Can we extract a relevant, available, and self-contained core of the Maven ecosystem?

Extracting the pillars of the community, and their dependencies.

Mathijs van der Schoot

Supervisor(s): Sebastian Proksch

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 28, 2024

Name of the student: Mathijs van der Schoot
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Casper Poulsen

Abstract

The Maven ecosystem, with an emphasis on Maven Central, contains a plethora of toy-projects. This paper addresses this problem by formulating a core containing the pillars of the Maven ecosystem, such that it can be exploited for research concerning library quality. The construction of said core is done by analyzing the availability, relevance and dependencies of packages in the Maven ecosystem. It involves answering questions regarding the distribution of library usages, the dependencies of popular libraries and the effect of a usage threshold filtering mechanism. We found the popular libraries to be utilized to an incredible extend, while their less popular counterparts are seldom, if ever, used. Delving into the creation of a core reveals its non-trivial nature, requiring intricate knowledge of the Maven dependency mechanism and its accompanying tools to construct. This paper explores the complexities, nuances and considerations involved.

1 Introduction

The Maven ecosystem and the public repositories it contains stand as keystones for Java developers, serving as epicenters for the exchange of Java libraries. At the heart of each library is its build file, called the *POM*. This file is not only instrumental in outlining the library's construction but more notably specifies the dependencies it relies on. These dependencies intricately interweave to form elaborate and complex dependency trees. These complex structures have captured the attention of researchers and has been the topic of many academic articles.

This paper is on extracting a subset of the dependency trees in the Maven ecosystem, for analysis purposes. Qualitative analysis for example, such that developers have assurance that the libraries in the subset are up to standard and can be safely included as a dependency in their project. With this goal in mind, we set the principles we want to adhere. The packages should be available, because packages broken for any reason are not useful for analysis. The packages should be relevant, because irrelevant ones are infrequently used which consequently means any analysis on them is irrelevant. Many packages on Maven Central are what we consider 'toy-projects', and should therefore be excluded from the subset. Lastly, we want the subset to be self-contained. This means every package in it can only depend on other packages in the subset, not on any outside it. It is an important principle, because analysis on the subset should include all dependencies involved. We explore the considerations it demands and the nuances of a self-containing clause in relation to the Maven dependency mechanism. We call the final subset, the 'core' of the Maven ecosystem, as it will contain the popular packages, which stand as pillars in the community. While the core we create might be useful to many, the main purpose of this paper is not to construct such a core, but to construct a method to do so. This means other researchers or individuals can take the method and modify it to their own needs, using the insights from this paper. We can now define the research question: can we construct a method to extract an available, relevant and self-contained core of the Maven ecosystem? We explore this question, with the creation of a core for qualitative analysis as tool to do so.

Prior research has been done on extracting a curated set of Java libraries. BlackBurn et al. have created the DaCapo set for benchmarking purposes[1]. This is unlike our goal, but it still worth mentioning as it is the first commonly used curated set of Java libraries for research purposes. Tempero et al. have laid the initial foundation for curated sets of Java applications for empirical analysis with their Qualitas Corpus[2]. This set is meant for analysis of bodies of code as well as the comparison core metrics. Dietrich et al. have made a similar contribution by forming a set of 76 executable Java libraries[3]. The major requirement for inclusion in the set was that the package was executable, so dynamic analysis could be performed. These works set out to create a curated set of libraries such that future research on them is replicable and produces stable results. This is different from our paper, because we are focussed on the *process* of creating such a curated set, with an emphasis on dependency tree's and library relevance. Furthermore, these works have resulted in sets of hundreds of carefully picked and analyzed libraries; but we do not care for the libraries themselves. We simply care for their position in the grander Maven ecosystem, and their relation to others.

Libraries.io[1] is a project, somewhat similar to ours. It maps projects and their dependencies, so developers can discover and compare libraries and monitor the existing dependencies they use for their projects, through notifications. It is a helpful tool for developers to maintain the health of their dependency tree, and useful for quality assurance purposes. However, it is different from ours, because we attempt to create a core within the ecosystem, for future research. The purpose and end result differs from Libraries.io in many ways, but might be complimentary as well. However, that depends on how the core is used.

