Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

GREGOR FENDT
BACHELOR THESIS

Bill of Materials Generation and Tracking

Submitted on 5 November 2018

Supervisors: Andreas Bauer, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

Erlangen, 5 November 2018

License

_____

Erlangen, 5 November 2018

Abstract

Software projects are growing and are reusing open source components more often. Reusing components saves development costs and grants other general benefits by using open source software. In order to get an overview of the code component architecture of software projects, the Professorship for Open Source Software created a tool. The tool extracts component and license information from build artifacts. The tool generates a model, which can be used to get an overview of all used components. Additionally, different processes can be applied to it, for example to check for license compliance or security vulnerabilities. Another important use case is the creation of Software Bill of Materials (BoM) artifacts. The BoM describes the components and licenses in a product. It is used to communicate component information throughout the software supply chain. Therefore this thesis focuses on developing a solution to automate the generation and tracking of such BoM artifacts.

Contents

1 Introduction

1.1 Motivation

Free/Libre and Open Source Software (FLOSS) has risen in importance over the years. The general image of FLOSS shifted from being a hindrance to the software industry, by destroying existing revenue models, to being a way of saving development costs and creating new business models. In 1998, the Open Source Initiative (OSI) was founded by Eric Raymond and Bruce Perens. As a non-profit organization it advocates for the benefits of FLOSS ("History of the OSI | Open Source Initiative," October 18). Due to the rise of importance and the significant difference in marketing strategies, open source was at first regarded as a threat to the classic commercial software vendors. In 2001, a Microsoft executive stated: „Open Source is an intellectual property destroyer. I can't imagine something that could be worse than this for the software business and the intellectual-property business.". Eventually, Microsoft changed its mind. In 2018 they even acquired GitHub, the biggest platform to publish open source code, for US$7.5 billion. Microsoft is not the only major software company involved with FLOSS. Others like Google, IBM and Oracle are also heavily invested in open source. For instance, all of them pay US$500 thousand annually for a platinum membership to the Linux Foundation, earning them a seat at the board of directors.

The rising value of FLOSS is not only seen in big corporate entities. In general the additions to open source projects, the total project size of open and the total amount of open source projects are all growing at an exponential rate (Deshpande & Riehle, 2008). Open source software isn't only growing steadily, but it is also used in a wide range of diverse applications. Some examples are Linux and most of its Distributions, which offer an alternative to proprietary operating systems, the Apache HTTP Server, which is estimated to be the most used web server over the whole internet, user applications like Mozilla Firefox and OpenOffice and interpreters/compilers like Python and the GNU Compiler Collection. All of them are successful open source applications that are used by millions of users including businesses. Established revenue models got shaken up by the success of the open source business model.

The rise of open source can be contributed to multiple advantages it offers. Enterprises extend the range of their product by giving the public access, which consequently leads to gaining access to a broader pool of users. Coupled with the open availability of the source code, they can function as co-developers. Eric S. Raymond outlines his opinion about FLOSS in the essay "The Cathedral & the Bazaar". He uses the metaphor of a bazaar to describe the development model of open source. In his opinion, the open availability of source code can help to create "co-developers" out of the user base. This "bazaar style" can lead to a more community-driven dynamic software development, which is distinguished by faster diagnosis of bugs and vulnerabilities, creating new features, suggesting fixes and improving the code (Raymond, 1999). A EU commissioned report supports his opinions by declaring that open source helps to reduce project failure and lower the cost of code maintenance (European Commission, 2017).
One of the biggest benefits of open source is the possibility of reusing entire existing open source components to construct a new software project. Abstaining from writing a component from scratch, enables developers to integrate functionality more quickly, while reducing the cost by saving time (Haefliger, Krogh, & Spaeth, 2008). Besides cost savings, reuse improves software quality and maintainability, by relying on proven components (Frakes & Kang, 2005). The already existing components can offer the aforementioned advantages with whom own solutions often cannot compete. Carlo Daffara, a researcher in the field of IT economics, who contributes to research projects by the European Commission, estimates the direct savings of the utilization of open source software for the European economy to be around EUR114billion per year. Furthermore, he rates that about 35 percent of the used software in the past five years was derived from FLOSS, making open source "[…] not a marginal contribution to the European economy" (European Commission, 2017). In today's commercial world the reuse of components is so prevalent that many software companies would not exist without it.

On the other side, there are new challenges to face with the reuse of FLOSS components. It is crucial for a developer to overview all the different code components he uses in his software projects, whether it is for rebuilding a specific version of the software, efficient detection of known vulnerabilities or to uncover any license inconsistencies.
When a new vulnerability in a software component gets reported it is important to immediately identify and resolve the issue before hackers can abuse it. Software products could always have exploits and projects using these products make themselves vulnerable as well. One of many examples became known in 2014, where a bug was found in the OpenSSL library used by many online services, applications and even operating systems. Confidential data like passwords could be extracted from these software products. The bug was named Heartbleed.
Most importantly open source software cannot exist without a license. Due to copyright law, any creative work with no license only grants the exclusive rights to the creator. As long as a software product does not have a license, nobody is allowed to modify, use or even copy it. In turn this means that every open source product must have a license to grant third parties certain rights. Every software publisher could create their own specific license for their product, but this is uncommon in open source. This would introduce legal complexities the common software developer does

not want to deal with, which has the consequence that this particular software would rarely get used, diminishing the positive factors of open source. Instead, there is a certain amount of standard open source licenses that are commonly used. The Open Source Initiative has only approved around 90 open source licenses. OSI reviewed these licenses and they comply with the Open Source Definition, which allows software to be freely used, modified, and shared ("Licenses & Standards | Open Source Initiative," n.d.b). Consequently, the effect of standardizing the licensing can be seen in big software projects, that reuse a lot of components, as they do not have a huge number of unique licenses. Nevertheless, not all open source licenses are compatible with each other. For example, the MIT License grants the user the right of commercially distributing the source code without disclosing their own project as long as they add a license and copyright notice. Whereas the GNU General Public License v3.0 (GNU GPLv3) requires the licensee to publish their software under GNU GPLv3 as well. This means no project licensed under the MIT License can use GNU GPLv3 software components as the GPL is more restrictive and must get propagated upstream. When reusing any open source component, one must comply with the terms of a license, which means that a project made up of multiple components must be checked for their compatibility to guarantee license compliance.

