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

DENNIS SCHEFFER
MASTER THESIS

# AN ARTIFACT CRAWLER FOR DETERMINING CODE COMPONENT ARCHITECTURES

## MAVEN-BASED CRAWLING

Submitted on 16 April 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, 16 April 2018

# License



---

Erlangen, 16 April 2018

# Abstract

Since it has become very common for software projects to incorporate open source components, it is increasingly important to analyze their dependencies to avoid potential risks like license violations or security vulnerabilities. To determine and solve such issues we need a representation of products regarding their component architectures. We call such a representation a *product model*. In software projects, dependencies are usually handled by build systems like Maven. Therefore this thesis focuses on the development of a crawler application that recognizes dependencies and corresponding metadata from Maven projects. This information is used to build a common product model and export the model in a machine-readable format so that the application can be integrated into existing toolchains. The crawler forms the basis for a larger application that aims to solve the aforementioned issues.

# Contents

# List of Abbreviations

**ADL** architecture definition language

**AGPL** GNU Affero General Public License

**API** application programming interface

**CLI** command-line interface

**EPL** Eclipse Public License

**GPL** GNU General Public License

**JVM** Java Virtual Machine

**LGPL** GNU Lesser General Public License

**NIST** National Institute of Standards

**OSC** open source component

**OSCM** open source component model

**OSI** Open Source Initiative

**OSS** open source software

**POM** project object model

**RTE** run-time environment

**SCM** source code management

# 1 Introduction

In recent years the prevalence of open source software (OSS) has become undeniable. It is rather normality than exception for software to contain open source components. Thus the collective interpretation of OSS has shifted from a solely ideological movement to an economic force. Even Microsoft, the years-long opponent of open source, has turned into a supporter by becoming a platinum member of the Linux Foundation (Linux Foundation, n.d.). It is unlikely that companies like Microsoft support open source development out of altruistic reasons. They have rather acknowledged that there are numerous incentives for participating in open source. These are not anymore just an individual's need to scratch one's itch but have turned into economic benefits (Riehle, 2007; Dahlander & Magnusson, 2005; Bonaccorsi & Rossi, 2006) and even have brought forth new business models (Riehle, 2009; Bonaccorsi, Giannangeli, & Rossi, 2006).

Despite the number of benefits some companies have been hesitant in the adoption of OSS. This may be due to potential security risks that OSS may inherit. It has even been reported by practitioners that 26% of open source libraries in Java have known security vulnerabilities (Contrast Security Inc., 2014). This may lead to companies that are developing safety-critical applications to refrain from using OSS even though others have argued that the OSS review process is very efficient in identifying bugs and vulnerabilities due to the potentially large number of participants (Payne, 2002).

Another factor that is slowing OSS adoption is due to the very nature of open source. After all OSS is published under open source licenses. The legal framework around OSS is what has made open source revolutionary in itself but can be undesirable to companies who wish to use OSS. This becomes apparent in the guidelines for open source licenses that the Open Source Initiative (OSI) has defined (Open Source Initiative, 2007). For instance these guidelines state that an open source license must require the source code of a program to be somehow available for users of the program. Therefore it is undesirable for companies who wish to develop proprietary software to publish their software under such a license.

But even if a company would only wish to *use* a certain open source component, it may still be problematic due to the legal nature of licenses. German and Hassan (2009) refer to a license as

- *a legal mechanism used by the copyright or patent owner (the licensor) to grant permission to others (the licensee) to use and exploit her intellectual property in ways that would otherwise be forbidden by copyright or patent law.*

Such a permission is of paramount importance to a software vendor who wants to use open source components. But these permissions often come with a set of obligations. A prominent example of a license with sometimes undesirable obligations is the GNU General Public License (GPL). Like other open source licenses the GPL grants a licensee the right to use the licensed source code but requires the licensee to maintain the GPL on modified versions of the source code. If a company wishes to modify an open source component by integrating a proprietary software component they would need to license the entire work using the GPL (Free Software Foundation, Inc., 2007b; German & Hassan, 2009). If the proprietary license cannot be re-licensed using the GPL, the company is in violation.

If this were all that one needed to keep track of regarding licenses it would be fairly easy to manage open source components. Unfortunately, there are many different open source licenses. The Open Source Initiative alone lists 83 approved licenses of which each has a different set of obligations (Open Source Initiative, n.d.). Keeping track of these becomes harder the bigger a software project becomes until it is nearly impossible to keep track of them manually. When faced with a software project that contains many open source dependencies it becomes necessary to have tools that help to keep track not only of used licenses but also of other properties like known security vulnerabilities.

## 1.1 Existing tools

Apart from commercial solutions there exist a number of academic and open source tools that aid in the tackling of the aforementioned problems. Fossology for example is a toolkit designed to help with compliance activities. It scans files regarding license information using pattern recognition methods. Fossology helps in the license clearance process by structuring the findings and providing a user interface to manually check as well as document clearing decisions (Gobeille, 2008; FOSSology Workgroup, 2017).

German and Penta (2012) suggest another method regarding the license compliance problem for Java software. They developed a semiautomatic process

consisting of component identification, provenance discovery, license identification, and licensing requirements analysis called *Kenen*. It differs from Fossology in the sense that the first step in the clearance process is to create a repository with pre-approved components. German and Penta (2012) argue that it is "good policy". For provenance discovery the authors analyze source and byte code using their tool *Joa* which uses pattern matching algorithms to determine a similarity and an inclusion index. The closest match is assumed to be the used open source component. The license identification process functions similar to Fossology using a pattern-matching-based tool called Ninka (German, Manabe, & Inoue, 2010). The clearing decision is also similar to Fossology as clearance can only be determined manually by someone with legal and software expertise.

A different tool for provenance discovery amongst other things is *Sourcerer*. It takes advantage of the fact that the source code of open source components is usually publicly available. *Sourcerer* builds a database of open source code and its meta information by crawling a number of different open source repositories. The retrieved data is analyzed regarding its structural and textual aspects. The database is primarily used to feed a source code search engine but also provides other services (Bajracharya, Ossher, & Lopes, 2014).

So far all the mentioned tools are collaborative to some extent but the data retrieval is always some centralized process. Another approach is proposed by *FLOSSmole*. It is a database that contains meta information about open source components somewhat similarly to Sourcerer and is mostly focused on research purposes. The crucial difference is that the data can be submitted by different individuals regardless of how the data was gathered. It is expected that people who use *FLOSSmole's* data give back to the community in a BSD-like fashion (Howison, Conklin, & Crowston, 2006).

Aside from open source license clearance and research a relevant aspect for tool support is security. It has increasingly been an issue since OWASP recognized that the usage of components with known security vulnerabilities is one of the biggest security problems up to date (OWASP Foundation, 2013). OWASP proposes a solution for this problem with the *OWASP Dependency-Check* (OWASP Foundation, 2017). It is a tool that generates reports regarding known security vulnerabilities of a Java or .NET application. The list of known security vulnerabilities is retrieved from data feeds by the National Institute of Standards (NIST).

## 1.2 Alternative approach

The tools in Section 1.1 are mostly solving specific problems. But the reality of development involving open source is that often many problems are relevant at the same time. If someone wanted to analyze a project's dependencies regarding their licenses and security vulnerabilities using for instance Fossology and the OWASP Dependency-Check, these tools would need to run independently because they do not have the same internal representation of project dependencies. To automate this process one would need to write a top-level application that runs these two separate applications. In theory this is not very problematic but such an application is not very reusable as it is tailored to a very specific use-case. If one were to extend the application nevertheless, it would soon become very inefficient and impractical. So to avoid problems regarding tool chain composition it is advantageous to develop a generic, serializable model. Such models are typically of software products and therefore are referred to as *product models* in the following.

German and Penta (2012) mention that component identification and the corresponding provenance discovery are crucial first steps for further analysis in their *Kenen* process. So a product model centered around component architectures seems sensible. Unfortunately, the term *component* is surrounded by controversy. Heineman and Councill (2001) provide a definition:

- *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.*

The identification of such components is not trivial since components are usually architectural units that cannot be directly applied to source code and are typically documented using some architecture definition language (ADL). In the case of open source components these documents are rarely available and if they are, they are not uniformly provided to users. This makes automated retrieval of component architectures difficult. But even if these documents were easily available, they usually do not define a component model. Without a common component model comparison of components becomes hard because the definition of components is only relative to the definition of a component model as Heineman and Councill (2001) implicate. Alternatively, German and Hassan (2009) state that a lax definition of components is sufficient for legal questions regarding open source. So until component models are further examined in section 3.1 it can be assumed that an informal definition is sufficient for other problems in open source, too.

A fundamental concept of open source development is the reuse of source code. To make reuse practical there usually is a package management system that

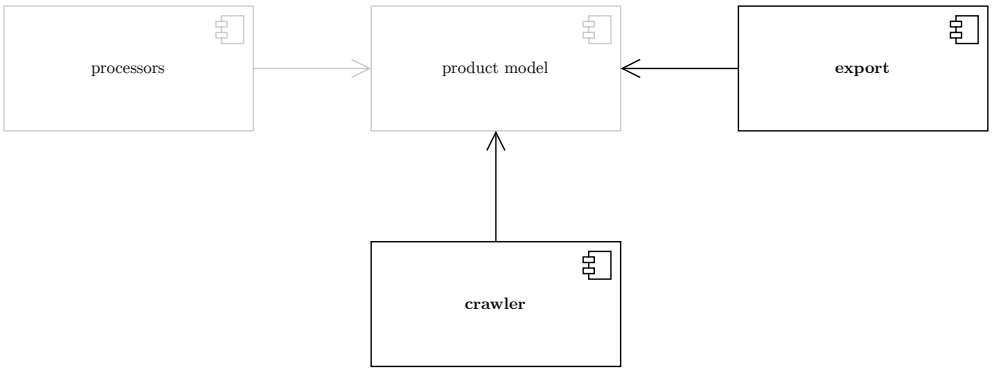


**Figure 1:** Component diagram showing system architecture

handles the downloading of required inter-dependencies[1] needed to build and run an application (German, Gonzalez-Barahona, & Robles, 2007). Due to their reusability and their ability to be composed, inter-dependencies can be thought of as software components. Consequently, it is sufficient to capture dependencies to create a representation of an open source project's component architecture.

The assumption is that these specific properties of OSS can be used to solve the problems highlighted in chapter 1. This thesis is part of the Open Source research group's effort at the Friedrich-Alexander university to build a tool for that purpose. The contribution of this thesis is a crawler tool that creates a representation of a software projects component architecture. The main idea in this process is that component architectures are identified by analyzing software artifacts. After extraction component architectures form the basis for further analysis in compliance to the product model. The product model is at the center of the system architecture as shown in figure 1. Crawlers are essential for feeding the product model with data but are just part of a larger application. This thesis will focus on the design of a crawler and an export component. In figure 1 the lower opacity of the remaining components signals that these components are not the focus of this thesis. But the bigger picture indicates that after a product model has been filled by a crawler it is used by processors to solve the problems mentioned before.

Due to the typically sheer amount of dependencies OSS usually requires some configuration management system according to German et al. (2007). This poses the opportunity to extract dependency meta data from the configuration management system at hand. Ultimately the application must support different kinds of languages and systems. But this thesis will focus on the analysis of Java software. In their *Kenen* process German and Penta (2012) haven relied on Maven2 for their provenance discovery. It makes all the more sense to focus on Maven in this thesis as it is one of the most popular build systems for Java OSS.

[1]Inter-dependencies will also be referred to as just *dependencies* throughout this thesis.

Consequently, the crawler is an extensible tool that fills a component-based product model with a respective product's architectural information like component hierarchy and metadata as well as exports the gathered data in a machine readable format (e.g. XML, JSON, YML) for further analysis with other frameworks. I will examine the requirements for such a crawler tool in chapter 2. Subsequently, I will shine some light on the architecture and design of the tool in chapter 3. This includes general design considerations in respect to the nature of the software products which are to be analyzed. Chapter 4 focuses on the implementation of the crawler and export modules. In conclusion chapter 5 highlights the evaluation of the crawler.

