, but ignoring dependencies with scope 'test' and 'provided', by using the maven-dependency-tree library and a scope artifact filter. Note that the information about a dependency is stored as a Maven artifact inside a dependency node, which represents the dependency as a graph. After having created a graph of all transitive dependencies, a pattern artifact filter is applied for all dependencies for which no license information is needed, e.g. internal libraries. This filter hides all dependencies in the representation of the graph by matching their fully qualified artifact name with the help of regular expressions.

The patterns of these regular expressions can be configured via the plugin definition in the POM, between the ignoreArtifact-tags. Note that dependencies of a hidden dependency are still visible, which means that the pattern artifact filter does not filter transitive dependencies. This allows e.g. filtering dependencies for which no license information is needed, such as internal libraries.

For each dependency remaining after filtering the graph, exactly three artifacts are downloaded from the Maven repositories. These are the OSS dependency used by the project itself, e.g., the jar, the license text artifact containing the text of the license and finally the license information artifact containing additional information about the license.

Maven classifiers [9] are used to distinguish between the various components of the same artifact information, such as a binary and a source component. Here, the additional artifact components are the license text file and the license information file. The classifiers are 'license' for the license text file and 'licenseInformation' for the license information file.

The JSON format is used with a predefined structure for specifying license information in a license information file. The root of the JSON structure is an array of license specification objects. These license specification objects specify the name of a license type, the source of a license type and the type of a license text source. The last three entries are free text fields. The first field provides additional information, such as disclaimers and acknowledgements incurred in the licensing of a dependent artifact. The second field holds the traceability justification, which contains descriptions of where information about the license type and license text was found. The last field is used for internal notes and can be considered optional.

```
{
  "licenseInformation": [
  {
    "licenseTypeName": "JDOM License",
    "licenseTypeSource":
"http://www.jdom.org/docs/faq.html#a0030
(accessed 13.12.2017)",
    "licenseTextSource": "META-
INF/LICENSE.txt within binary jar"
  }],
  "additionalInformation": "This
software component uses this Open Source
Software developed by the JDOM Project
(http://www.jdom.org/)",
```

```
  "traceabilityRationale": "see
licenseTypeSource and META-
INF/LICENSE.txt",
  "internalInformation": "This is an
example for internal information"
}
```

Listing 1: Example of a JSON file.

The JSON schema enforces some of the entries. The "license type" name and source, as well as traceability rationale of an OSS must be given. If the OMP cannot find these mandatory entries, then its execution will be aborted.
Note that the OSS and the license information file must be available in a Maven repository; otherwise the OMP also aborts its execution.
Some OSSL contain clauses that make it necessary to include a pre-formulated phrase of acknowledgement. We devised a routine that checks the license information JSON files containing certain OSSL types for also containing an appropriate phrase as it is required e.g. by the "Indiana University Extreme! Lab Software License 1.1.1" [10]. This functionality has also been included directly into the OMP for now. It is planned to move this functionality to a server-side plugin of the Maven repository.

4.2.2 Checking and ensuring licensing constraints: In the second stage the OMP checks rules that apply to license types. If the rules are not satisfied, a message describing the violation is printed out and the build process is aborted. Some types of licenses have licensing constraints such as not handing out the source code.
To achieve enforcing the rules, the OMP retrieves all used license types based on the license information artifacts of all used OSS. This list is then checked against an internal set of rules for violations.
The OMP is extensible with regard to new license types and their applicable rules.

4.2.3 Creating list of used OSS: In the last stage, the OMP creates documents based on the three artifacts downloaded for each dependency. These documents are a Comma Separated Value (CSV) file and an Office Open XML (.xlsx) file, both containing the same information. The latter is more user friendly as it provides a customized representation.
The first column holds the Maven group id, the Maven artifact id and the version of a dependency. The second column provides an enumerated list of all license types. The third column specifies the file that contains the license text of the license types given in the second column. The fourth column contains the additional information.
Furthermore, there are four optional columns, which are employed for internal use only and may be enabled with the enableInternalNotes-tag. These columns cover the internal note, the traceability rationale and an enumerated list for the license type source and license text source.
It is worth noting that the OMP should be set in the project configuration to be executed before the assembly-maven-plugin. This way the generated license overview can be added to the deployable release ZIP as part of the execution of the assembly-maven-plugin.

As a side note, during the development of the OMP, we encountered an issue arising from incompatibilities between the used software dependencies. Maven uses the eclipse aether API [11] for repository tasks since version 3.1.0. As the OMP also uses the eclipse aether API, it can only be used with Maven versions 3.1.0 or higher. This issue was resolved on our side by making the decision to upgrade all development systems to a common lowest version of Maven.

5. CONCLUSION AND FURTHER WORK

5.1 Conclusion

In the beginning of our work described in this paper, we were confronted with the manual task of creating documentation of used OSS for software releases in an otherwise mostly automated process. Due to our previous work with automating aspects of OSS compliance, we had a foundation upon which we were able to develop a management process for license information and tool-supported automation of the document creation. This solution provides the following improvements and benefits.

5.1.1 Better understanding: Before the implementation of the measures described in this paper, the communication between team members in dealing with OSS repeatedly resulted in misunderstandings. These misunderstandings could be resolved with the help of the developed domain model. The general understanding in the team regarding OSS could be improved in this way. In addition, new employees can now become familiar with the material more quickly.