The Product Model is a tool developed at the professorship for open source software at the Friedrich-Alexander University Erlangen-Nürnberg. The eponymous data model, which the tool creates, displays the license and component information of an analyzed software product. At the moment it obtains the component architecture of a software project from build scripts and source files. By creating a generic model, it enables developers to get a greater understanding of the component architecture and it is possible to apply algorithms and other processes to the model or parts of its data. Common tools would be ones that check for license compliance, detect any non-compatible licenses or check for known vulnerabilities under the components. However, this thesis focuses on another use case the creation of an artifact that is called the Software Bill of Materials.

1.2 Bill of Materials (BoM)

Traditionally the BoM is used in manufacturing, it specifies how a main product is built up from subcomponents. These components can be any kind of parts, materials or assemblies that again can have their own BoM. Out of these relationships a tree data structure can be formed down to the leaves, which are components that don't have a BoM and thus have no relevant subcomponents. When the BoM is well-kept, it helps companies establish accurate records of the used components. It is helping to rebuild products, because it is easier to estimate material costs, manage the inventory and purchase the essential parts. They can be found in different parts of a business, throughout the supply chain in design, engineering and production departments. Different kinds of BoMs exist that are partly unique to specific business sectors and they are all differently designed, depending on the topic. As the Manufacturing BoM focuses on which assemblies, parts and materials are needed, the Engineering BoM specifies technical details of the design and the Sales BoM interprets all components as sale items.

Analogous a Software BoM can be designed to keep track of all the software components used in a software project. The Software BoM is structured similarly to a regular BoM, inasmuch a software product has root components which itself can have subcomponents and so on. As the amount and size of open source projects are continuously growing (Deshpande & Riehle, 2008) and on average around 30% of added functionality in FLOSS projects is reused code (Sojer & Henkel, 2010), it is important to have an overview of all these components. It is not solely interesting for parties directly involved with the project but the board of directors, the legal team and potential acquirers or investors want precise information about the product structure as well. The Software BoM can not only be used to battle the above-named problems of license compliance and component security vulnerabilities but also to ease the communication between parties in the software supply chain. For example, as Commercial software products are reusing FLOSS components too, the purchasers are also interested in shedding light on potential licensing deficiencies. Though buyers may have potential preferences to how the BoM is structured and which information is included, this would force suppliers to create specific BoMs for each request.

To facilitate the communication process and get rid of redundant work a standard for BoMs is necessary. A standard not only communicates information in a precise and uniform manner, but it also allows another processes and tool to be built upon. Which is helpful to reduce the initial workload for developers to create a BoM as they don't need to create their own model and can utilize existing supporting software. This thesis will take a closer look at the Software Package Data Exchange (SPDX) standard. SPDX may be the most known standard to communicate Software BoM information.

Since the focus of this paper lies on the Software BoM, from now on it will be solely referred to as BoM.

1.3 Thesis Structure

This Thesis focuses on the automatic generation and tracking of BoM artifacts. The goal is to create BoM out of the data extracted by the Product Model tool
At first, the basic structure of the Product model tool will get explained in the 2nd chapter.
Because the BoM artifacts are created as SPDX files, the 3rd chapter is an introduction to the SPDX standard.
The 4th chapter lists all necessary requirements the thesis needs to fulfill.
In the 5th chapter implementation details are discussed to clarify how the product model helps in generating SPDX files.
If the requirements of this thesis got fulfilled will be handled in the 6th chapter.
At the end, the 7th chapter gives a conclusion of the whole thesis and suggests some improvements.

2 The Product Model

The generation of the BoM is based on the Product Model tool. It was created at the Professorship for Open Source Software at the Friedrich-Alexander University Erlangen-Nürnberg, to present the component architecture of software projects. Thus, it is a collective project of the research group. Another thesis was already based on this tool. Dennis Scheffer developed the maven crawler that fills the data model with information. In his master thesis can be found more detailed information, especially about the design and functioning of the maven-based crawler (2018). In this chapter I will give a brief introduction to the tool and its data model structure. The exact crawler implementation is not relevant to handle the model, because it does not matter which kind of crawler is used to fill it with information.

2.1 Introduction

The Product Model received its name from its goal, which is creating a generic serializable component *model* out of a software *product*. The definitions of software component and component model which are used in this thesis got specified from Councill and Heineman (2001) and shall be as follows:

> *"A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."*

> *"A component model defines specific interaction and composition standard. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model."*

Not only the tool is called the Product model, but the created component model also bears the same name. The project contains all other supportive processes that fill or

process the data model. Whereas, the component model is a representation of the extracted data, resulting from analyzing software artifacts. This generic and serializable model can be used to build other tools or processes upon it (e.g. license compliance tools, vulnerability checks).The Product model is intended to be expandable to analyze all kinds of software projects. Yet the current version is only able to analyze projects created with the build tool Maven, therefore it is also limited to Java. The model can be exported as a JSON, XML or YAML file.

2.2 Structure of the product model

The parent class of the model is the *Product* representing the main software project. It is holding information like which build tool got used and the version control system but most importantly information about all the declared licenses and the root components of the project. A *Product* can have none to multiple root components, these *Components* and their subcomponents make up the tree-like structure of a component model.
A *Component* can have multiple dependencies, relating them to other *Components*, a List of *MetaData* and only one *Artifact*.
Dependencies are *Relationships*, they declare how *Components* are interconnected with each other. For this reason, they have a source and target *Component* and a *RelationshipType*. Right now, the only *RelationshipTypes* are how the *Components* got linked together, either dynamically or statically. However, as the only applicable programming language is Java and Java doesn't link any library references while compiling, *Components* can never get statically linked together.
An *Artifact* is the actual file associated with a *Component*, so there can only be one per *Component*.

Each *Component* has an arbitrary amount of *MetaData.* At the Moment, only the *InterfaceData* and *LicenseData* classes implement the *MetaData* interface. *LicenseData's* only information is a String declaring a found license and *InterfaceData* holds information about an interface, the component offers to another component, in the *InterfaceInfo* class. An UML diagram displaying the associations between the classes of the product model classes can be seen in Figure 1.