# 2 Requirements

Requirements analysis typically presupposes the identification of relevant stakeholders. But since the project of which this thesis is a part of is still in its early stages the relevant stakeholders are limited to the developers involved. In the future there will be more stakeholders involved and therefore the requirements below are subject to change. The focus of this thesis lies in developing the functionality first. The functional requirements are explained in sections 2.1 to 2.6. Non-functional requirements are not the main focus of this thesis but nonetheless there are a few key non-functional requirements which are explained in section 2.7 and section 2.8. More non-functional requirements will become relevant at later stages of the project.

## 2.1 F01: Crawler based on product model

- The software fills the product model based on provided inputs with architectural component data from given software artifacts according to a specific crawler implementation.

This requirement is the central functionality of the tool. A crawler must identify a software projects component architecture based on the product model. The crawling process must be able to support different types of future crawler implementations as the products may be represented by different build systems or even programming languages.

## 2.2 F02: Partial crawling

- The software supports crawling of partial products. The sum of parts must be equivalent to the entire product model.

Partial crawling can improve performance of the tool as it may be possible to run multiple threads in parallel that produce partial product models. If one were to

look at this requirement only from the performance angle, it may as well be a non-functional requirement. The functional angle is that this requirement is also necessary for the combination of partial product models resulting from different crawler implementations. After all it is possible to use multiple build systems or programming languages in a single project. Also, it turns out that in the case of Maven multi-module projects it is also necessary to crawl each module individually and then put the parts together.

## 2.3 F03: Exportable product model

- The software exports product model data in a machine readable format so that the tool can be used as part of a toolchain.

The crawler tool must be able to export the product model in some machine readable format to be re-usable as part of a toolchain. The most common formats for structured data are XML, JSON and YAML. So it makes sense to support all of these data types as exportable types.

## 2.4 F04: Maven project support

- The software supports the analysis of Maven projects. Maven is one of the most common build tools for Java. This serves as a reference implementation for other, future crawlers.

As mentioned in the requirements **F01** and **F02** the tool must support different crawler implementations. This thesis will only focus on the implementation of a Maven based crawler as the implementation of additional crawlers would be out of its scope.

## 2.5 F05: Identify component relationships

- The software identifies relationships between components like static linking, dynamic linking or web calls.

Certain problems in OSS like license compliance are dependent on the manner in which an open source component is used. Therefore, it is important to identify what type of relationship certain components have so that future processors[1] have the necessary information to solve such problems.

[1] Processors as mentioned in section 1.2

## 2.6 F06: Product data version

- The software supports versioning/timestamping of product data so that a comparison of changes over time will be possible.

One of the goals of the tool is to monitor changes over time in product models. Therefore it must be possible to differentiate between snapshots of product models. This can be achieved by adding timestamps to product model snapshots.

## 2.7 Q01: Modifiability and extensibility

- The software is easily extensible by additional crawlers. It is sufficient to implement one interface for new crawler implementations.

As this thesis is part of an academic project it can be assumed that a couple of different developers will work on future crawler implementations. Therefore it is crucial to make modifiability as easy as possible. A crawler must comply to some interface so that the underlying implementation can start a crawler. This is unavoidable complexity. So to keep it as simple as possible the implementation of a single interface must be sufficient. That way a developer does not need to understand the full source code to start working on a crawler implementation.

## 2.8 Q02: Reliability

- The software identifies more than 50 percent of all components of a given software product.

The tool must have some measure as to how well it performs. Therefore the tools reliability is measured by how many components in a project it can identify. This measure is the minimal threshold for confidence in the tool. Anything below the threshold will result in the tool being unreliable.

## 2.9 Evaluation scheme for requirements

To ensure that the software is functional it is in order to run tests even though the successful completion of tests does not ensure the program's correctness. In fact the absence of errors can never be guaranteed with the exception of formal testing methods which are not suitable for this thesis. If one had unlimited resources one

could approximate correctness by running very thorough tests. But most of these would be out of scope in the context of this thesis. To ensure minimal confidence in the code, the tests cover a 100% lines of code. This is accomplished through unit tests.

In terms of non-functional requirements a separate approach is taken. **Q01** can be easily evaluated by looking at the number of interfaces that need to be implemented for a new crawler. For that there is no need for any special test cases. But **Q02** is quite tricky to evaluate. This is due to the fact that it is difficult to get a test oracle that can be asked during a test run whether the number of recognized components is sufficient. If there was a program that was capable of posing as an oracle then the implementation of a crawler that does the same would be obsolete. To get at least the slightest grasp of whether **Q02** is met, the software is tested against a small number of handpicked projects where the number of components has been determined manually. This is used to design a number of integration tests to build confidence that the crawler's units correctly operate with each other.

# 3 Architecture and Design

To satisfy the requirements established in chapter 2, a number of software architectural design aspects have to be discussed first. Section 3.1 introduces these aspects by addressing the definition of components in relation to their link to component models since software components are an integral part of the results produced by the application. This is followed by the actual crawler design considerations in section 3.2.

## 3.1 Component models

The term component is somewhat controversial and it is therefore necessary to define the term in the context of this thesis in order to achieve requirement **F01** as its goal is to create a **component-based** representation of architectural information. So what exactly is a component?

Historically a component refers to some electronic building block that can be embedded in arbitrary circuits, provided that the circuits adhere to the rules imposed by the building blocks. Typically, this means that circuits provide some standardized sockets where certain components can be embedded. Like it is the case with many other interdisciplinary terms, computer science has adopted the term component to refer to something vaguely similar – hence the controversy. Szyperski (2002) prominently refers to a component as "a unit of composition with contractually specified interfaces and explicit context dependencies only". He acknowledges that his definition is rather general and that there is more to components than just that. Especially, since components can be fundamentally different from each other, a respective component has to impose rules for things like interaction and composition with other components. Such rules form a component model which is part of the mentioned *explicit context dependencies*. Heineman and Councill (2001) refer to a *component model* as an entity that defines components' specific interaction and composition standards. This makes component models the essence of what specific components can do. Therefore, the definition of specific components is implicated by their component model. As

component models are relevant to the understanding of components, throughout this thesis the definition by Heineman and Councill (2001) will be used:

> *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.*

With the understanding about component models it can be deduced that it is not practical to compare components that comply to different component models. If one were to compare such components nevertheless, it would be under the assumption of a very general definition of components like the one proposed by Szyperski (2002). In that case all that components would have in common would be that they are units of composition, have interfaces and explicit context dependencies only. But their manner of composition and context dependencies might be fundamentally different which makes them uncomparable in that respect. Therefore, one could only compare components regarding their interfaces. Unfortunately, the reality of components is that such an approach results in a rather abstract comparison and it does not solve the questions from chapter 1 which are relevant to open source software. Regarding open source the consequence is that it is necessary to not look at a component as a general building block but as an **open source component (OSC)**.

Like Crnkovic, Sentilles, Vulgarakis, and Chaudron (2011) defined components relative to their respective component models, the sensible way of defining OSC is to define the **open source component model (OSCM)** and make the definition of OSC implicit to the OSCM. Incidentally the Maven build system has many characteristics that can be attributed to a component model. Whether these characteristics apply to other build systems is out of the scope for this thesis.

An examination of how software that is developed with Maven fits into existing categorizations of component models, helps in the understanding of Maven's component model aspects. In the general case Lau and Wang (2007) propose an idealized component life cycle in order to derive a taxonomy of component models. They identify four categories – Design without repository, design with deposit-only repository, deployment with repository and design with repository. Figure 2 shows a component life cycle based on Maven in an analogue fashion to the one proposed by Lau and Wang (2007). During the builder phase Java code is written by developers. After that Maven can be used to compile the code into class files which by default are packaged into a jar file. During the repository phase dependency jars are placed in the local repository. These dependencies are defined in a Maven **project object model (POM)** file. Maven resolves transitive dependencies and downloads all the required dependencies from a remote Maven repository if the respective dependencies cannot be found in the local repository (Apache Software Foundation, 2018c). Note that dependencies

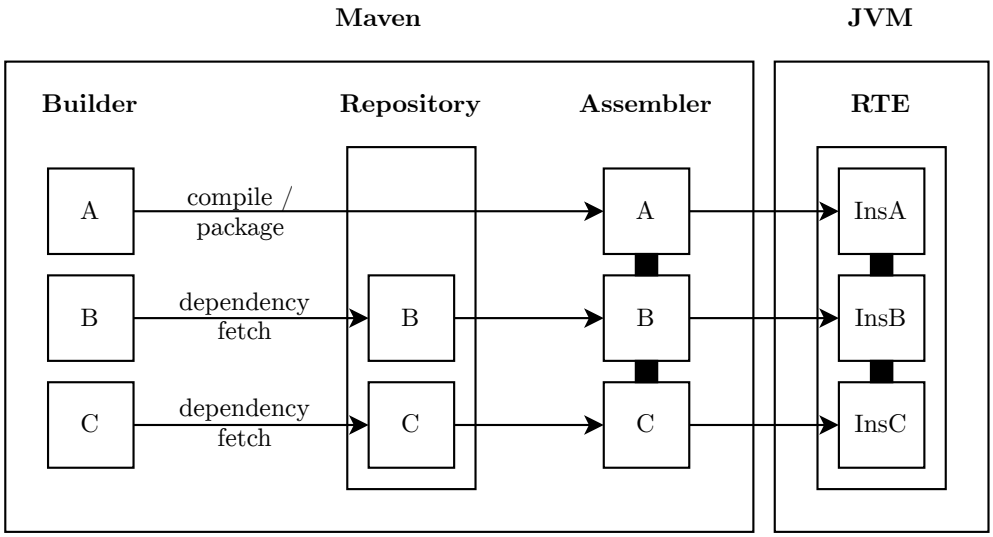


**Figure 2:** Composition with Maven in an idealized component lifecycle.

are treated the same way as locally installed components. Then the repository phase is followed by an assembler phase. Typically, there is no need for Java to be assembled in the sense of native software but Maven provides an optional step using plugins like the Apache Maven Assembly Plugin that packages the main jar and all its dependencies in a single jar file (Apache Software Foundation, 2017a). Components are then passed to a run-time environment (RTE) – in this case the Java Virtual Machine (JVM). The way a component's classes are loaded by the JVM is highlighted more closely in section 3.2.3.

After adapting a typical Maven lifecycle to the idealized component lifecycle proposed by Lau and Wang (2007), it seems that Maven fits best into the deployment with repository category. Note that Maven deviates from the categorization in the sense that locally packaged components are not necessarily installed in the local repository as shown in figure 2. Nevertheless, keeping in mind that a component model consists of rules for component composition and interaction, this means that composition of components is restricted to the assembler phase using something like the Maven assembler plugin. Consequently, multiple components can be assembled to a single jar file which may in turn result in a new component. Interaction between components is restricted by the run-time environment which is the JVM. Any interactions that are allowed by the JVM are allowed for components. This means that components are not required to interact using standardized interfaces but may sometimes be more similar to software modules where interaction is only restricted by the built-in visibility of classes. This may change with the modification of components to work on Java 9 as a key feature of Java 9 is Jigsaw which lets developers properly encapsulate the classes of a component while allowing access to intended classes only from outside the component

(Oracle America, Inc., 2017).

In conclusion the properties of Maven components can be transformed into rules to formulate a component model:

1. Components are singular files that can be loaded by the JVM.
2. Components are only composed during the Assembler phase of the idealized component life cylce.
3. Interaction between components is restricted by the JVM.