While the end result is now quite clearly described, we also wish to answer some sub-questions regarding dependency tree's and dependency problems, in order to make informed decisions regarding the core. These sub-questions are as follows:

1. **How are usages of libraries distributed?**
2. **How common are libraries that are isolated to their own group?**
3. **What is the effect of a usage threshold filter on a self-contained core?**
4. **How frequently do relevant packages rely on irrelevant ones?**

Sub-question 1 asks after the dependents of libraries. What is noticeable when browsing through *mvnrepository.com* is the vast variance in usages between different libraries, especially visible in on popular page[4]. We wish to analyze this phenomenon, in order to dissect the relevant and popular from the irrelevant and unused.

[1]https://libraries.io/

Sub-question 2 is after artifacts that are only depended on by artifacts within their group. In this context, group meaning artifacts with the same Maven group ID. Such libraries may have many usages, but only by libraries from the same developers. Therefore, these libraries do not satisfy the relevance clause and should not be included in the core.

Sub-question 3 is an important stepping stone towards the final goal of this paper. Because we will be excluding infrequently used libraries from the core, we construct a usage filter and need to explore its effect on the core, with various usage thresholds.

Sub-question 4 delves into the dependencies of so-called relevant packages. What we consider relevant will be discussed in a later section, but simply put, this sub-question asks after which dependencies important packages rely on. This question is in relation to sub-question 3, because we are interested in how many popular/relevant packages are removed by a usage threshold filter, because they depend on unpopular/irrelevant packages. Such packages would be removed from the core due to the self-contained clause.

In order to answer these questions, the paper is divided into 6 sections. After this introduction, section 2 will discuss the methodology. It contains the steps taken and most of the considerations involved. Section 3 contains the results, meaning answers to the research sub-questions and the final decisions regarding the core. Section 4 is about some considerations regarding responsibility in research and section 5 is a discussion of the methods used, and their accompanying threats to the generalizability and validity of this research. It will also contain some suggestions for further research into this topic. At last, section 6 concludes this paper, it gives a quick summary and touches on interesting findings.

2 Methodology

An integral part of this research is the application *maven-explorer*[2]. We will give a concise summary of the major components of the application, but we highly advise using it for yourself, in order to get a good grasp on the research discussed in this paper. Furthermore, this paper discusses some aspects of Java and Maven in detail, but assumes the reader understands the fundamentals a priori.

2.1 The Maven dependency mechanism

Maven has many mechanisms to include and manage dependencies, such as scope, exclusions and version ranges. One could write a thesis on this topic, but only some are important for this research.

The version range mechanism allows a developer to specify a dependency version such as: [1.5,); meaning, any version from 1.5 onwards. This is resolved to the most recent version. In a dependency tree, it is possible for multiple versions of the same package to occur. This could lead to runtime issues or Java classpath conflicts, so Maven employs a heuristic called 'dependency mediation' to resolve this. It chooses the version nearest to the root node in the dependency tree. Lastly, Maven supports optional dependencies and dependency exclusions. These are mechanisms to omit certain packages from the dependency tree.

These are some important mechanisms that dictate what the final dependency tree of a project looks like. They are important to our research, yet we are mostly interested in the output rather than the methods that produce the output. For more information on these topic, visit the Apache Maven website [5]. The one mechanism that does have a direct influence on this research and the creation of our core, is dependency scopes. Each dependency specified in a POM file is noted with any of the following scopes: 'compile', 'runtime', 'test', 'provided', 'system' or 'import'. The latter is depreciated and occurs little, but such dependencies are still required to run the project and therefore do not get special treatment. The first 2 scopes are always transitively included in the JAR file and are required to run the application. The 'test' scoped dependencies are only required for testing purposes and are therefore not required to run the main application, nor are they transitively included, nor do they occur in the resulting JAR file. 'Provided' and 'system' dependencies are required to run but assumed to be provided by the Java environment and are therefore never included in a JAR file. However, they are required to run a project, whether they are transitive dependencies or not. In summary, all scopes except 'test' are required to execute a project and are therefore important for the 'self-contained' clause of the research question. For reference, note that a list including non-required dependencies is often much larger, because of many non-transitive dependencies and optional dependencies. This paper is focussed only on the required dependencies.