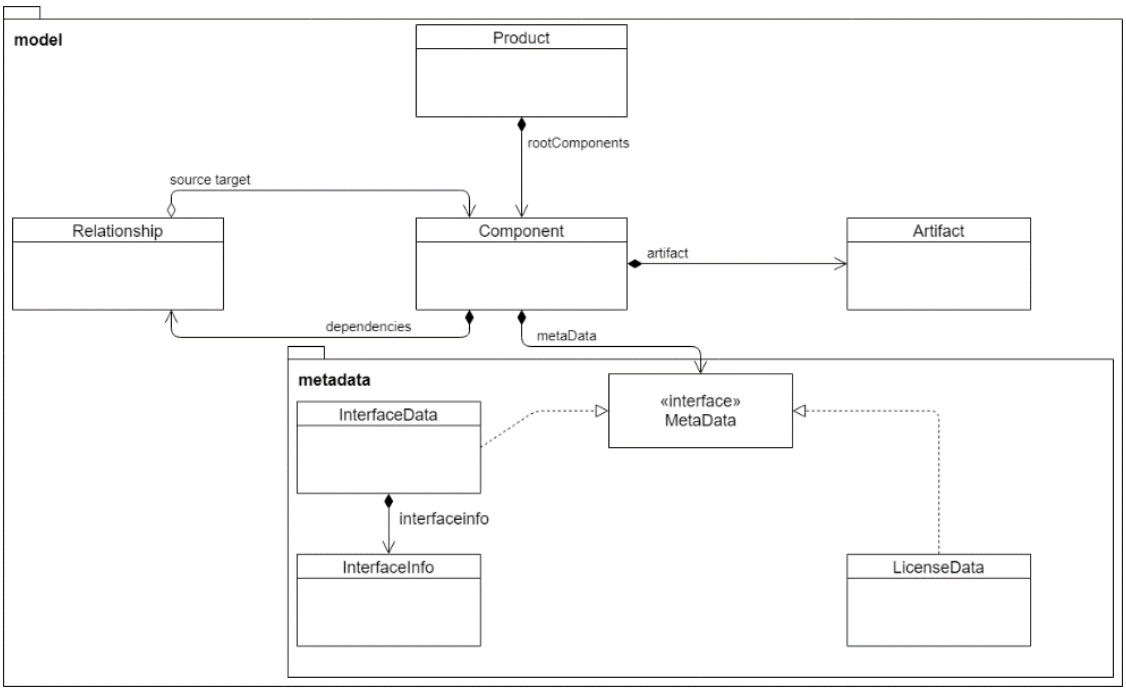


Figure 1: Structure of the product model

3 Introduction to Software Package Data Exchange (SPDX)

Since I create a SPDX artifact to communicate the BoM information, this chapter gives an overview about the SPDX standard. Firstly, a definition of SPDX is given then I explain what kind of different ways there are to use SPDX. The 3rd subsection focuses on the structure of SPDX documents, since this is the type of SPDX artifact which gets created. Lastly an overview over the tools that can be used on SPDX artifacts is given.

3.1 Introduction

SPDX[1] is one of the key pillars of the Open Compliance Program of the Linux Foundation, it might be the most used standard to communicate software BoM information (components, licenses and copyrights of software packages). The firsts draft of the specification began in 2010 by the SPDX workgroup under the Linux Foundation and in 2016 the latest iteration, version 2.1, got released. The workgroup consists of individuals, different groups and companies which are interested in advancing a standard that is aimed at creating complete re-usable license information of software components. There are 3 SPDX teams, each one associated either with the technical, business or legal responsibilities that exist. By standardizing the way how to communicate BoM information throughout the supply chain, they try to facilitate the exchange of license and other policy compliance. Their vision is, for every party in the chain to declare their package information reliably, so no information must get analyzed or documented twice. Additionally, having a uniform format helps to communicate precisely from the software developer to the end-user, while not forcing every party to create their own tools and processes to deal with BoM information. These factors contribute to big companies engaging with SPDX, including Siemens, Wind River, ARM or Intel ("About | Software Package Data Exchange (SPDX)," n.d.a).

[1] https://spdx.org/

3.2 Types of Usage

There are three different mutually exclusive ways of utilizing SPDX, all of them are made to capture license information about the software and present them in a human- and machine-readable format ("Using SPDX | Software Package Data Exchange (SPDX)," n.d.e).

3.2.1 License Identifier

The simplest way to use SPDX, is to add License Identifiers to the source code (e.g. "// SPDX-License-Identifier: MIT"). Near or at the top of a file this one liner is placed, making it easy for machines to process and parse. It is a precise and simple way to represent FOSS license information. It is recommended to use them not as a replacement of present copyright or license notices, but as a supplement (Linux Foundation, 2016, pp. 100–101).
License IDs can simply just be one identifier, but if multiple licenses apply to a file the standard offers operators to form SPDX License Expressions. The "AND"- and "OR"-operator declare if multiple licenses must be complied with or one can decide which license is to be used. The "WITH" operator signals that the license is associated with an exception, declaring it differs from the standard definition, for example granting additional permissions. Finally, the "+"-Operator can declare that also later versions of a license can be used ("SPDX IDs: How to use | Software Package Data Exchange (SPDX)," n.d.c).
The REUSE initiative[2], a project by the Free Software Foundation Europe, recommends using SPDX identifiers in each license and source file ("REUSE practices - version 2.0," 14-Dec-17).
Just in 2017 the Linux Kernel added identifiers to their files to make it easier for compliance tools to determine the correct license.

3.2.2 License List

Another type of using the SPDX standard is to take advantage of the SPDX License List. SPDX offers two lists, one consisting of around 300 open source licenses [3]and the other one consisting of around 30 license exceptions[4]. The License List can be used to supplement the License Identifiers instead of the plain text. an immutable link to the list can be added. The exception list can be used for License Expressions directly linking to license exceptions. Linking to these lists, grants access to the exact license/exception text and the full name. The License List cannot only be used to exchange the identity of a license.

[2] https://reuse.software/about/
[3] https://spdx.org/licenses/
[4] https://spdx.org/licenses/exceptions-index.html

Further applications are to use the list for internal references or processes and to match found license text, per given matching guidelines and templates, to the License List ("Using SPDX License List | Software Package Data Exchange (SPDX)," n.d.g). Github checks if a project's "LICENSE" file matches a list of licenses and if it does the name and key of the corresponding license in the SPDX License List is returned ("REST API v3 | Licenses," 26-Oct-18). In companies like Siemens, MicroFocus or Wind River the license list got adopted for internal use. Moreover, are tool companies, which are dealing with license compliance, utilizing the license list (e.g. Black Duck, FOSSology, Protecode, …) ("Business Team/Adoption - SPDX Wiki," 25-Jan-2016).

3.2.3 Documents

The most detailed way is the Creation of SPDX Documents, it sums up all license information of files and packages in a project, in addition to metadata like information about the document creator or review comments. SPDX Documents are made to exchange license information of software projects, they are independent from their projects and can be exchanged isolated from the actual software product. They can carry all the necessary information a BoM needs which enables them to be a good fit for this thesis. Therefore, a more detailed description of the SPDX Documents is necessary ("Using SPDX Documents | Software Package Data Exchange (SPDX)," n.d.f).
The REUSE initiative recommends to only automatically generate BoMs,

3.3 Structure of SPDX Documents

3.3.1 Formats

SPDX Documents use 2 different main formats to describe the license information of files and their superordinate packages. The first one is the Tag/Value format it is a simple text-based format specifically created for the documents. Through the simple listing of packages and no nesting of their sub-packages, it is easier for human to get a rough overview over the used packages. Whereas the RDF/XML format got developed by the World Wide Web Consortium and therefore once converted to this data model there are already tools available to use. However, the graph structure makes it harder to single out components at the same time it already resembles the "package hierarchy" ("Using SPDX Documents | Software Package Data Exchange (SPDX)," n.d.f).