In order to define an OSCM this model has to be extended by the *open source aspect* of components. The Open Source Initiative suggests in their definition of open source that as long as the used license ensures that a product complies to the rules stated in their definition of open source, the product is open source (Open Source Initiative, 2007). As trivial as it may sound an OSC is therefore licensed using open source licenses.

Components that comply to the rules imposed by the OSCM are OSCs. Notably, it is possible that after composition the resulting component no longer complies to the rules. This happens when for example some component of a composition is a proprietary component that uses a number of OSCs. Provided the OSCs licenses allow such usage, the composition may be legal but the entirety may not be licensed using an open source license anymore. In such a case the result of the composition is simply called an assembly (Crnkovic et al., 2011). It is in turn not an OSC. But as long as such assemblies comply to the other rules, they will still be considered as components throughout this thesis.

## 3.2 Crawler design

Now that a basic understanding of what a component is has been established, to fulfill requirement **F01** a crawler basis has to be designed. Considerations from section 3.1 that are of relevance to this task are:

- **C1:** Dependencies which are resolved by a build system can be considered as components. If they are licensed using an open source license, they can be considered OSCs.
- **C2:** A component has **one** corresponding file which is considered the component's artifact.

These considerations are realized by the product model which is provided to the crawler module – as mentioned in section 1.2. Figure 3 shows a diagram of what it generally consists of. The top-level is called *Project*. It corresponds to a project

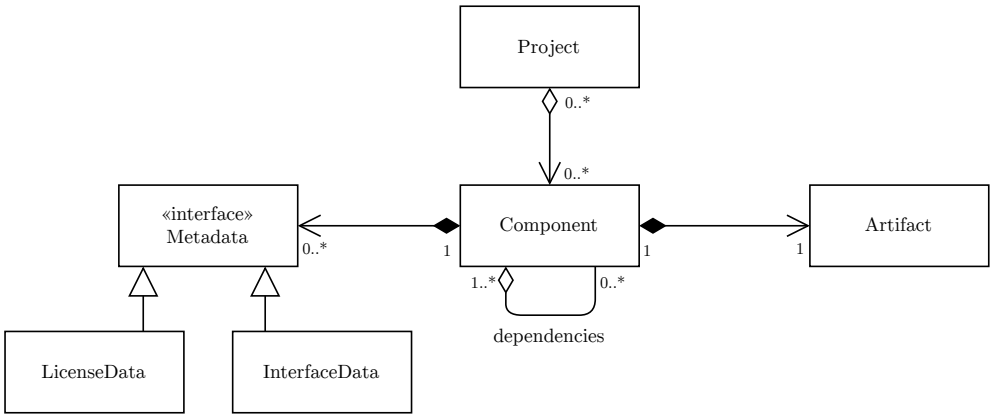


**Figure 3:** The generic product model.

that is provided as input to the crawler application and whose dependencies are to be determined. A *Project* has a number of root components. In the simplest case a *Project* has only one root component. In the case of Maven multi-module projects for example it has more than one.

The centerpieces of the model are *Components*. As required by consideration **C2** a *Component* has exactly one *Artifact*. In accordance to **C1** dependencies are modeled as a relationship between *Components*. A component can have an arbitrary number of dependencies and a dependency can be required by an arbitrary number of components as long as it is required by at least one. If a dependency is never required, then it is not part of a project. A *Component* can have an arbitrary number of *Metadata* objects. This accounts for the uncertainty of what information different crawler implementations may be able to provide. Examples for possible metadata are *LicenseData* and *InterfaceData* but it is likely that the product model will be extended by more metadata in the future.

A crawler implementation will produce a *Project* object based on the product model regardless of the implementation details of the crawler. This makes it fairly easy to implement crawlers using the strategy pattern as defined by Gamma, Helm, Johnson, and Vlissides (1994). The strategy pattern is categorized as a behavioral design pattern which splits algorithms from their context and makes the algorithms interchangeable. Figure 4 shows how it is realized in this thesis. A strategy must implement the *Crawler* interface. The interface provides a method to start a crawl run and to retrieve the *Project* result. A concrete implementation would be the *MavenCrawler* which is explained in more detail in section 3.2.1. The *CrawlContext* contains a strategy and is the handle for starting a crawl run and retrieving the result. The *CrawlContext* also contains crawler independent input parameters like the path to the project to be analyzed or a working directory for the strategy. The manner in which the *CrawlContext* is used, remains the same regardless of the strategy.

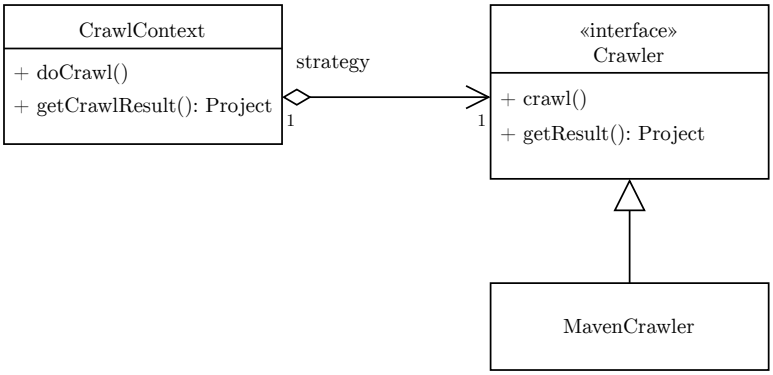


**Figure 4:** Crawler design with strategy pattern.

The fact that any new strategy only needs to implement the *Crawler* interface, is already enough to satisfy the requirement **Q01** but Gamma et al. (1994) suggest that the strategy pattern has even more benefits that apply to this thesis:

- The *Crawler* interface can implement common functionality for all crawler implementations. This may become relevant in the future when crawler implementations may need to run concurrently for example.
- Since the actual behavior is implemented in strategies, the *CrawlContext's* complexity stays manageable even if the context will be extended by a lot in the future.
- The use of strategies eliminates a lot of conditional statements. If crawlers were implemented in a single class, there would be a lot of conditions that check for specific build systems of the project to be analyzed.

Gamma et al. (1994) also state some disadvantages of the strategy pattern. Here are the disadvantages and reasons why none of them have an impact on this thesis:

- **Strategies are only suitable if the differences in implementation of strategies are meaningful to clients. Otherwise a client (or program) that uses the *CrawlContext* does not know what strategy to choose.** Since the strategy pattern is primarily used to account for different build systems of projects to be analyzed, this disadvantage does not apply. The reason for that is that it can be determined fairly easily what build system is used.
- **The strategy pattern sometimes causes unnecessary overhead since the *CrawlContext* provides each strategy with the same information regardless of whether the strategy needs all that information to function or not.** This may become a problem if the *CrawlContext*

becomes more complex in the future but can be disregarded in this thesis because the *CrawlContext* is only designed to work with the *MavenCrawler* for now. Naturally, all the necessary information passed by the context is needed by the strategy here.

- **The strategy pattern causes the creation of more objects.** In some cases this can be quite an extensive overhead but in this thesis the strategy pattern only causes one additional object to be created – the *CrawlContext*.

All this makes the strategy pattern very suitable for the crawler design but that does not mean that other patterns were not considered. For instance, if the crawler is a process that returns products, the entire process can be thought of as the building of a complex object. In that case the builder pattern (Gamma et al., 1994) would be suitable. It provides a common interface for the building process which makes concrete builders interchangeable. This would be similar to different strategy implementations. But the crucial difference is that each concrete builder returns different products which do not share a common interface according to Gamma et al. (1994). This does not serve the requirement **F01** since a crawler implementation always returns the same kind of product. Also, the builder pattern imposes methods for the creation of product parts which may provide better structuring of some building processes. But this leaves a concrete builder with less flexibility. This might make the implementation of a concrete builder more difficult since the underlying problem of analyzing projects may be vastly different depending on the given build system. Other than that the strategy pattern simply provides more advantages for the crawler than the builder pattern. This leaves the strategy pattern as the clear choice.

### 3.2.1 Maven crawler design

To satisfy requirement **F04** a crawler has to be designed that works on Maven projects. Since it has been established in section 3.2 that crawlers are implemented as strategies, the design of the strategy can be handled independently from the rest of the application as long as it implements the *Crawler* interface. Another useful property of the strategy pattern comes in handy during the design of the strategy. After all the pattern enables the encapsulation of any internal models used by the strategy (Gamma et al., 1994). This means that it is possible to define a Maven specific product model that possibly contains information that is not needed by the general product model. This is especially useful in the context of this thesis because the development of the strategy and the general product model is happening in parallel.

The general product model and the Maven model are quite similar even though the terminology of the Maven model is inspired by Crnkovic et al. (2011). The

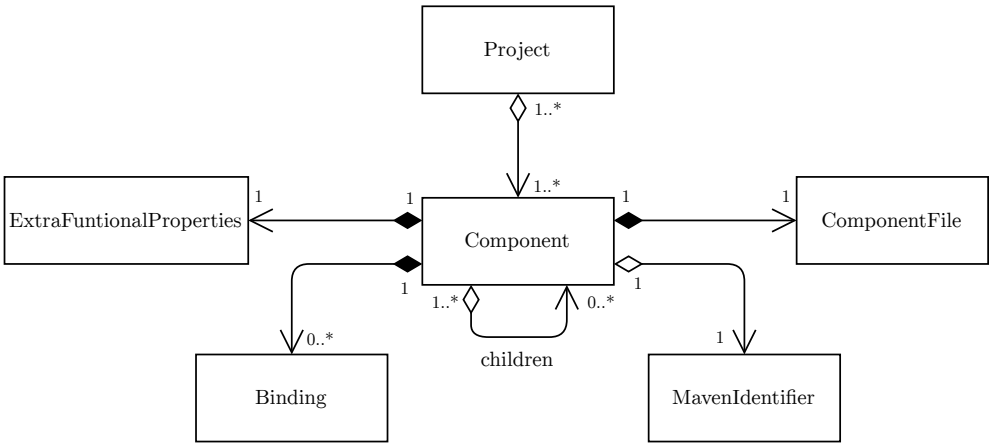


**Figure 5:** The model used by the Maven crawler.

relevant difference is how metadata is incorporated in the Maven model. In the context of the Maven model it is not necessary to account for the uncertainty of what information the strategy is able to provide as opposed to the general product model. Therefore instead of having a variable number of objects implementing the *Metadata* interface, it is sufficient for each *Component* to have an *ExtraFuntionalProperties* object. These extra functional properties contain information about Maven modules, open source licenses and other relevant metadata. What is realized as *InterfaceData* of the general product model is called *Binding* in the Maven model. Another specialty of the Maven model is that each *Component* has a *MavenIdentifier*. This is due to the special property of Maven components being identified by their group id, artifact id, version and packaging type. This is referred to as **Maven coordinates** in the Maven POM reference (Apache Software Foundation, 2018c).