2.2 Software

We implemented a small pipeline to construct the final data set. It starts with maven-explorer, whose output is fed into the Maven dependency plugin, whose output we use to generate a dependency graph. This section of the paper discusses the pipeline in detail, such that any reader could reproduce the setup.

Maven-explorer

The maven-explorer application simply downloads and analyzes packages it finds in Maven Central indices[3]. These indices are files denoting the changes to the Maven Central repository for a single week. We use these indices because they are unbiased. An index simply lists packages chronologically, without any knowledge of relevance or availability. Maven-explorer extracts the package coordinates it finds in the indices and processes them. During processing, it detects whether there is an issue with the package. These issues are discussed further in section 2.3. The maven-explorer application is utilized through a Docker composition called *depgraph-deployment*[4]. It starts with Maven Central index 323, which is week 46 of 2015, and continues to subsequent indices, chronologically. It processes all packages it finds, along with their dependencies. All transitive dependencies are included as well, regardless of exclusions, optional dependencies and scopes. Moreover, dependencies whether tran-

[2]https://github.com/cops-lab/maven-explorer

[3]https://repo.maven.apache.org/maven2/.index/

[4]https://github.com/fasten-project/depgraph-deployment

sitive or not, are prioritized over packages found in the indices. This means any dependencies are processed before index packages are.

Maven dependency plugin

The second part of our pipeline is the Maven dependency plugin[5]. This plugin has many capabilities, one of which is producing a dependency list through the command: *'mvn dependency:resolve'*. When executed in a directory containing a POM, this command will print all dependencies needed for the execution of the current directory's project. This is more nuanced than one might think. Depending on various factors, the output of this command is vastly different. There are 2 methods to execute it: on the POM file of the target library, or on a constructed POM file with a single dependency on the target library. The first method will include test dependencies, provided & system dependencies and optional dependencies. An issue with this method is that optionality is not always given, only in some cases. The cases in which it is given are not documented in any way, nor discussed on any online forums to the best of our knowledge. An example is *commons-logging:commons-logging:1.1.3*, which does not denote any dependencies as optional, and *org.springframework:spring-context:3.2.14.RELEASE* which does. Both contain optional dependencies according to the POM's, but they are noted as optional only by the latter. The second method only prints compile and runtime dependencies. It does not print provided or system dependencies, as Maven is not concerned with such dependencies, only the end-user is.

We end up with one method that does not reliably print optional dependencies, and another method that does not print provided and system dependencies. Attempts at merging the two methods in order to have the best of both has not borne fruit. Likely the only way would be to generate the effective POM with *'mvn help:effective-pom'* and analyze it. Note that the effective pom is needed rather than a plain POM file, because of the parent POM and the import scope mechanisms. We attempted this, and merged the results into the output of the second method. It failed, and likely requires intricate and undocumented knowledge of the Maven dependency mechanism, to do this correctly. It is certainly possible because Maven is open source[6], but it is not viable for this research.

We will also not merge the results from the first method with the results from the second method because the output of the first method does not specify optionality consistently. It would still be a perfectly valid way to create a core. Including the full list in the core, in order to do analysis on all of them is certainly a reasonable option. However, because they are often left unused by dependents and we did not want to include excess dependencies, we decided to exclude optional dependencies from the core. It forces us to only use the second method. Provided and system dependencies are not present in its output. System dependencies are uncommon, but provided ones are not.

To execute this method, we put a library's identifying GAV coordinate into the template POM file, and use a simple bash script to navigate to the template POM and execute the command. Its output can now be used to generate a dependency graph.

[5]https://maven.apache.org/plugins/maven-dependency-plugin/

Graph

The output format of the dependency:resolve command is quite trivial. It is a text file, where every new line is a dependency. Transitivity of dependencies is disregarded, because a required dependency is required regardless of transitivity. For each library, we now know the required compile and runtime dependencies, regardless of depth. Optional dependencies are not included and dependency exclusions are taking into account. This means that for every library, we know all libraries it depends on, if it is included as a dependency itself (as explained in the previous section). We can now easily parse the data and turn it into a graph structure.