3.3.2 Contents

A SPDX document based on the 2.1 version of the specification does contain different sections. Some of those sections or the fields/attributes which they are made up of, are tagged as mandatory, whereas others are not. David Wheeler, a contributor to the specification, proposes that developers can focus on the information that is relevant for them, if they are dealing with SPDX internally. To build their document without paying much attention to the mandatory tags. Nevertheless, a huge part of SPDX is the exchange of package information, sometimes even without the described software. This is when he heavily advises to use those mandatory tags to form a consistent standard (Wheeler, 19-Jul-18). In a UML diagram of the SPDX model can be seen.

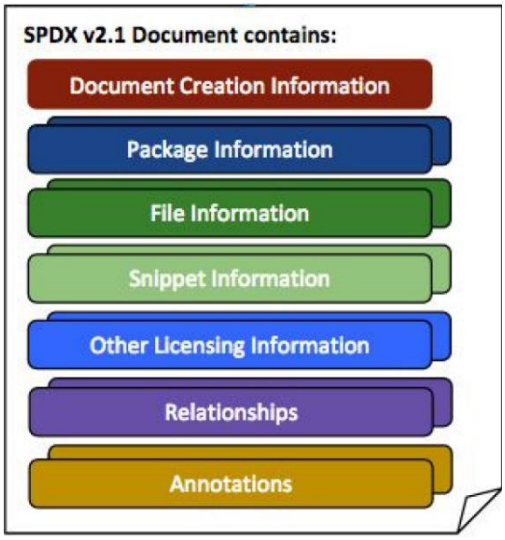


Figure 2: SPDX document sections v2.1 ("Using SPDX Documents | Software Package Data Exchange (SPDX)," n.d.f)

Document Creation Information
This section is needed once per SPDX file, to display valuable meta-information about the document creation. Some of the fields are the SPDX and license list version which provide details for the compatibility for processing tools, while fields like creator information, creation date and the creator comments add more information about the creation context.

Package Information
Package Information is required for every package that is described in the document. Packages can contain one or more files. The package section contains an SPDX id field to uniquely identify the package within the document which enables the formation of relationships and allows packages to contain other sub-packages. Amongst other things it contains the originator, the concluded license information of the package and the summary of all other licenses that are used within itself.

File Information
One instance required for every specific file in an included package. These files could be any type from license information, other documents to source code or binaries. The section resembles the package information by containing fields to represent the identification and license information.

Snippet Information
The use of Snippet Information is optional, it is used to declare when specific parts of a file got added from another original source. Such as when a file includes a part that originally got created under a different license. Besides an SPDX identifier it also has fields to denote where in the original host file the snippet information applies to.

Other Licensing Information
The Other Licensing Information lists all declared or concluded licenses of the package that cannot be found on the SPDX License List. License Information mainly exists out of the license identifier and the extracted license text.

Relationships
The relationship between two SPDX elements get described in this section. SPDX elements can be SPDX documents, packages or files. A relationship id is assembled by the source and target SPDX identifier of the two elements and a relationship type that describes how those elements are connected to each other.

Annotations
Annotations contains review information and other notable remarks about SPDX elements. They are used to convey specific information to improve reviews of SPDX documents and to make the communication of SPDX documents even more precise.

(Linux Foundation, 2016)

3.4 SPDX Tools

Owing to the SPDX standard, the development of share- and re-usable tools has been facilitated. On their website they refer to community and commercial tools, which claim to meet the SPDX specification. However, in this thesis the focus is on the tools directly built by the workgroup. The tools offer the conversion from the Tag/Value format to the RDF/XML format and vice-versa. Furthermore, they are offering other formats an RDF file can be converted to, such as a spreadsheet format where additional information can easily be added or a conversion to an html file. Other tools allow the verification of an SPDX file, which checks if all mandatory tags are filled, the comparison with one ore multiple other SPDX files or the merging of SPDX documents. The source code to these tools including a java representation of model are all available for download on github[5] ("Tools | Software Package Data Exchange (SPDX)," n.d.d).

[5] https://github.com/spdx/tools

4 Requirements

This chapter covers all the conditions that should be met by the implementation of this thesis. In chapter 0, the results will be assessed using different evaluation schemes.

4.1 RQ1: SPDX representation

The SPDX specification is a precise and standardized way to represent BoM information. The implementation should be able to create SPDX files out of the product model, which gets generated by a crawler.

Evaluation: When a SPDX file gets created by the SPDX tools, the file always gets verified at the end. The requirement is fulfilled when the creation process of the SPDX file does not return any verification errors.

4.2 RQ2: Human readable report

The files created with formats supported by the SPDX tools have a fixed design. Additionally, they always expect certain information and will display the information given to them. Even though the SPDX formats are called human readable, they can get quite unwieldy with bigger projects. With a self-made file, one can decide how and which attributes of the project are displayed.

Evaluation: Test if the implementation can generate a simple human-readable BoM artifact.

4.3 RQ3: Comparison of BoM artifacts

The intention of BoM artifacts is to represent the information of a software project. If one wanted to compare software projects, one would just need to compare their BoM. The implementation should be able to compare two BoM artifacts, displaying the differences between them. This way changes in changes when developing software can be tracked.

Evaluation: Test if the implementation creates a comparison file and validate if the created file shows component and license differences.

4.4 RQ4: Can be used in a continuous integration / continuous deployment pipeline

Agile software development is widely used to create software these days. Therefore, the implementation should be able to create BoM artifacts after every iterative step. Every change relevant to the BoM artifact should be distinguished.

Evaluation: This requirement is just depending on RQ1 and RQ3. When they are fulfilled and the implementation gets product model data, it can produce and distinguish BoM Artifacts after every iteration.