As mentioned before the Maven model is the internal model of the *MavenCrawler* strategy. The strategy does a number of procedures sequentially. So to keep the strategy manageable it is split into a number of so called *CrawlProcessors*[1]. *CrawlProcessors* can do arbitrary procedures but always take *Project* objects as input. The output can be any type but is usually a Maven model *Project*. That way they can be piped sequentially. Note that it is the programmers responsibility to make sure that the output of a *CrawlProcessor* is compatible with the input of its subsequent *CrawlProcessor*. Figure 6 shows all the *CrawlProcessors* as well as the order in which they are executed:

- The **ResourceProcessor** looks at the input path provided by the *CrawlContext* to determine whether the input project is a valid Maven project.
- The **PomProcessor** uses the input projects POM file to determine depen-

[1]Not to be confused with the processors module mentioned in section 1.2

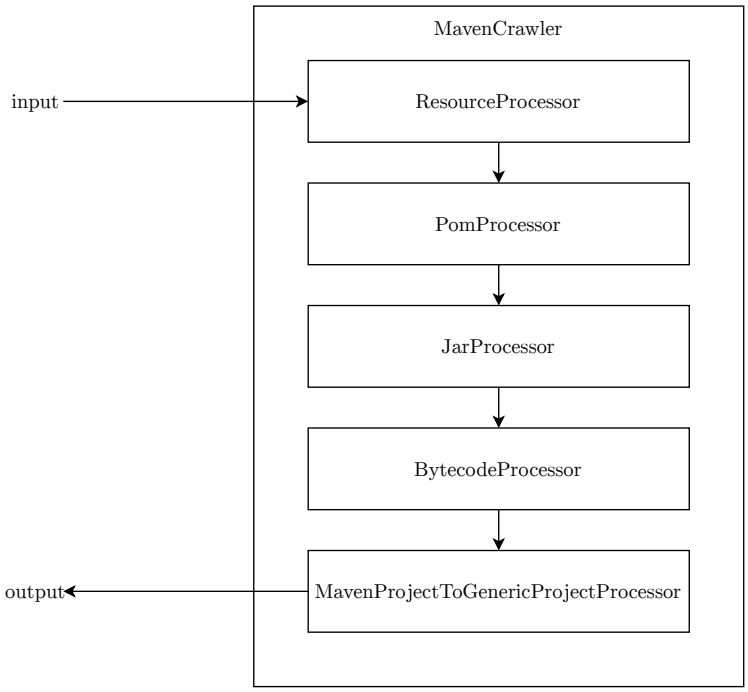


**Figure 6:** Internal procedures of MavenCrawler.

dencies and builds an initial *Project* object.

- The **JarProcessor** looks for the artifacts corresponding to each *Component*. In the course of that the *JarProcessor* usually builds the project to be analyzed in order to get the resulting artifacts.
- The **BytecodeProcessor** analyzes the artifact of each respective component and determines the classes/interfaces that it requires from other components.
- The **MavenProjectToGenericProjectProcessor** is responsible for converting the internal Maven model to the generic product model that the strategy is required to return.

### 3.2.2 Partial crawling

Since *Projects* are not trivial data structures it is necessary to design a sensible algorithm in order to satisfy requirement **F02**. While designing such an algorithm, it is crucial to be aware of the fact that dependencies in projects may overlap. So when overlapping dependencies are found while combining projects, the relationships between *Component* objects need to be redirected. Another aspect in the case of Maven dependencies is that dependencies in a project are

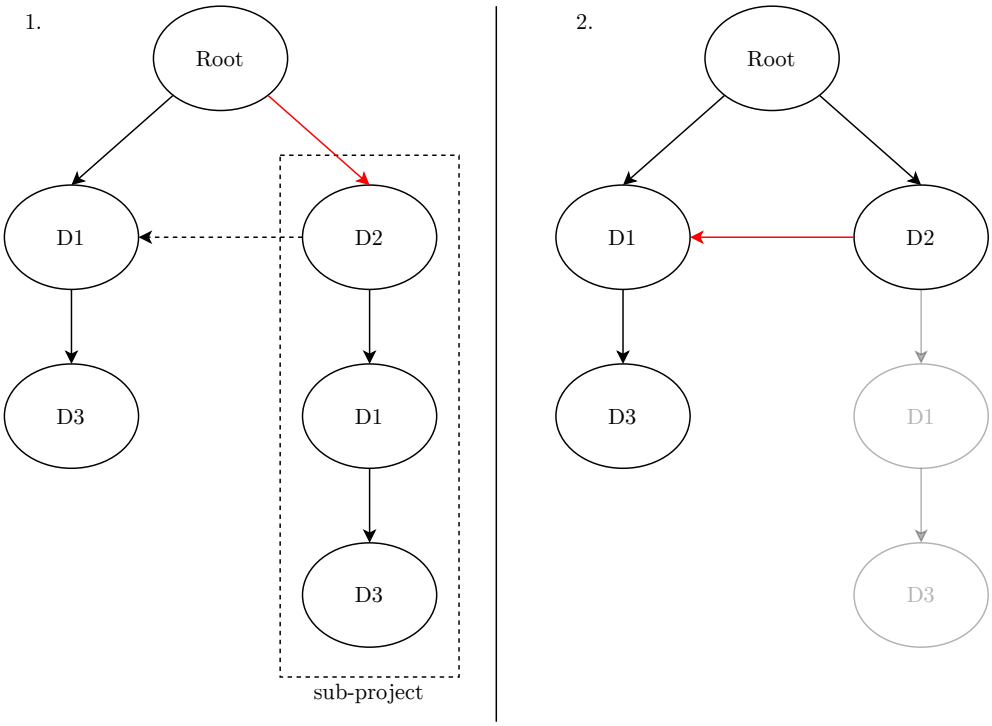


**Figure 7:** Appending a sub-project.

uniquely identified by their coordinates as mentioned in section 3.2.1 and therefore their transitive dependencies can always be deterministicly resolved. This means that if overlapping dependencies are found, then there is no need to look at the respective dependency's dependencies[2].

Figure 7 shows an example of how such an algorithm would operate on a dependency tree:

1. The subproject consisting of *D2, D1 and D3* is appended to the *Root* node of the project consisting of *Root, D1 and D3*. Since *D2* can not be found in the project the node is added as a dependency of *Root*. Subsequently, the algorithm checks whether the dependencies of *D2* can be found in the project. In fact *D2* has a common dependency (*D1*) with the project.

2. The dependency of *D2* is redirected to the project's *D1*. Since it is assumed that the dependencies of *D1* are the same for all partial projects, there is no need to further examine the dependencies of *D1*. The subproject's *D1 and D3* can be deleted since *D2* no longer has any reference to them.

Assuming that a *Component* in a *Project* can be found in constant time, the algorithm has a worst case complexity of $O(n^2)$ where $n$ is the number of nodes

[2]Dependencies' dependencies will also be called dependencies' *children* throughout this thesis.

in the subproject. But the algorithm will usually run substantially faster because the worst case assumes a very unlikely subproject and partial projects often contain overlapping dependencies like utility libraries and such. The details of the algorithm will be further examined in section 4.1.2.

### 3.2.3 Component Relationships

In terms of open source license compliance it is important to consider in what manner software components interact to form a complex program. The reason for that is that certain open source licenses impose obligations (as mentioned in chapter 1) on a licensee if a component interacts with the licensed component in a certain way. A software component licensed under the GNU Lesser General Public License (LGPL) for example does not become viral – meaning it does not propagate its license to other software components – if the components are dynamically linked (Free Software Foundation, Inc., 2007c). Other licenses like the GNU Affero General Public License (AGPL) do become viral even if software components only interact using web calls (Free Software Foundation, Inc., 2007a).

Therefore, requirement **F05** states that the crawler tool must support the identification of such types of interactions between components which are also called component relationships throughout this thesis. But since this thesis is focused on Java using the Maven built system, it is necessary to examine how linking is accomplished in the JVM which in turn requires a deeper delve into the inner workings of the Java language.

Figure 8 shows a simplified life cycle of java classes. Initially there are Java files that simply contain source code. The source code is then processed by the Java

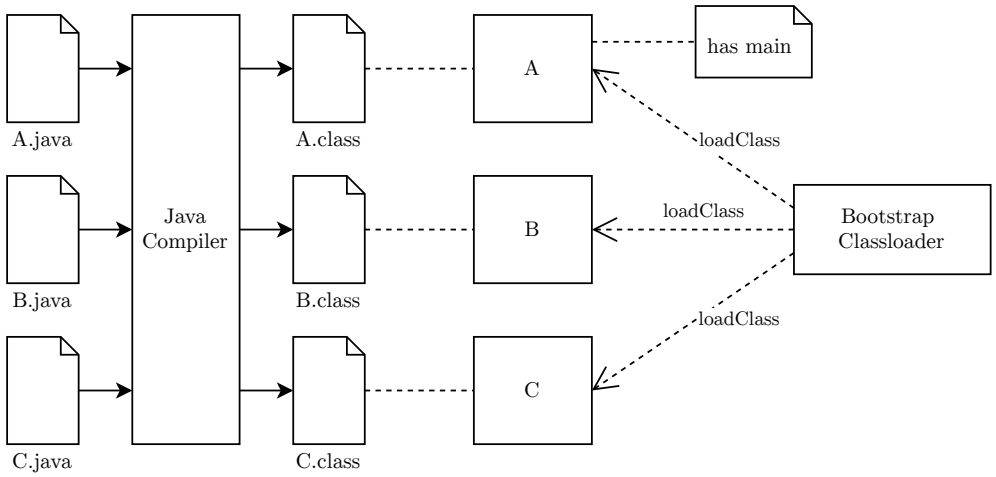


**Figure 8:** Java from source code to loaded classes.

compiler which produces class files. These class files contain Java bytecode which represents an encoded data structure containing all the class specific information needed by the JVM to run the classes. This for instance contains arrays of fields, methods as well as symbolic class references among other elements in a class's constant pool. Methods in turn contain JVM compatible instructions that are executed when the methods are run. This can be thought of as the assembly of a compiled C file.

The JVM is dependent on class loaders to resolve class files. There are two types of class loaders. The first type is only the default bootstrap class loader and the second type is user-defined class loaders. Usually the default class loader loads the initial class ($A$) containing the main method which is the entry point to the program. Eventually some class loader resolves other classes ($B$ and $C$). Resolution takes place when a JVM instruction makes a symbolic reference to the run-time constant pool (Oracle America, Inc., 2015). In the case of Figure 8 class $A$ is loaded by the bootstrap class loader. While running the main method some JVM instruction references class $B$. The class loader used to load class $A$ is then used to load class $B$. This is the case even if some other class loader than the bootstrap class loader were used. The same happens with class $C$. This means that classes are always dynamically loaded when Java programs run in the JVM. Even if class files are packaged as jar files, the mechanism remains the same since jar files are only zip archives containing class files. Generally though, it can not be determined how classes are loaded since there is always the possibility that user-defined class loaders are used. This makes a nuanced identification of component relationships difficult. Nevertheless, it is assumed throughout this thesis that Java classes are dynamically linked and consequently that static linkage is impossible using Java.

This leaves requirement **F05** with the problem of web calls. There is no singular way of making web calls in Java. If one were to analyze software artifacts regarding this aspect, one would have to consider a multitude of web call possibilities. This would be quite an effort considering that an easier approach is to simply run the software that is to be analyzed and listen for outgoing web calls. But such an approach is out of the scope for this thesis since the crawler tool is based on the analysis of software artifacts. That being stated, requirement **F05** can only be partially satisfied in the sense that components written in Java will be identified as dynamically linked components. Whether static and dynamic linkage in other programming languages can be detected is to be determined in future work.

# 4 Implementation

Since the crawler application analyzes Maven projects it makes sense that the crawler application itself is implemented using Maven – especially, because the application requires a number of dependencies. Among them are dependencies from the Apache Maven project. The core dependency is the Apache Maven Artifact Resolver (Apache Software Foundation, 2018a) which is used for dependency resolution. See appendix A for a full list of dependencies required by the crawler application. Except for one dependency all the dependencies in appendix A are licensed using the Apache 2.0 license (Apache Software Foundation, 2004) which is a fairly permissive license. The one dependency not licensed under the Apache license is the Logback library which is dual-licensed under the Eclipse Public License (EPL) and the LGPL (Quality Open Software, 2018). Since Logback is only used for logging it would make sense to choose the library as a LGPL licensed component. In that case Logback can be used without major obligations like having to propagate the LGPL to the entire work. That is because it is dynamically linked which is always the case for Java as established in section 3.2.3.

Now that general implementation considerations have been addressed the following will dive into important implementation details starting with the Maven crawler implementation in section 4.1. After that the implementation of the model exports is highlighted in section 4.2. Subsequently, the implementation chapter is concluded by some details about the command-line interface for the crawler application in section 4.3.