Most nodes in the graph are libraries that have not been fed into dependency:resolve itself, but exist because they are dependencies of libraries that *have* been fed into dependency:resolve. Such nodes have a status: incomplete. Throughout the rest of this paper, keep in mind that incomplete nodes have no *known* dependencies themselves, because they have not been processed. The difference between complete and incomplete nodes is significant, thus it will be clearly noted which node statuses apply to which results throughout this paper.

2.3 Data set

Maven-explorer		Dependency:tree		Graph	
Processed	Successful	Processed	Successful	Complete	Total
302,450 *100%*	*267,271* *88%*	*54,795* *18%*	*47,152* *16%*	*47,152* *16%*	*109,429* *36%*

Table 1: This table details the number of packages throughout the pipeline. 'Processed' is an application's input, 'successful' is an application's successfully processed output. 'Complete' denotes the graph nodes with status: complete. 'Total' is the total number of nodes in the graph, regardless of status.

Table 1 denotes the data set throughout the pipeline. The data set is fed from left to right, each step removing some part of the data set, except the right-most column. Maven-explorer outputs about 12% less than the input due to problems that arise when finding and analyzing packages. It might be an invalid POM file, or the JAR file cannot be found or any one of many other possible issues. Many libraries output by maven-explorer are not input to dependency:resolve. The reason is that maven-explorer is relatively quick, but the dependency:resolve plugin is not. We have a time constraint that does not allow for all maven-explorer output packages to be processed by dependency:resolve as well. The set of libraries input into dependency:resolve is simply a random set of 54,795 libraries, picked from the successfully processed packages from maven-explorer. Dependency:resolve outputs about 2% less than its input. This is due to a variety of problems, among which are artifacts stored in non-HTTPS repositories, missing child-modules and unresolvable plugins. These libraries are likely still executable, but we simply ignore packages home to such problems, because we believe it does not pose a major threat to the generalizability of our data. These problems may skew the data in a minor way, but due to time constraints, we deemed it not worth addressing.

2.4 Extracting packages

The research question dictates libraries in the core have to be available, relevant and self-contained.

Availability

Firstly, we define availability to be libraries who have been successfully processed by the dependency:resolve command. This command traverses all required compile and runtime dependencies, downloads the POM's and analyzes them. There is no guarantee that the packages that we consider unavailable are not executable. As mentioned, errors with plugins and artifact repositories occur in this part of the pipeline and that does not mean they cannot be executed. Luckily, the pipeline we set up does guarantee the converse: that all libraries in the graph, including their required dependencies, are available.

Relevance

In order to find relevant packages, we use the in-degree metric. In the context of this research, this means usage count. This metric is simple, yet rather effective in discerning the relevant from the irrelevant. We present a filter to omit the irrelevant based on an in-degree threshold. A problem with this metric is the possibility of isolated libraries, which is the topic of research sub-question 2. These are packages which are only depended on by other packages in their group. Such libraries might have a large in-degree metric, and yet are unused by other developers, thus we consider them irrelevant. If a library is transitively depended on, through another library in their group, we do not consider it isolated. They will be further analyzed in section 3.2. Most important is that the in-degree metric does not count dependents from the same group. Consequently, the usage threshold filter only counts dependents from other groups. As side note, many other metrics besides in-degree are viable for our use-case, some of which are briefly discussed in section 5, but time constraints prohibit us from using them. We can now define relevance as the following: packages that are not filtered out by the usage threshold filter are relevant.

We opt to include incomplete packages into the final self-contained core. Excluding such libraries significantly reduces the data set, and leaves out many prominent packages in the ecosystem. We want the core to contain relevant packages, so omitting many prominent ones would be destructive.