5 Implementation

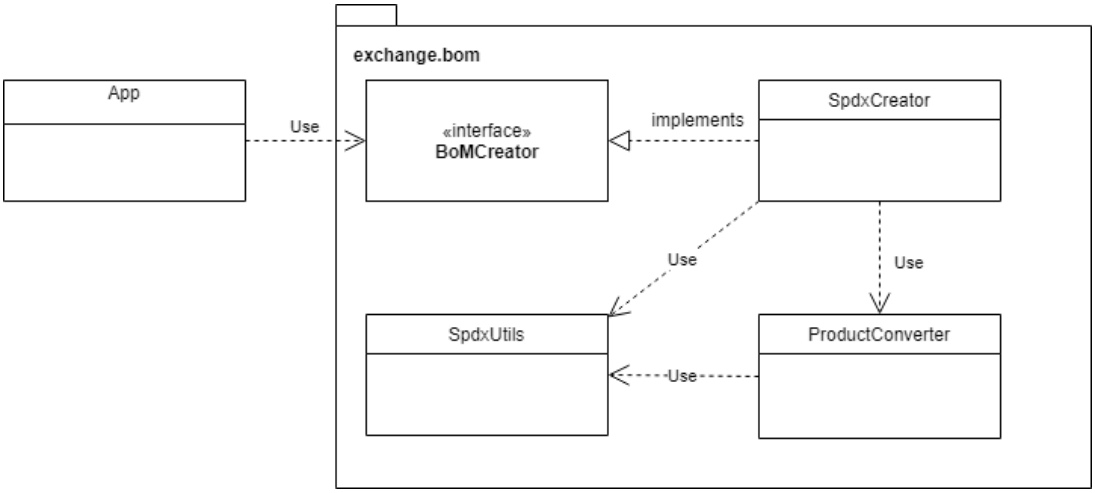
The next subchapters are covering the implementation details, using the class names of the Product Model tool and the RDF model[6], which is provided by the SPDX tools. Starting with the general design and structure of the implementation then going over to the *BoMCreator.* The interface which the main project interacts with to create BoM artifacts. Afterwards we discuss *BomCreator's* realization, the *SpdxCreator.* Lastly the 4th subchapter is about the *ProductConverter* and how it is transforming the product model into the RDF model.

Two additional maven dependencies[7] got added to the tool, both licensed under the Apache License 2.0. It requires the licensee to provide attribution, but as the Product Model tool is still only used internally, there is no need yet to include them in the licensing ("Apache License, Version 2.0," 19-Oct-18).

5.1 Architecture and Design

The Architecture and Design chapter is not a big portion of the thesis. At the start I thought about designing my own BoM artifact and the corresponding model. However, the added value the SPDX standard offers, can probably not get outdone by some self-designed solution. The SPDX tools not only offer utility functions but an RDF model as well.
The *BoMCreator* is the actual interface between the main tool and the BoM generation. A crawling process creates a *Product* and transmits it to the *BoMCreator.* To realize the functions of the *BoMCreator,* the *SpdxCreator* uses the *ProductConverter* and the *SpdxUtils.* The *ProductConverter* creates the *SpdxDocument* with the help of the RDF model and *SpdxUtils.*
All the named classes are interacting with the SPDX tools, either by using the verify or compare functionalities, converting an RDF file or just filling the RDF model.

[6] See: Appendix A for a UML diagram of the RDF model
[7] See: https://mvnrepository.com/artifact/org.spdx/spdx-tools/2.1.12 and https://mvnrepository.com/artifact/com.fasterxml.uuid/java-uuid-generator/3.1.5



The standard interaction between the classes can be seen in Figure 3.
Figure 3: Interaction between the implemented classes

5.2 BoM Creator

The Product Model tool interacts with the implementation, using the *BoMCreator* interface depicted in Figure 4. When starting the Product Model tool, command line arguments are used to start the correct processes. Three of them are important for the BoM generation. The first one, chooses between three different formats (RDF/XML, Tag/Value, Spreadsheet) a BoM artifact can get created in. The other two are for using the compare and verify utilities of the *BoMCreator*. The component model is extracted from a software product by a crawler, it is called *Product*. Only with the data included in the *Product* the *BoMCreator* can create a BoM artifact. The comparison of BoM artifacts can be between two files or a file and a *SpdxDocument.* Verification can only be used on external files, as the internally files get verified anyways after their creation process. Depending on the command line arguments and whether a crawl process finished, different methods of the *BoMCreator* get called. The *BoMCreator* can create a simple and a complex BoM artifact and can create a verification and/or comparison file.

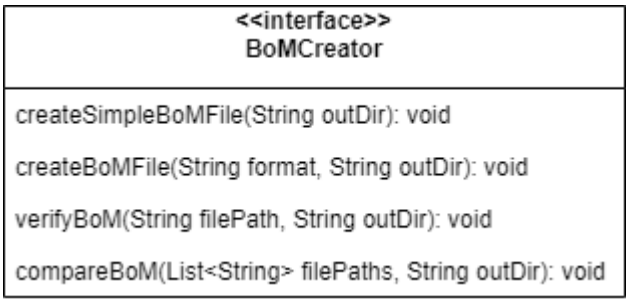


Figure 4: Interface BoM Creator

5.3 SPDX Creator

The realization of the *BoMCreator* is the *SpdxCreator,* which means all the options and information gets passed onto it. It can be instantiated with different arguments. After a crawl run the constructor is called with a *Product* as an argument, which automatically creates a *SpdxDocument* using the *ProductConverter*. Another possibility is to call the constructor without any arguments at all. Though, then only the tools which use external files can be used, if a *SpdxDocument* isn't set subsequently.
The *SpdxCreator* implements the four methods of the *BoMCreator*. The first one creates a simple BoM file, this is implemented using *SpdxUtils. SpdxUtils* creates a List of Strings out of a given *SpdxDocument*. The list then gets written into a text file. Furthermore, *SpdxCreator* creates SPDX files in different formats depending on the given format value. It always generates an RDF file as the SPDX tools are using an RDF model internally and only transform RDF files into another format.The comparison of two BoM aritfacts is also implemented using the help of the SPDX tools, but the *CompareSpdxDocs* class had to get modified because it called System.exit. The produced file lists the differences between the files, mainly focusing on the packages and licenses. When verifying a SPDX file, all errors or missing mandatory fields get written into a text file.

5.4 Product Converter

As the name implies, the ProductConverter converts a *Product* into a *SpdxDocument*. It tries to use all the available information of the *Product,* but some attributes like the *InterfaceData* do not have a representation in a *SpdxDocument*. Furthermore, its goal is to fill all the attributes, that are tagged "mandatory" by the SPDX specification. Even if filling those fields just means assigning values, which declare that the necessary information could not be determined. Because this is done correctly no verification errors occur when creating a SPDX file out of the *SpdxDocument.*

In the upcoming subchapters, all used and mandatory fields as well as all mandatory sections of a *SpdxDocument* get discussed. Namely, the Document Creation Information, the Package Information, the Relationships and Other Licensing Information. Missing sections are the File Information, the Snippet Information and the Annotations. The product model right now has no information about the included files of a package, so no File Information can be added to the document. In this context, the word snippet refers to parts of a file that are included from another source maybe indicating different licensing as the file. Such information is not detected by the maven crawler either. Lastly, annotations are mostly used for reviewing other *SpdxElements,* but right now there is not any useful information that can be added this way.