## 4.1 Maven crawler implementation

This section focuses on two particular aspects of the *MavenCrawler* implementation. Section 4.1.1 highlights dependency resolution and section 4.1.2 focuses on the algorithm used for appending a subproject mentioned in section 3.2.2. Other aspects handled by the *MavenCrawler* like the conversion of the internal Maven model to the general product model are not attended to in detail. It is worth

noting though that requirement **F06** is satisfied during this conversion[1]. There the product model receives a timestamp using *Instant.now()*.

### 4.1.1 Dependency resolution

As mentioned in section 3.2.1 the *MavenCrawler* is divided in five subtasks called *CrawlProcessors* – the **ResourceProcessor**, **PomProcessor**, **JarProcessor**, **BytecodeProcessor** and **MavenProjectToGenericProjectProcessor**. The first and last processor are respectively responsible for transforming the input and output of the *MavenCrawler*. But the dependency resolution is a three-step process which corresponds to the **PomProcessor**, **JarProcessor** and **BytecodeProcessor**. Each produces a partial result which is extended by its subsequent *CrawlProcessor*.

#### Dependency tree resolution

The building of the dependency tree is handled by the **PomProcessor**. It uses the Maven Artifact Resolver[2] (Apache Software Foundation, 2018a). The artifact resolver internally produces a dependency graph – that is a graph of all the dependencies required directly and transitively containing dependency conflicts. Dependency conflicts occur when two or more dependencies require the same dependency but in different versions. In that case Maven generally attempts to resolve these conflicts by determining one of the conflicting versions to be used. After conflict resolution the artifact resolver returns a dependency tree (Bentmann & Irawan, 2014).

For that to work the *MavenCrawler* needs to be initialized with a valid path for the local Maven repository. The local repository is a directory containing artifacts of dependencies. The artifact resolver will always use artifacts from the local repository first before downloading them from a remote repository. The path to the local repository is passed to the **PomProcessor** by the *MavenCrawler*. The first step for the **PomProcessor** is to put the POM file of the input project inside the local repository. So when the artifact resolver starts resolution from this root POM, it will not try to download the file from a remote repository. Among other information, POM files contain all the required dependencies of a component. The artifact resolver iterates over these files and collects dependencies as well as downloads POM files of dependencies from remote repositories in case they are not present in the local repository.

[1] Handled by the *MavenProjectToGenericProjectProcessor*

[2] Formerly known as Eclipse Aether.

The artifact resolver returns a dependency tree that can be traversed using a visitor. The visitor design pattern allows to apply new operations to each of the elements of a data structure without having to change the classes of the data structure (Gamma et al., 1994). In the case of the **PomProcessor** the new operation is a visitor that traverses the dependency graph and builds the internal Maven model. This visitor is called *MavenDependencyVisitor* and implements the interface *DependencyVisitor* in listing 4.1. The interface allows detailed control over the traversing of the structure. If the *visitEnter* method returns false, then the visitor will not visit the child nodes of the current node. Similarly, if *visitLeave* returns false, the visitor will not visit further siblings of the current node. During the traversing of each node the corresponding POM file is parsed in order to retrieve wanted metadata for the product model. For parsing the POM files the class *MavenXpp3Reader* is used which is provided by the crawler application dependency *maven-model*.

```
1 package org.eclipse.aether.graph;

3 public interface DependencyVisitor {
      boolean visitEnter(DependencyNode node);
5
      boolean visitLeave(DependencyNode node);
7 }
```

**Listing 4.1:** DependencyVisitor interface.

Figure 9 shows the dependency tree's visualization of the crawler application itself. It holds limited information about its dependencies because complex relationships among dependencies are eliminated by conflict resolution. Nevertheless, the dependency tree provides interesting information. It can be easily seen where certain dependencies are introduced from. For instance, since the red root node is representing the crawler application, dependencies of degree one are directly required by it. Another interesting aspect is that the maximum degree of dependencies is three with five dependencies. Most of these are transitively introduced by the *maven-aether-provider* dependency which is the implementation of Maven-like resolution needed by the artifact resolver. The dependency is introduced by the crawler application instead of the artifact resolver implementation because it is necessary to wire the provider to the artifact resolver inside the crawler application. But if one wanted to reduce the number of dependencies, he could easily determine with this dependency tree that the *maven-aether-provider* dependency is a good starting point for optimization.

### Artifact identification

The second step in dependency resolution is to identify the artifacts of each component. For now the only supported artifact type is *Jar*. Therefore, this process

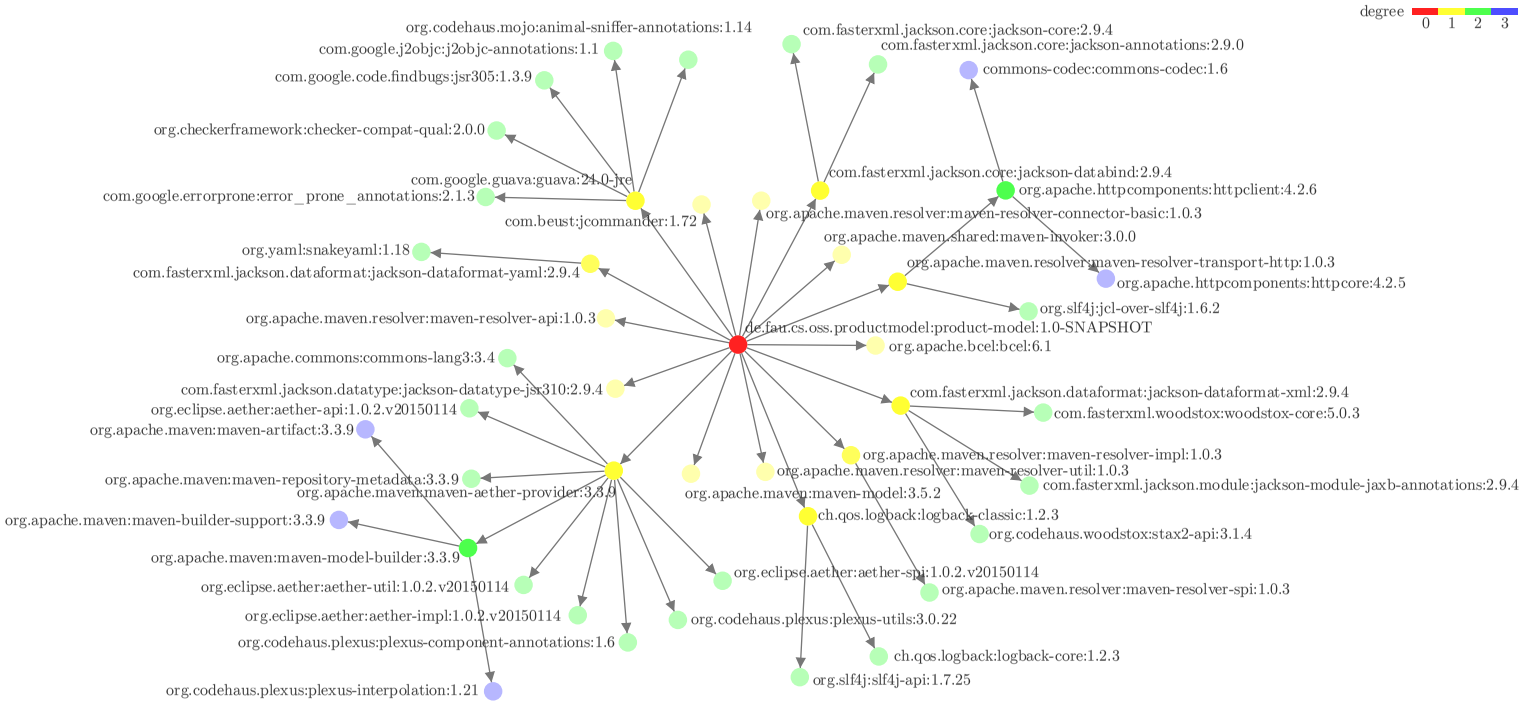


**Figure 9:** Crawler application dependency tree.

is handled by the **JarProcessor**. For each jar-component determined by the previous processor it will look for the corresponding Jar-Files in the local repository. Like with the **PomProcessor** the path to the local repository is provided by the *MavenCrawler* to the **JarProcessor**. A *Project* provided as input for the **JarProcessor** must always have a root component. Such a component represents the entire software project to be analyzed. Typically the artifact of the root component is not present in the local repository. In such a case the **JarProcessor** will attempt to build the artifact. Since the project to be analyzed is a Maven project a build will automatically download all the dependencies to the local repository.

Generally, for building the artifact the Apache Maven Invoker is used. It requires a local Maven installation to be present. The invoker will look for a Maven installation in the *maven.home* system property or the *M2_HOME* environment variable (Apache Software Foundation, 2017b). This issue is further addressed in section 4.3. Alternatively, the **JarProcessor** can use a Maven wrapper script if the *CrawlContext* specifies a build script[3]. The consequence of this is that the **JarProcessor** will not require a local Maven installation but will instead use a wrapped Maven script. If the build fails, there is no artifact to associate the root component with and therefore the **JarProcessor** will throw an exception.

Regardless of which method is used for building, the *install* command of the Maven default lifecycle is issued. Maven has the characteristic that whenever a command is issued, the previous phases of the default lifecycle are executed first (Apache Software Foundation, 2018b). That is why the *install* command will result in Maven producing artifacts (*package* phase) even though the command is only responsible of placing the artifacts in the local repository. The following phases are executed when issuing an *install*:

1. validate
2. compile
3. test
4. package
5. verify
6. install

The consequence is that the build can be a lengthy process because prior to installing the artifacts to the local repository, Maven will execute unit tests (*test* phase) as well as integration tests (*verify* phase). To circumvent this problem the system property *skipTests* is set to true while issuing the *install* command. This system property is honored by the Surefire plugin which is typically used for unit

[3]This would be determined in the *ResourceProcessor*

tests as well as the Failsafe plugin which is typically used for integration tests (Apache Software Foundation, 2018d). If tests are run as part of other plugins in the build process, they will run nevertheless.

#### Relationship identification

Finally, the **BytecodeProcessor** determines relationships between components. To do that it needs to analyze the artifacts previously determined by the **JarProcessor**. As mentioned in section 3.2.3 jar-files are nothing more than zip archives. Therefore, the first thing to do is unpack the class files of all artifacts. Each class file is associated with a component. Subsequently, the **BytecodeProcessor** traverses through the *Project* passed from the **JarProcessor** and looks at all the symbolic references to other classes in each class file's constant pool. To get the constant pool of a class file the Apache Commons Byte Code Engineering Library is used (Apache Software Foundation, 2017c).

Since the constant pool also contains references to Java language specific classes which are not particularly interesting in the context of this thesis, the constant pool of each class file has to be filtered. For that it is sufficient to look at the fully qualified names of the references. Java language classes will have a fully qualified name that starts with "java/". Since apart from regular classes the only other types of references in the constant pool can be arrays, it is also necessary to filter all the references whose fully qualified names start with "[Ljava" (Oracle America, Inc., 2015). That leaves each constant pool with references to classes either inside the same component or to other components.

Once the references have been determined the **BytecodeProcessor** adds the respective *Bindings* to the *Project*. This excludes *Bindings* if they represent a relationship of a component to itself. Whenever a *Binding* is added to a *Project*, there is also a relationship added between the *Component* objects involved in the *Binding*. This results in components having dependencies that have previously been eliminated by conflict resolution in the **PomProcessor** *if* the dependencies are *real* – meaning that the classes of a declared dependency in a POM file are actually used.