5.1.2 Higher efficiency: Previously, every release of each software product required the manual creation and maintenance of a list of used OSS. This meant that the entries of the table had to be checked against the dependency tree. With up to 100 OSS components used per product, this required a considerable amount of time, because e.g. by adding a new OSS to the product, numerous transitive libraries had to be checked and manually entered into the table. Checking the OSS and documenting the necessary information is still done. However, this now has to be done only once when the OSS is integrated into the software project instead of every time during the release process. Especially for patch releases that have to be delivered on short notice, the new approach brings significant time advantages.

5.1.3 Better reusability: If a specific version of OSS is checked when adding it to a software project for the first time and the results are stored in the Maven repository, they are available to all other projects from then on. Multiple redundant analysis of the OSS is prevented in this way, which leads to a uniform basis of information across all our projects.

5.1.4 Better traceability: At the time an OSS is introduced to a project, finding the license information is usually not a problem as the OSS is usually current and the required information is maintained and readily available. The situation is different with legacy OSS that was integrated in the past. Understanding licensing entries for an OSS that are somewhat older can prove to be extremely difficult, in rare

cases even impossible. Introducing the mandatory field "traceability rationale" used for internal use and the optional field "internal information“ no longer causes these problems and provide helpful information for audits.

5.1.5 Less error-prone: The previous manual creation and maintenance of the OSS lists did not prevent mistakes, e.g. an OSS that was no longer used had been overlooked and was not deleted from the OSS list, or it could be overlooked that the version of an already used and listed OSS has changed. These errors can no longer occur using the described approach, since OMP handles the information about the OSS based on the project configuration. It also informs the developer about any missing OSS information and aborts the build process. Furthermore, it detects violations of rules defined for license types.

5.2 Further Work

The existing solution is in active use by developers and fulfills our immediate needs. In the future, we want to develop additional processes and tools to automate the OSS compliance process more and we also want to improve the usability for the developers dealing with OSS and the associated OSSL information.

5.2.1 Improve OSS compliance automation: The following points have been planned to be addressed in the future in order to reach a more robust OSS compliance:

- We want to introduce the usage of tools for analyzing the source code of OSS, so that embedded and shaded transitive OSS that has not been declared can be detected.
- It is planned to use a “bill of material difference tool” (BOM diff tool). Given the BOM for a product version 1.1 and the BOM for the previous version, e.g. 1.0, the tool computes the delta and present new OSS added or retired in version 1.0. [12]
- The OSS Compliance process that we currently use for OSS libraries shall be adapted to be used for other free resources such as XML schemas or icon and theme files as well.

5.2.2 Improve User Experience: We want to assist the development team more in following the OSS compliance process. To achieve this, we envision an OSS Compliance Manager (OSSCM) tool that could contain the following functionalities:

- Currently both the files that contain the license texts and the JSON files need to be created with a text editor and uploaded to the Maven repository by hand. If there is an error found in one of these files, the file has to be downloaded, deleted in the Maven repository, corrected and uploaded again to the Maven repository. This is also has to be done on all local Maven repositories, e.g. on developer systems. The OSSCM could simplify these activities by providing a GUI to the user for creating and editing the license texts and license information, including synchronization with the Maven repository.
- A module of the OSSCM could display an overview of used OSS of a software development project, focusing on missing license texts, license information and source code.

6. REFERENCES

[1] S. Dyck, D. Haferkorn, J. Sander, An organizational-technical concept to deal with open source software license terms, Proceedings of the 20th World Multi-Conference on Systemics, Cybernetics and Informatics, WMSCI 2016, July 5 - 8, 2016, Orlando, Florida, USA, http://publica. fraunhofer.de/dokumente/N-417588.html, (date accessed: December 1, 2017)

[2] Apache Maven, https://maven.apache.org/, (date accessed: February 13, 2018)

[3] I. Haddad, PhD, Open Source Compliance in the Enterprise, The Linux Foundation, 2016, http://www.ibrahimatlinux. com/uploads/6/3/9/7/6397792/open_source_compliance_in_the_enterprise_2016.pdf (date accessed February 6, 2018)

[4] Sonartype Nexus, https://www.sonatype.com/nexus-repository-sonatype, (date accessed February 6, 2018)

[5] Open-Source-Software 2.0. Leitfaden, Bitkom e. V, https://www.bitkom.org/noindex/Publikationen/2016/Leitfaden/Open-Source-Software-20/FirstSpirit-1498131485664160229-OSS-Open-Source-Software.pdf (date accessed February 6, 2018)

[6] E. J. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Pearson Professional, 2003

[7] Apache Maven – Introduction to Repositories, https://maven.apache.org/guides/introduction/ introduction-to-repositories.html (date accessed February 6, 2018)

[8] Apache Maven – Plugin Developers Centre, https://maven.apache.org/plugin-developers/index.html (date accessed: February 6, 2018)

[9] Apache Maven – POM Reference, https://maven.apache.org/pom.html (date accessed: February 6, 2018)

[10] Oracle® VM – Licensing Information User Manual for Oracle VM Manager Release 3.4, https://docs.oracle.com/cd/E90714_01/E64079/html/vmlic-license-indiana.html (date accessed: February 6, 2018)

[11] Using Aether in Maven Plugins, https://wiki.eclipse.org/Aether/Using_Aether_in_Maven_Plugins (date accessed: February 6, 2018)

[12] The Linux Foundation, Using Open Source Code, https://www.linuxfoundation.org/using-open-source-code/ (date accessed: February 6, 2018)