5.4.1 Document Creation Information

The Document Creation Information is a fundamental section of the *SpdxDocument*, it holds important metadata about the project itself. All fields of the section, in which version of the specification they got added and if they are mandatory can be seen in Figure 5.

Mandatory	Added	Field Name
X	1.0	2.1 SPDX Version
X	1.0	2.2 Data License
X	2.0	2.3 SPDX Identifier
X	2.0	2.4 Document Name
X	2.0	2.5 SPDX Document Namespace
	2.0	2.6 External Document Reference
	1.2	2.7 License List Version
X	1.0	2.8 Creator
X	1.0	2.9 Created
	1.0	2.10 Creator Comment
	1.1	2.11 Document Comment

Figure 5: Document Creation Information (Linux Foundation, n.d.)

SPDX Version: The version of the SPDX specification the Implementation uses, which is the latest one, version 2.1. It is provided so tools know how to interpret the information and to enable possible compatibility with tools based on future versions.

Data License: The Creative Commons CC0 1.0 license[8] gets used as *Data License*, so the fields in the *SpdxDocument* where the creator can write down any meta information, like comments or annotations, don't get restricted by intellectual property. Making the reuse and sharing of documents possible.

SPDX Identifier: "SPDXRef-Document" is used to refer to the *SpdxDocument* as a *SpdxElement* internally. For external references the *SPDX Document Namespace* and a *Checksum* are added.

Document Name: Is just the name given by the creator. In our case it is the name of the *Product.*

[8] https://creativecommons.org/publicdomain/zero/1.0/legalcode

SPDX Document Namespace: The Namespace is a unique identifier to refer to *SpdxElements* in the document. It is composed of the *Products* homepage URL, the *Products* name and a version 5 UUID. Ideally it would be the creator's homepage plus the direct path to the SPDX documents on the website, plus the UUID. The UUID is generated out of the homepage and the name, using the Java UUID Generator[9]. This is not perfect either, because software products with different versions get assigned the same ID, but the *Product* does not have any kind of version information. The Namespace is also used to initialize the *SpdxDocument* container class.

License List Version: Refers to the Version of the SPDX License List. The field is filled with the latest version, version 3.2. This indicates, that when the list gets updated it is possible for licenses in the document to be outdated. Right now, it is a bit of unnecessary information as the found license information does not get transformed into listed licenses yet.

Creator: Identifies the creator or creators of the document to assess the reliability of the document's information. In our case it is the Product Model tool and the Professorship for Open Source.

Created: Time and date of the creation, in the Coordinated Universal Time (UTC) format, to realize if the SPDX document is outdated.

(Linux Foundation, 2016, pp. 9–17)

[9] https://mvnrepository.com/artifact/com.fasterxml.uuid/java-uuid-generator

5.4.2 Package Information

Every package of the analyzed software project has a Package Information section. Figure 6 is a table of all the included fields of the section. The *SpdxDocument* not only includes all root packages and their sub packages but also the software project itself as a package. Though the project is described by the *Product* class, which describes different information as the *Component* class. This is the reason why some fields are filled with different information for the main package, representing the project than for all the other packages, created out of the *Components*. The *ProductConverter* ensures that no duplicate *Packages* get created. *Components* are considered equal if their name and version are the same.

Mandatory	Added	Field Name
X	1.0	3.1 Package Name
X	2.0	3.2 Package SPDX Identifier
	1.0	3.3 Package Version
	1.0	3.4 Package File Name
	1.0	3.5 Package Supplier
	1.0	3.6 Package Originator
X	1.0	3.7 Package Download Location
	2.1	3.8 Files Analyzed
X	1.0	3.9 Package Verification Code
	1.0	3.10 Package Checksum
	1.2	3.11 Package Home Page
	1.0	3.12 Source Information
X	1.0	3.13 Concluded License
X	1.0	3.14 All Licenses Information from Package
X	1.0	3.15 Declared License
	1.0	3.16 Comments on License
X	1.0	3.17 Copyright Text
	1.0	3.18 Package Summary Description
	1.0	3.11 Package Detailed Description
	2.0	3.12 Package Comment
	2.1	3.13 External Reference
	2.1	3.14 External Reference Comment

Figure 6: Package Information (Linux Foundation, n.d.)

Package Name: Is the given name by the creator. For the main package it is the name of the *Product,* whereas all other packages use the name of the *Component* they are representing.

Package SPDX Identifier: "SPDXRef-" and an ID string is used to refer to this *SpdxElement* internally. For external references the document namespace is added. The RDF model provided by the SPDX tools just uses an integer counter, starting at one, to create the ID string.

Package Version: Is just the same as the *Component's* version.

Package File Name: For all *Components* the file path of their *Artifact* gets used as the file name. The *Product* does not save the file path of the project itself, however it is not a mandatory field and can be overlooked.

Download Location: In contrary to the main package, the other packages don't have a specified download location. But since the field is mandatory, the field is filled with "NOASSERTION" to signal that the tool could not determine this field.

Package Verification Code: The code gets generated by an algorithm that uses the Secure Hash Algorithm 1 (SHA-1) on all files to identify the content of a package. The SPDX tools can generate the code out of *SpdxFiles*, but *Components* have no information of included files, hence it cannot be created. This field is a special case, when *Files Analyzed is* set to true or omitted then the *Package Verification Code* is mandatory, otherwise it can be left out. According to the specification it is only allowed to set *Files Analyzed* to false when the package does not contain any file information, which is true in our case.

Package Checksum: The *Component's Artifact* information gets used to create a *Checksum* with the help of the SPDX tools.

Concluded License & Declared License: The *Concluded License* is the license declared by the creator and the *Declared License* is the license declared by the author. The main package has always the same license as declared and concluded license. It is either one license or a *ConjunctiveLicenseSet* depending on whether the *Product* had one or multiple declared licenses. When *LicenseData* was found in a *Component* the declared and concluded licenses are the same. If a *Component* has no *LicenseData* the *Concluded License* inherits the license from the nearest "parent" package while the *Declared License* is a *SPDXNoneLicense* signaling no license was found in the package.