Figure 10 shows a visualization of the resulting *Project* based on the crawler application's dependencies. Aside from looking more confusing it can be seen that the degree of some components has changed in comparison to the dependency tree shown in figure 9. This is due to the fact that the crawler application uses classes of transitive dependencies. This lowers the degree of the transitive dependencies even though they are not directly required by the crawler application. If one were to remove a direct dependency that causes a transitive dependency which is required by the actual application, it would cause unforeseen behavior. That

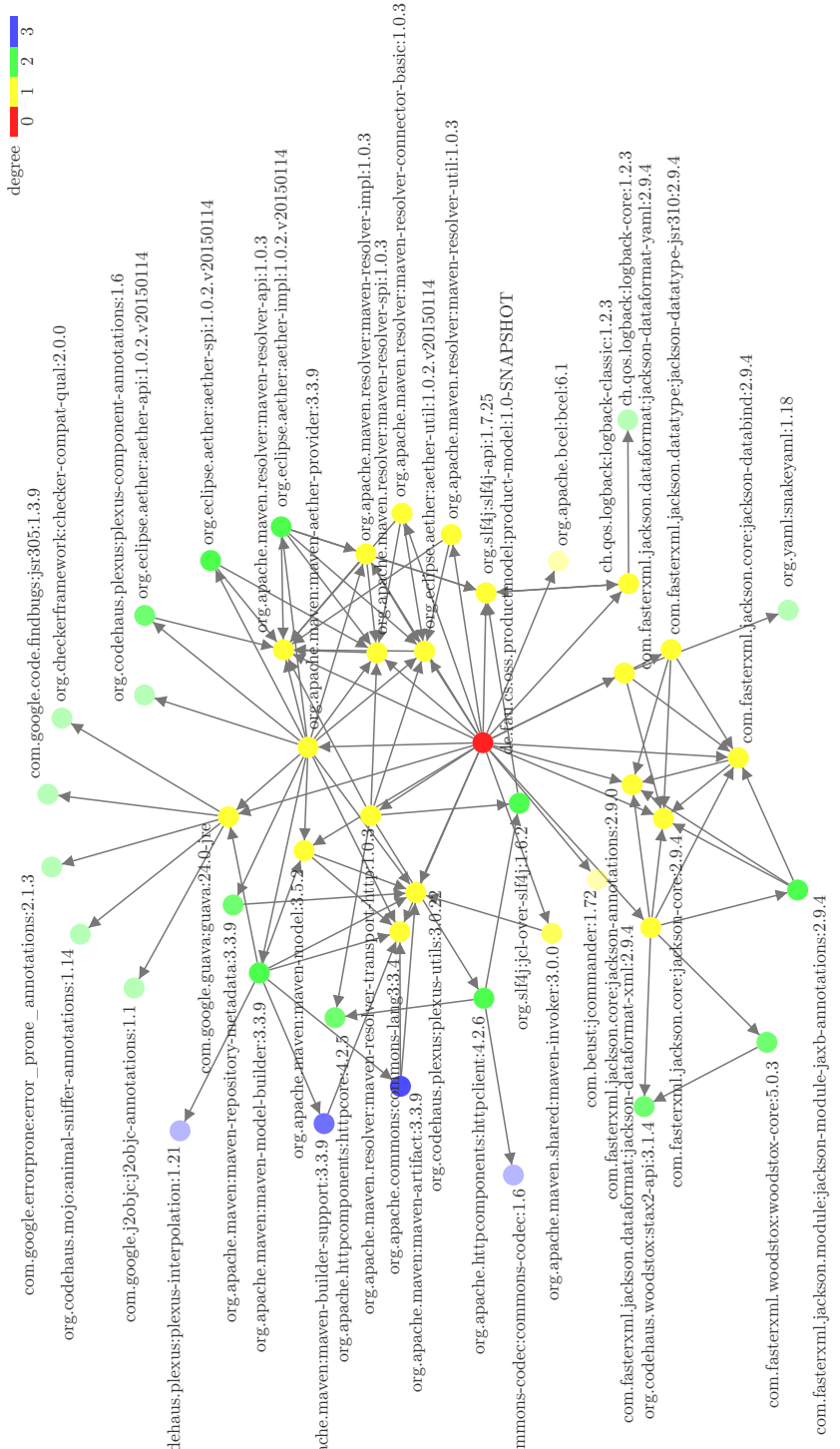


**Figure 10:** Crawler application dependency graph.

is why the analysis of the byte code and the reconstruction of dependencies is necessary.

### 4.1.2 Partial crawling implementation

Originally, the need for partial crawling was motivated by the outlook of different crawler implementations producing partial products that need to be put together. But it turns out that sometimes this is already required inside the *MavenCrawler*. When the project to be analyzed is a multi-module project, it is not possible to run the **PomProcessor** only from the project root because the dependencies of modules would not be considered that way. Therefore, for each module a dependency tree has to be determined and afterwards the partial dependency trees need to be combined. To accomplish this, the internal model's *Project* class implements an *appendProject* method as shown in listing 4.2.

A *Project* has a *HashMap* containing all the *Component* objects that are part of the dependency graph. This serves as a sort of index for the data structure. Each component's key is its Maven coordinate as a string[4]. That way components can be found in constant time using their key which improves the performance of the algorithm.

```
1  public void appendProject(String parentKey, Project subproject) {
       (...)
3      Component parent = components.get(parentKey);
       Component subprojectRoot = subproject.getRoot();
5      this.doAddComponent(subprojectRoot);
       parent.getChildren().put(
7        subproject.rootNode,
         components.get(subproject.rootNode));
9
       relinkChildren(subprojectRoot);
11 }
```

**Listing 4.2:** appendProject method.

The *appendProject* method takes the *Project* to be appended[5] and the key for *Component* to which the *Project* shall be appended. The general idea of the algorithm is to stop whenever a component is already part of the *Project*. The exception to this idea is the start of the algorithm. For the first component it is necessary to always look at its children because if the first component is the root of a Maven module, it is possible that the component is part of the *Project* but its children could not have been determined.

[4]Maven coordinates are presented as "groupId:artifactId:version"

[5]Also called *subproject* in this section

The dependencies of the project to be appended are redirected in the *relinkChildren* method. The first thing done in that method is to put the component whose children are to be checked in a *HashSet* containing all the components that have already been visited while iterating over the children. This avoids infinite iterations when circular dependencies occur. Then the subproject's root and each of its children will be paired and put into a buffer. This is followed by the actual iteration over the children. The loop will always take the parent-child pair at the index *zero* of the buffer while there are any left and do one of the following two operations after adding the child to the *visited* set:

- If the child is already part of the *Project* to which the subproject is being appended, then redirect the dependency. In this case the children of the dependency do not need to be checked.
- If the previous is not the case, then the child is added to the *Project* as a new dependency and the children of the child are added as parent-child pairs to the buffer provided that the children are not in the *visited* set yet.

At the end of each iteration the buffer entry at index *zero* is removed from the buffer. That way the loop will definitely come to a halt at some point. The best case complexity of the algorithm is $o(m)$ where $m$ is the number of the root component's children of the subproject. The worst case complexity is $O(n^2)$ where $n$ is the number of components in the subproject. It is quadratic because there is a second loop inside the main iteration when the parent-child pairs are added to the buffer. But the algorithm usually runs a lot faster than the worst case because related projects often have overlapping dependencies like utility libraries and such.

## 4.2 Model export

The model export becomes quite simple when using the *Jackson* library for converting the model to serialized representations. *Jackson* is a widely used library for JSON serialization and parsing (FasterXML, LLC., 2018). Most objects can be serialized and parsed out of the box but sometimes classes have to be implemented using *Jackson* annotations.

The relevant annotations used for serialization in the product model are *JsonManagedReference* and *JsonBackReference*. These annotations are used to indicate parent/child relationships between nodes (FasterXML, LLC., 2016). This is used for component relationships.

Although *Jackson* is originally intended for JSON representations, there are extensions for *Jackson* that enable the framework to produce XML (*jackson-*

*dataformat-xml*) and YAML (*jackson-dataformat-yaml*) data formats without having to implement a data model using different annotations. These extensions simply override the *ObjectMapper* class which is responsible for serialization and parsing of data.

Appendix B shows an excerpt of an exported JSON representation. It demonstrates how dependencies as well as their metadata translate into serialized representations. Note that if certain string values can not be determined during a crawl run, the export will display *none* as the value. This is the case for "homepageUrl" in appendix B.

## 4.3 Command-line interface

As mentioned in section 1.2 the motivation for the crawler application is to be used in existing tool chains. So it makes sense to provide a command-line interface (CLI) for that purpose. That way a user can easily combine the application with existing tools, e.g. in a build script. An alternative would be to provide the application via some web application programming interface (API). But that would require a web server to be implemented which is an substantial effort. Since there is no special requirement for a web API, the much simpler alternative of providing a CLI is chosen. To build the CLI the library *JCommander* is used. It makes parsing command-line arguments very easy by adding annotations to the respective parameters (Beust, 2017).

The supported options are shown in listing 4.3. The CLI always requires the *-c* flag to be set because it is the only operation that the tool supports at the moment. In the future there will be more options to choose from. Another interesting option is the *-m* option. This is required to be set if a local maven installation is used to build the project to be analyzed. As mentioned in section 4.1.1, the **JarProcessor** uses the Apache Maven Invoker for that. It requires either the *maven.home* system property or the *M2_HOME* environment variable to be set. Therefore, the CLI makes sure that the *maven.home* system property is set by calling the *System.setProperty* method before starting a crawl run. That way a user will not have to make sure that the *M2_HOME* environment variable is set prior to starting the crawler application.

```
1 Usage: <main class> [options]
    Options:
3     --crawl, -c
        Do a crawl run.
5     --export, -e
        File path to which model should be exported.
7       SUPPORTS: json, xml, yaml
      --input-dir, -i
```

```
9         Input directory for the crawler.
      --output-dir, -o
11        Directory where produced files are stored.
      --maven, -m
13        Path to maven installation base directory.
```

**Listing 4.3:** Command-line interface usage description.

### 4.3.1 Containerization with Docker

It has become apparent that the crawler application has two external dependencies. The first is a Java RTE and the second is a Maven installation. The latter is optional in some cases. But since the application will support more crawler implementations in the future, it is very likely that the number of external dependencies is going to rise. This makes providing a stable system difficult on which the crawler application reliably runs. A solution for this problem is to run the application inside a container. To date one of the most popular containerization technologies is *Docker*. Docker containers can be easily built using so called *Dockerfiles* which are basically blueprints for the containers. In such a file the respective versions of external dependencies can be statically defined which attributes to a stable environment. Additionally, docker images can be hosted in docker repositories like *Docker Hub*[6] which makes deployment very convenient.

Such a docker image for the crawler application can be built using the Dockerfile depicted in listing 4.4. The image is built from the minimal Alpine Linux base image which is only roughly 8 MB large (Alpine Linux Development Team, 2017). The only additional programs installed are a openJDK8 and its RTE as the Java installation as well as Maven. For the Maven installation a specific version can be pinned in case the build of a project to be analyzed requires that. A user can also add additional dependencies needed for the build here.

```
1  FROM alpine:3.7

3  RUN apk add --update \
       openjdk8-jre \
5      openjdk8 \
       maven=3.5.2-r0
7
   COPY docker/maven/assets/start.sh /start.sh
9  COPY target/product-model-1.0-SNAPSHOT-jar-with-dependencies.jar
      /product-model.jar

11 ENTRYPOINT ["/start.sh"]
   VOLUME ["/project"]
```

[6] https://hub.docker.com/

```
13 VOLUME ["/output"]
```

**Listing 4.4:** Dockerfile for crawler application

The entry point for the Docker image is the *start.sh* script. It determines the installation directory of Maven inside the container and starts the crawler application with the appropriate command-line options. The only additional thing a user needs to do when running the Docker image, is to mount the input directory to the *"/project"* volume and the output directory to the *"/output"* volume.