All Licenses Information from Package: Contains all the found license information in the package. To get the license information from all sub packages, the *ProductConverter* has to go through all *Components* at the start and pass on the *LicenseData* from sub packages to their "parents".

Copyright Text: Neither the *Product* nor the *Components* contain any copyright information. As it is a mandatory field it must get filled with "NOASSERTION" to signal that the tool could not determine the field.

(Linux Foundation, 2016, pp. 18–38)

5.4.3 License Information

Mandatory	Added	Field Name
X	1.0	6.1 License Identifier
X	1.0	6.2 Extracted Text
	1.1	6.3 License Name
	1.1	6.4 License Cross Reference
	1.1	6.5 License Comment

Figure 7: Other Licensing Information (Linux Foundation, n.d.)

The subchapter explains which kind of license information gets added to the different *SpdxElements* in the *SpdxDocument.* In the implementation licenses get identified just by their extracted name, as this is the only available information in *LicenseData*. Using this identifier, the implementation ensures that no duplicate licenses get added. Every time license information is created, the *ProductConverter* already checks if the id can be matched to a listed license. Though the chances of this happening are low and in the normal case *ExtractedLicenseInfo* gets created instead. To improve upon it, one could transform the found license name to the most similar representation in the License List. As mentioned before, the main package has a single license information or a *ConjunctiveLicenseSet.* The *ConjunctiveLicenseSet* is a SPDX License Expression adding together multiple licenses. Normal packages could have multiple *LicenseData* as well, so they can get a *ConjunctiveLicenseSet* assigned too. Often when scanning the POM files of maven projects, no license information can be found for dependencies. Without extracted license information, a *SpdxNoneLicense* is filled in for the declared license and the concluded license inherits the closest license information from its nearest "parent" package. The following fields are only needed for licenses not included in the SPDX License List.

License Identifier: Unique identifier that can be used in the package and file information. Since version 1.2 of the SPDX specification it is allowed to use strings as identifiers. The identifier is always composed of "License-Ref-" plus the ID. The found name in *LicenseData* is used as the identifier.

Extracted Text: This field represents the found license text of a package. The product model does not have any information besides the license name, because of that the license name is the only information defining the license text.

5.4.4 Relationships

Mandatory	Added	Field Name
X	2.0	7.1 Relationship
	2.0	7.2 Relationship Comment

Figure 8: Relationship (Linux Foundation, n.d.)

The relationship section includes all the information about the relations between *SpdxElements*. The only interesting field is the *Relationship.* The content of this field is, similarly as the product model, composed of a source and a target *SpdxElement* identifier and a relationship type. There always needs to be a "Describes" relationship between the *SpdxDocument* and at least one *SpdxPackage* to organize the document. Between parent and sub packages the relationship "Contains" is set.

6 Evaluation

In chapter 4 requirements, which the implementation must fulfill, got set up. Now it is time to check every one of them and use them evaluate the resultant implementation.

6.1 Check RQ1: SPDX representation

The first requirement was to create a valid SPDX file. During the generation of SPDX files no verification errors occur. This indicates that the file is properly structured and no mandatory fields are missing. Yet, the creation process can still be improved upon. The product model tool could add new relevant information, which in turn makes it possible for the SPDX documents to become more detailed as well. But the resultant artifact is a satisfactory result. The main goal of this thesis was, to create a way to automatically generate BoM artifacts out of the product model. The created SPDX files can communicate BoM information in a standardized and precise way.

6.2 Check RQ2: Human readable report

Requirement two, was to create a human readable file. The results of the thesis can create a simple text file. The file right now displays all the components and their licenses in a treelike structure like the RDF format. However, for the report to add value it is held simpler than RDF files. This makes it easier for humans to get a rough overview of the components and their licenses. The testcases only check if a text file was created but do not validate the content. The content was solely checked manually, inspecting different reports by hand. The text file gets also created by using the *SpdxDocument,* it might be smarter to change it to use the product model directly. This way a BoM artifact could be created without relying on the SPDX tools.

6.3 Check RQ3: Comparison of BoM artifacts

The third requirement was to be able to compare BoM artifacts and display their differences. In the implementation two external BoM files can be compared and one external BoM file can be compared with the internal model. Similarly to RQ2, only the creation of such a file gets tested automatically and the content of the files was only checked by hand. However, the comparison functionality is based on the SPDX tools and they test all their classes thoroughly.
The comparison file displays every little difference between two SPDX files, even when just the creation time differs. Also, it displays a summary over all missing or differing licenses and components.

6.4 Check RQ4: Can be used in a continuous integration / continuous deployment pipeline

Ultimately, RQ4 required the implementation to be able to be integrated into a continuous integration / continuous deployment pipeline. As mentioned in the evaluation scheme, the fulfillment of RQ1 and RQ3 imply the fulfillment of RQ4.
After every iterative step in development, a crawl run could be started and a BoM artifact can be created. Because of RQ3 changes of components or licenses can detected between different versions of a product.

7 Conclusion

7.1 Results

The goal of this thesis was to automatically generate and track BoM artifacts. In the introduction, the motivation for BoM artifacts was made clear. This was followed by some basics about the structure of the Product Model tool and the SPDX specification. The definition and evaluation of the requirements could have been defined more detailed. The architecture and design of this thesis was kept very short, as the reasonable decision was made to use the RDF model from the SPDX tools. While explaining the implementation details some not so relevant fields, from the SPDX specification could have been left out, but his way it resembles as a general guidance.

In the end, the Product Model tool got extended to now generate and track BoM artifacts and thereby fulfills the general goals and requirements of this thesis.

7.2 Improvements

The product model itself could get extended to hold more detailed information. This may only be possible by improving the crawler or create other processes to analyze a project.
The fields, that get set in the *ProductConverter,* could get revisited. Especially the relationships. It might be better to use another one or multiple relationships to represent the component model. The license information found could get transformed into identifiers of the SPDX License List. This way a full license text would become available. The implementation cannot import SPDX files yet, but it would be a necessary feature to make human annotations possible.