# 5 Evaluation

As mentioned in section 2.9 the functionality of the application – meaning its functional requirements – is verified with tests. Even though the successful completion of tests is not a guarantee for correctness of the program, it is still a valid indicator. Section 5.1 highlights the evaluation of the functional requirements whereas section 5.2 focuses on the non-functional requirements. The chapter closes with other quality considerations which are not part of the original requirements in section 5.3.

## 5.1 Functional requirements

Each of the functional requirements is realized as part of a class or set of classes as highlighted in chapter 3 and 4. Therefore, it is sufficient to test all the relevant classes and their interactions to gain confidence in the correct realization of the requirements. As mentioned in section 2.9 the minimum threshold for confidence in the implementation of the functional requirements' correctness is a code line coverage of 100%. Table 5.1 shows the coverage of the top-level packages of the crawler application at the time of this writing. It can be seen that the *crawler* package as well as the *export* package have a coverage of 100% which are the packages that implement all the functional requirements.

| **Element** | **Class, %** | **Method, %** | **Line, %** |
|---|---|---|---|
| crawler | 100% (33/33) | 100% (259/259) | 100% (1277/1277) |
| export | 100% (5/5) | 100% (19/19) | 100% (68/68) |
| model | 100% (11/11) | 86% (80/93) | 85% (224/263) |

**Table 5.1:** Unit test coverage.

But this is not enough to verify that the interactions between the software's units is correct. That is why integration tests are run as well. The projects chosen

as input for the integration tests are *Bootique*[1] and *Hddiff*[2]. The functionality of these projects is not important in the context of the integration tests. They were simply chosen because they were sufficiently large but not so large that the integration tests would run too long to be practical. One of the challenges here is to reliably set up the integration test environment so that the tests produce deterministic results. This is accomplished using the Maven SCM Plugin which is a plugin that can do a variety of operations on source code management (SCM) systems (Apache Software Foundation, 2016). Listing 5.1 shows part of the SCM plugin configuration from the crawler applications POM file. It can be seen that the execution of the plugin is bound to the *pre-integration-test* phase which is the appropriate phase to set up the integration test environment (Apache Software Foundation, 2018b).

```
<execution>
  <id>clone-hddiff</id>
  <goals>
    <goal>checkout</goal>
  </goals>
  <phase>pre-integration-test</phase>
  <configuration>
    <checkoutDirectory>
      ${project.basedir}/target/test-classes/hddiff
    </checkoutDirectory>
    <connectionType>connection</connectionType>
    <connectionUrl>
      scm:git:https://github.com/sweble/hddiff.git
    </connectionUrl>
    <scmVersionType>tag</scmVersionType>
    <scmVersion>hddiff-2.0.4</scmVersion>
  </configuration>
</execution>
```

**Listing 5.1:** Integration test project download.

The *model* package does no have a coverage of 100% because it contains the general product model which is still in development at the time of this writing. Also, the CLI has not been verified since it is most likely going to change significantly in the future. The following will explain how each functional requirement is verified in more detail.

### F01: Crawler based on product model

This requirement is the top-level functional requirement. It states that the software produces a product model based on a crawler implementation. Since the

[1] https://github.com/bootique/bootique
[2] https://github.com/sweble/hddiff

only crawler implementation available is the *MavenCrawler*, this requirement is dependent on requirement **F04**. But in general **F01** is about having a framework that is able to run different crawler implementations. Since the crawler implementations are realized as strategies they can be replaced by stubs during testing. Stubs are well suited in this case because the strategies return values that can be hard-coded in a stub (Meszaros, 2007). Error behavior of the implemented strategy pattern is also tested using a saboteur which is a variation of general stubs (Meszaros, 2007). In the case of the crawler application all stubs are injected using Java reflection.

#### F02: Partial crawling

The partial crawling requirement is only implemented for the internal Maven model used by the *MavenCrawler* because the general product model is still in development at the time of this writing. But in principle the algorithm described in section 3.2.2 should be applicable to the general product model. Regardless, this requirement can be entirely verified for the internal Maven model by unit tests since it is only an operation on the *Project* data structure. But it is also applied during integration tests since the operation is used in Maven multi-module projects which the test projects are.

#### F03: Exportable product model

The serialization of the product model is handled by the *Jackson* framework which is already well-tested. Therefore, it is only necessary to test the export regarding the completeness and correctness (e.g. correct timestamp format) of the exported model. This is accomplished with unit tests. Similarly to requirement **F01** the export's error behavior is tested using a saboteur.

#### F04: Maven project support

To verify this requirement the *MavenCrawler* strategy is tested. It is quite extensive and is therefore split into *CrawlProcessors*. Each of these *CrawlProcessors* is tested as individual units and the *MavenCrawler* itself is tested using stubs as *CrawlProcessors* similarly to **F01**. To ensure that these units work correctly, they are integrated all at once and tested during integration tests. This completes requirement **F01** as well.

**F05: Identify component relationships**

As determined in section 3.2.3 the crawler application can only recognize dynamic linking since static linking is strictly not possibly in Java and the detection of web calls is out of scope for this thesis. So this requirement is only partially met.

**F06: Product data version**

This requirement is almost trivially satisfied since it only involves setting a timestamp on the product model. This is verified using unit tests and can be seen in appendix B ("analyzeTime" field).

## 5.2 Non-functional requirements

As mentioned in section 3.2.1 requirement **Q01** is already satisfied through the crawler design since it only requires easy implementation of additional crawlers by having exactly one interface to implement. This leaves the evaluation of the application's reliability (requirement **Q02**). The threshold for minimal reliability was set at 50% of components recognized. This is verified through integration tests.

To produce deterministic results the projects used for integration tests are checked out using static tags from their respective git repositories. In the case of *Hddiff* for instance the tag *hddiff-2.0.4* is used. This is enough to run the *MavenCrawler* with the downloaded projects as input. But the results of the *MavenCrawler* can not be easily verified. To verify that the correct number of dependencies is recognized by the *MavenCrawler* it is necessary to find out what the correct number actually is. Unfortunately this is a manual process. The results of this manual process can be seen in the appendices C and D.

Since the *MavenCrawler* only determines non-optional dependencies this leaves *Hddiff's* dependencies with a total number of *25* and *Bootique's* dependencies with *16*. These numbers are confirmed by the integration tests. This means that the requirement is satisfied. But it may very well be that other projects produce subpar results. This is because the component identification process relies on only the *Maven* build system to handle dependencies. So if manually copied dependencies are used, they are not detected by the application. Similarly, dependencies introduced by different build systems in subprojects, are not detected, as well. These issues could be eliminated by future crawler implementations.

## 5.3 Other quality considerations

Throughout the development of the crawler application it has become apparent that some code quality requirements are implied instead of explicitly stated as non-functional requirements. One of these implied requirements is the wish to keep the code as simple as possible so that it stays maintainable as well as readable. To determine methods which might be too complex, the cyclomatic complexity introduced by McCabe (1976) was calculated. This was accomplished using the *MetricsReloaded*[3] plugin of the *Intellij IDEA* integrated development environment. If a method's cyclomatic complexity was greater than five, the method was refactored. The consequence is more easily readable code. The threshold of five was arbitrarily chosen.

[3]https://plugins.jetbrains.com/plugin/93-metricsreloaded

# 6 Conclusion

The underlying task of the crawler application is to enable solutions for difficult problems in using OSS as mentioned throughout chapter 1. One of these problems is that OSCs often have known security vulnerabilities. The crawler application can be used to find out whether affected components are used in software projects. This is accomplished by analyzing the byte code of all the Java classes used as mentioned in section 4.1.1. The resulting bindings can be found as metadata of type *Interface* in the exported product model. Using this information it is possible to look up whether known security vulnerabilities like the ones published by the NIST may apply to certain parts of software projects. This is already useful information but the identification process of interfaces may have limitations in some cases. This is due to the fact that classes may not be directly portrayed in byte code when for example custom class loaders are used. Also, it may be difficult to capture classes in byte code that have been instantiated using the Java reflection API.

Another open source specific issue mentioned in chapter 1 is the problem of detecting licensing conflicts. The crawler application is intended to provide all the licenses of each component so that other software can determine licensing conflicts. The *MavenCrawler* takes all of the metadata except interface information from each component's POM file. These files provide an optional field where licenses can be declared which is used to feed the results of the crawler application with licensing information. The unfortunate reality is that this optional field is rarely provided. In case of the test projects for the crawler application, only *13* of a total of *41* (31,7%) components provided licensing information in their POM files. This may be due to developers not caring about providing complete POM files but is more likely the case because a lot of Java OSCs were originally not developed using Maven. When such a component is hosted on a Maven repository the respective POM files are probably automatically generated. Since the detection of licenses is not trivial, the field is left out in that case. A solution would be to enhance the product model with some secondary information source for each component. This may be accomplished in similar fashions as used by the existing programs mentioned in section 1.1.

It can be seen that there are still issues that need to be addressed in the future but the crawler application already provides a fairly reliable dependency graph in the form of machine-readable exports. Even though some relationships between components may not be detected, the underlying dependency tree includes all the dependencies that an analyzed software project requires if the project only uses *Maven* for dependency management. The resulting product model enables future work on the mentioned issues as well as potentially more use cases and the *MavenCrawler* itself serves as a reference for future crawler implementations.

While future crawlers could use Gradle, CMake and even NPM as their base build systems, other use cases of the *MavenCrawler* may involve comparison of the data provided by POM files with secondary data sources. Another use case which could be particularly interesting for practitioners is research on effective visualizations of the gathered models. This could be useful since even small project's dependencies quickly become very confusing as demonstrated throughout this thesis. These examples show that collecting data in the form of a product model is equally useful to academic research as well as to engineers building software with OSCs.

# Appendix A: Crawler application dependencies

| Artifact id | Version | License | Description |
|---|---|---|---|
| bcel | 6.1 | Apache v2.0 | Library for analyzing bytecode. |
| guava | 24.0-jre | Apache v2.0 | General utility library for Java. |
| jackson-databind | 2.9.4 | Apache v2.0 | General data-binding package used for export. |
| jackson-dataformat-xml | 2.9.4 | Apache v2.0 | XML data format implementation. |
| jackson-dataformat-yaml | 2.9.4 | Apache v2.0 | YAML data format implementation. |
| jackson-datatype-jsr310 | 2.9.4 | Apache v2.0 | Jackson extension for supporting Java 8 date/time data types. |
| jcommander | 1.72 | Apache v2.0 | Library for building command-line interfaces. |
| logback-classic | 1.2.3 | EPL v1.0 / LGPL 2.1 | Logging utility |
| maven-aether-provider | 3.3.9 | Apache v2.0 | Implementation of Maven resolution for maven-resolver. |
| maven-invoker | 3.0.0 | Apache v2.0 | Java API for issuing commands to local maven installation. |
| maven-model | 3.3.9 | Apache v2.0 | Used for parsing POM files. |
| maven-resolver-api | 1.0.3 | Apache v2.0 | The API of the Maven artifact resolver. |
| maven-resolver-connector-basic | 1.0.3 | Apache v2.0 | Implementation of repository connections. |
| maven-resolver-impl | 1.0.3 | Apache v2.0 | Implementation of the maven-resolver-api. |
| maven-resolver-transport-http | 1.0.3 | Apache v2.0 | Implementation of the HttpTransporter service. |
| maven-resolver-util | 1.0.3 | Apache v2.0 | Utility used for handling dependency scopes. |

**Table A:** Crawler application dependencies

# Appendix B: Product model export excerpt

```
{
  "name" : "hddiff-parent",
  "homepageUrl" : null,
  "vcs" : "https://github.com/sweble/hddiff",
  "declaredLicenses" : [ "GNU Affero General Public License" ],
  "buildTool" : "MAVEN",
  "rootComponents" : [ ... , {
    "name" : "hddiff",
    "namespace" : "de.fau.cs.osr.hddiff",
    "version" : "2.0.4",
    "artifact" : {
      "filePath" : "/output/jars/target/dependency/hddiff-2.0.4.jar",
      "hashAlgorithm" : "SHA512",
      "hash" : "30970635F17893DDE4E39EEC176CE73B
                B849F99A800E1F55159E94D5E002038E
                CC0F1419DE7429391E248794558364B6
                2ABC120A2272E4DB2B148B126C9447CE"
    },
    "metaData" : [ {
      "description" : "Interface",
      "value" : {
        "provider" : "de.fau.cs.osr.utils:utils:3.0.6",
        "requiredFrom" : "de.fau.cs.osr.hddiff.editscript.EditOpInsert",
        "providedInterface" : "de.fau.cs.osr.utils.StringTools"
      }
    }, ... ],
    "dependencies" : [ {
      "target" : {
        "name" : "utils",
        "namespace" : "de.fau.cs.osr.utils",
        "version" : "3.0.6",
        "artifact" : {
          "filePath" : "/output/jars/target/dependency/utils-3.0.6.jar",
          "hashAlgorithm" : "SHA512",
          "hash" : "..."
        },
        "metaData" : [ ... ],
        "dependencies" : [ ... ]
      },
      "type" : "DYNAMIC_IMPORT"
    }, ... ]
  } ],
  "analyzeTime" : "2018-04-05T09:01:42.645Z"
}
```

**Listing B:** Excerpt from JSON product model export of the Hddiff project.

# Appendix C: Bootique dependencies

| Artifact Id | Group Id | Optional |
|---|---|---|
| aopalliance | aopalliance | No |
| asm | org.ow2.asm | Yes |
| bootique | io.bootique | No |
| bootique-test | io.bootique | No |
| bootique-test-badspi-it | io.bootique | No |
| cglib | cglib | Yes |
| guava | com.google.guava | No |
| guice | com.google.inject | No |
| guice-multibindings | com.google.inject.extensions | No |
| hamcrest-core | org.hamcrest | No |
| jackson-annotations | com.fasterxml.jackson.core | No |
| jackson-core | com.fasterxml.jackson.core | No |
| jackson-databind | com.fasterxml.jackson.core | No |
| jackson-dataformat-yaml | com.fasterxml.jackson.dataformat | No |
| javax.inject | javax.inject | No |
| jopt-simple | net.sf.jopt-simple | No |
| jsr305 | com.google.code.findbugs | Yes |
| junit | junit | No |
| snakeyaml | org.yaml | No |

**Table C:** Bootique dependencies

# Appendix D: Hddiff dependencies

| Artifact Id | Group Id | Optional |
|---|---|---|
| commons-codec | commons-codec | No |
| commons-collections | commons-collections | No |
| commons-compress | org.apache.commons | No |
| commons-io | commons-io | No |
| commons-lang3 | org.apache.commons | No |
| diff | com.sksamuel.diff | No |
| gson | com.google.code.gson | No |
| hddiff | de.fau.cs.osr.hddiff | No |
| hddiff-perfsuite | de.fau.cs.osr.hddiff | No |
| hddiff-wom-adapter | de.fau.cs.osr.hddiff | No |
| jaxb-impl | com.sun.xml.bind | No |
| joda-convert | org.joda | Yes |
| joda-time | joda-time | No |
| ptk-common | de.fau.cs.osr.ptk | No |
| rats-runtime | xtc | No |
| Saxon-HE | net.sf.saxon | Yes |
| slf4j-api | org.slf4j | No |
| swc-engine | org.sweble.wikitext | No |
| swc-parser-lazy | org.sweble.wikitext | No |
| sweble-engine-serialization | org.sweble.engine | No |
| sweble-wom3-core | org.sweble.wom3 | No |
| sweble-wom3-json-tools | org.sweble.wom3 | No |
| sweble-wom3-swc-adapter | org.sweble.wom3 | No |
| utils | de.fau.cs.osr.utils | No |
| xalan | xalan | Yes |
| xercesImpl | xerces | No |
| xml-apis | xml-apis | No |
| xml-resolver | xml-resolver | Yes |
| xmldiff | fc.xml.diff | No |
| xz | org.tukaani | Yes |

**Table D:** Hddiff dependencies

# Appendix E: Content of compact disc

The attached compact disc contains the source code files as well as other project files of the crawler application and the digital version of this thesis including all the LaTeX files for this writing. The top-level files on the disc are:

**thesis.pdf**: This is the digital version of this thesis.

**latex**: This is a directory containing all the tex files, graphics and the bibliography file of this thesis.

**source**: This is a directory containing the entire project folder of the crawler application.

# List of Figures

# References


Alpine Linux Development Team. (2017). About. https://alpinelinux.org/about/. Accessed: 2018-03-25.

Apache Software Foundation. (2004, January). Apache license. https://www.apache.org/licenses/LICENSE-2.0. Accessed: 2018-03-23.

Apache Software Foundation. (2016, July). Maven scm plugin. https://maven.apache.org/scm/maven-scm-plugin/. Accessed: 2018-03-25.

Apache Software Foundation. (2017a, August). Apache maven assembly plugin. http://maven.apache.org/plugins/maven-assembly-plugin/. Accessed: 2018-01-09.

Apache Software Foundation. (2017b, January). Apache maven invoker – usage. https://maven.apache.org/shared/maven-invoker/usage.html. Accessed: 2018-03-24.

Apache Software Foundation. (2017c, December). Commons bcel. https://commons.apache.org/proper/commons-bcel/. Acessed: 2018-03-24.

Apache Software Foundation. (2018a, February). Apache maven artifact resolver. https://maven.apache.org/resolver/. Accessed: 2018-03-23.

Apache Software Foundation. (2018b, March). Introduction to the build lifecylce. https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html. Accessed: 2018-03-24.

Apache Software Foundation. (2018c, March). Pom reference. https://maven.apache.org/pom.html. Accessed: 2018-03-18.

Apache Software Foundation. (2018d, March). Skipping tests. http://maven.apache.org/surefire/maven-failsafe-plugin/examples/skipping-tests.html. Accessed: 2018-03-25.

Bajracharya, S., Ossher, J., & Lopes, C. (2014, January). Sourcerer: an infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, *79*, 241–259. doi:10.1016/j.scico.2012.04.008

Bentmann, B. & Irawan, H. (2014, July). Aether/dependency graph. http://wiki.eclipse.org/Aether/Dependency_Graph. Accessed: 2018-03-24.

Beust, C. (2017, May). Jcommander. http://jcommander.org/. Accessed: 2018-03-25.

Bonaccorsi, A., Giannangeli, S., & Rossi, C. (2006, July). Entry strategies under competing standards: hybrid business models in the open source software industry. *Management Science*, *52*(7), 1085–1098. doi:10.1287/mnsc.1060.0547

Bonaccorsi, A. & Rossi, C. (2006, December). Comparing motivations of individual programmers and firms to take part in the open source movement: from community to business. *Knowledge, Technology & Policy*, *18*(4), 40–64. doi:10.1007/s12130-006-1003-9

Contrast Security Inc. (2014). The unfortunate reality of insecure libraries.

Crnkovic, I., Sentilles, S., Vulgarakis, A., & Chaudron, M. R. V. (2011, September). A classification framework for software component models. *IEEE Transactions on Software Engineering*, *37*(5), 593–615. doi:10.1109/tse.2010.83

Dahlander, L. & Magnusson, M. G. (2005, May). Relationships between open source software companies and communities: observations from nordic firms. *Research Policy*, *34*(4), 481–493. doi:10.1016/j.respol.2005.02.003

FasterXML, LLC. (2016, November). Jackson annotations. https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations. Accessed: 2018-03-25.

FasterXML, LLC. (2018, February). Jackson project home @github. https://github.com/FasterXML/jackson/blob/master/README.md. Accessed: 2018-03-25.

FOSSology Workgroup. (2017). Fossology. https://www.fossology.org/. Accessed: 2017-12-01.

Free Software Foundation, Inc. (2007a, June). Gnu affero general public license. https://www.gnu.org/licenses/agpl-3.0.en.html. Accessed: 2018-01-09.

Free Software Foundation, Inc. (2007b, June). Gnu general public license. https://www.gnu.org/licenses/gpl-3.0.en.html. Accessed: 2017-12-01.

Free Software Foundation, Inc. (2007c, June). Gnu lesser general public license. https://www.gnu.org/licenses/lgpl-3.0.en.html. Accessed: 2018-01-09.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional.

German, D. M., Gonzalez-Barahona, J. M., & Robles, G. (2007, October). A model to understand the building and running inter-dependencies of software. In *14th working conference on reverse engineering (WCRE 2007)*. IEEE. doi:10.1109/wcre.2007.5

German, D. M. & Hassan, A. E. (2009). License integration patterns: addressing license mismatches in component-based development. In *2009 IEEE 31st international conference on software engineering*. IEEE. doi:10.1109/icse.2009.5070520

German, D. M., Manabe, Y., & Inoue, K. (2010). A sentence-matching method for automatic license identification of source code files. In *Proceedings of the*

*IEEE/ACM international conference on automated software engineering - ASE '10*. ACM Press. doi:10.1145/1858996.1859088

German, D. M. & Penta, M. D. (2012, May). A method for open source license compliance of java applications. *IEEE Software*, *29*(3), 58–63. doi:10.1109/ms.2012.50

Gobeille, R. (2008). The fossology project. In *Proceedings of the 2008 international working conference on mining software repositories* (pp. 47–50). ACM.

Heineman, G. T. & Councill, W. T. (2001, June 11). *Component-based software engineering: putting the pieces together*. ADDISON WESLEY PUB CO INC.

Howison, J., Conklin, M., & Crowston, K. (2006). FLOSSmole: A collaborative repository for FLOSS research data and analyses. *International Journal of Information Technology and Web Engineering*, *1*(3), 17–26. doi:10.4018/jitwe.2006070102

Lau, K.-K. & Wang, Z. (2007, October). Software component models. *IEEE Transactions on Software Engineering*, *33*(10), 709–724. doi:10.1109/tse.2007.70726

Linux Foundation. (n.d.). Corporate members. https://www.linuxfoundation.org/membership/members/. Accessed: 2017-12-01.

McCabe, T. (1976, December). A complexity measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. doi:10.1109/tse.1976.233837

Meszaros, G. (2007). *Xunit test patterns: refactoring test code*. Addison-Wesley.

Open Source Initiative. (n.d.). Licenses by name. https://opensource.org/licenses/alphabetical. Accessed: 2017-12-01.

Open Source Initiative. (2007, March). The open source definition. https://opensource.org/osd. Accessed: 2017-11-27.

Oracle America, Inc. (2015, March). The java® virtual machine specification: java se 8 edition. https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf. Accessed: 2018-01-08.

Oracle America, Inc. (2017, August). The java® language specification: java se 9 edition. https://docs.oracle.com/javase/specs/jls/se9/jls9.pdf. Accessed: 2018-03-19.

OWASP Foundation. (2013). 2013 top 10 list. https://www.owasp.org/index.php/Top_10_2013-Top_10. Accessed: 2017-12-04.

OWASP Foundation. (2017). Owasp dependency-check. https://www.owasp.org/index.php/OWASP_Dependency_Check. Accessed: 2017-12-04.

Payne, C. (2002, January). On the security of open source software. *Information Systems Journal*, *12*(1), 61–78. doi:10.1046/j.1365-2575.2002.00118.x

Quality Open Software. (2018). Logback project. https://logback.qos.ch/. Accessed: 2018-03-23.

Riehle, D. (2007). The economic motivation of open source software: stakeholder perspectives. *Computer*, *40*(4), 25–32.

Riehle, D. (2009). The commercial open source business model. In *Lecture notes in business information processing* (pp. 18–30). Springer Berlin Heidelberg. doi:10.1007/978-3-642-03132-8_2

Szyperski, C. (2002). *Component software: beyond object-oriented programming (2nd edition)*. Addison-Wesley Professional.