Appendix A:

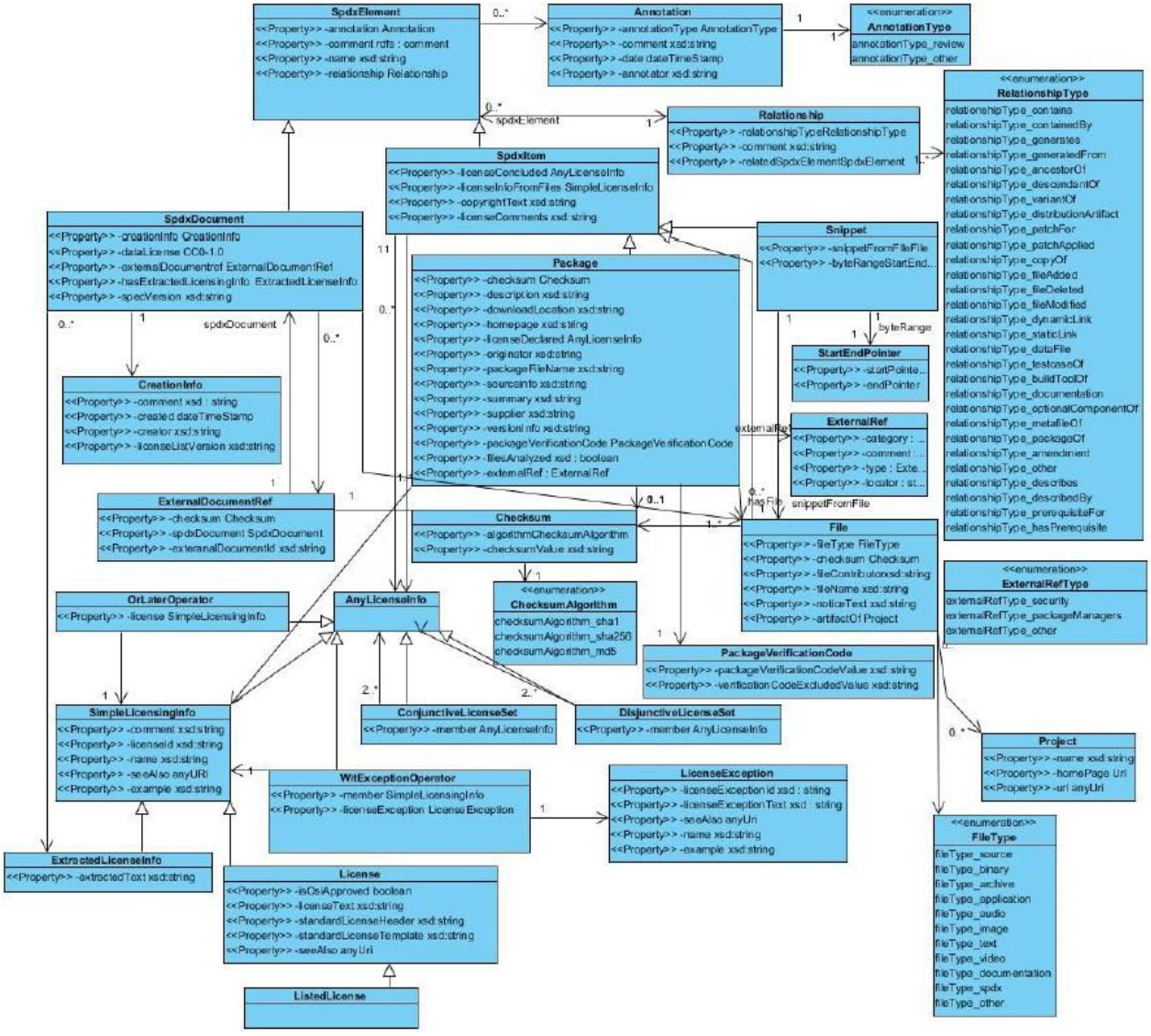


Figure 9: RDF model (Linux Foundation, 2016)

References

About | Software Package Data Exchange (SPDX). (n.d.a). Retrieved from https://spdx.org/about

Apache License, Version 2.0. (19-Oct-18). Retrieved from https://www.apache.org/licenses/LICENSE-2.0

Business Team/Adoption - SPDX Wiki. (25-Jan-2016). Retrieved from https://wiki.spdx.org/view/Business_Team/Adoption

Councill, W. T., & Heineman, G. T. (2001). *Component-based software engineering: Putting the pieces together*. Boston, Mass.: Addison-Wesley.

Deshpande, A., & Riehle, D. (2008). The Total Growth of Open Source. *Russo B., Damiani E., Hissam S., Lundell B., Succi G. (Eds) Open Source Development, Communities and Quality. OSS 2008. IFIP – the International Federation for Information Processing, Vol 275*, 197–209. https://doi.org/10.1007/978-0-387-09684-1_16

European Commission. (2017). The economic and social impact of software & services on competitiveness and innovation (SMART 2015/0015), 197–198.

Frakes, W. B., & Kang, K. (2005). Software reuse research: status and future. *IEEE Transactions on Software Engineering*, *31*, 529–536. https://doi.org/10.1109/TSE.2005.85

Haefliger, S., Krogh, G. von, & Spaeth, S. (2008). Code Reuse in Open Source Software. *Management Science*, *54*, 180–193. https://doi.org/10.1287/mnsc.1070.0748

History of the OSI | Open Source Initiative. (October 18). Retrieved from https://opensource.org/history

Licenses & Standards | Open Source Initiative. (n.d.b). Retrieved from https://opensource.org/licenses

Linux Foundation. (n.d.). Software Package Data Exchange(SPDX) one pager v.21.

Linux Foundation. (2016). Software Package Data Exchange(SPDX) Specification Version: 2.1.

Raymond, E. S. (1999). *The cathedral and the bazaar: Musings on Linux and Open Source by an accidental revolutionary* (1. ed.). Beijing: O'Reilly.

REST API v3 | Licenses. (26-Oct-18). Retrieved from https://developer.github.com/v3/licenses/

REUSE practices - version 2.0. (14-Dec-17).

Scheffer, D. (2018). An Artifact Crawler for Determining Code Component Architectures. Friedrich-Alexander University Erlangen-Nürnberg, Professorship for Open Source Software.

Sojer, M., & Henkel, J. (2010). Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments. *Journal of the Association for Information Systems*, *11*, 868–901. https://doi.org/10.17705/1jais.00248

SPDX IDs: How to use | Software Package Data Exchange (SPDX). (n.d.c). Retrieved from https://spdx.org/ids-how

Tools | Software Package Data Exchange (SPDX). (n.d.d). Retrieved from https://spdx.org/tools

Using SPDX | Software Package Data Exchange (SPDX). (n.d.e). Retrieved from https://spdx.org/using-spdx

Using SPDX Documents | Software Package Data Exchange (SPDX). (n.d.f). Retrieved from https://spdx.org/using-spdx-documents

Using SPDX License List | Software Package Data Exchange (SPDX). (n.d.g). Retrieved from https://spdx.org/using-spdx-license-list

Wheeler, D. (19-Jul-18). david-a-wheeler/spdx-tutorial. Retrieved from https://github.com/david-a-wheeler/spdx-tutorial#spdx-tutorial