Self-containing

Lastly, the self-contained clause. We define a self-contained set of libraries to be a set where all libraries it contains, also contains its dependencies. A factor to take into account is that this holds transitivity. This results in an unfortunate and complicated edge case. For explanatory purposes, we give example scenario, figure 1. Using this example, we can see that if library A and all its required dependencies pass a usage threshold filter, library A is not necessarily self-contained because library B is dependent on Cv2, which does not pass the usage threshold. This situation is possible due to multiple versions in a dependency tree as given in the example, but

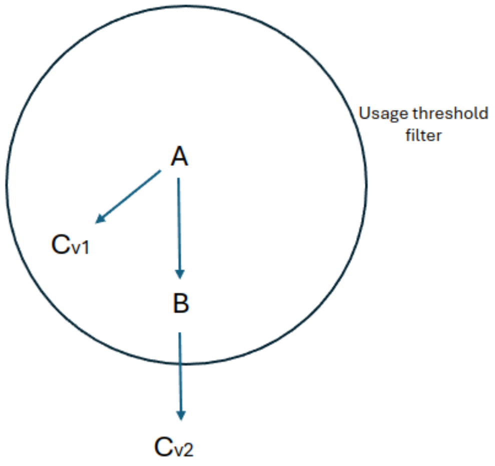

Figure 1: A possible scenario of relationships between libraries A, B and C version 1 and version 2. The circle denotes some usage threshold filter.

also due to exclusions. This means library A has a direct dependency on library C version 1, and as per example, only a transitive dependency on library C version 2. From this follows that while A is not dependent on Cv2, A is still left out of the core because its dependency B is. Depending on the purpose of the core, one can handle this in various ways. At the cost of the self-contained clause, we could disregard the fact that B is technically not in the core. This means for the purposes of library A, library B *is* in the core, while in fact it is not. If we would want to do analysis on library A and its dependents, this would be fine. However, it would invalidate any analysis on library B, because it leaves out its dependency Cv2. The manner in which our core is constructed, is to leave out library A. We want every library in our core to be valid for analysis. Note that this is a choice based on the use-case of the core. We want an analysis on our core to be a complete analysis of all compile and runtime dependencies involved.

There is still a major edge case to account for: the incomplete nodes, because we do not know the dependencies of such nodes. In the example, this problem would occur when library B is incomplete. We would be aware of Cv1, but not of Cv2, which (without our knowledge) does not pass the threshold. Almost all nodes in the data set are either incomplete or dependent on an incomplete node, which means that for almost all nodes, we cannot know for certain whether they are entirely self-contained.

There are various solutions to this problem. Our chosen solution is assuming the dependencies of incomplete nodes all pass the usage threshold. This is a false assumption, yet becomes truer for higher usage threshold values. As given in the results section, highly used libraries tend to only depend on other highly used libraries. This means that incomplete nodes that have a large in-degree, likely also have dependencies with a large in-degree. Furthermore, note that the incomplete libraries themselves are still subject to a usage threshold filter, only their dependencies are not. Another possible solution would be, if library B in figure 1 is incomplete, to exclude B from the core without excluding A. This is again at the cost of the self-contained clause, but we will not have to make false assumptions as is the case for the chosen solution. We did not opt for this solution because it excludes incomplete libraries from the core; the problem being that most popular libraries are incomplete, but we do want the popular libraries in our core.

Summing it up, the libraries in our core adhere to the following principles:

- **Available:** maven-explorer and the dependency:resolve command do not detect any errors.
- **Usage threshold:** the packages have at least some number of usages by libraries outside its group.
- **Self-contained:** if the core contains a package, all non-test dependencies of said package are also in the core.
  - **Regardless of status:** both complete and incomplete packages will be in the core.
  - **Incomplete assumption:** dependencies of incomplete libraries are assumed to conform to the above principles.

These principles are certainly not set in stone. The purpose of the core has great influence on the principles it should conform to, such that the principles change per use-case.

3 Results

We have subdivided the data set by status and isolation. The following sections of the paper will use different subsets, thus the reader should take note which set is referenced.

3.1 Usage distribution

Figure 2 shows a histogram of library usages and consequently answers research sub-question 1. Note that this histogram includes all nodes, regardless of status and isolation. That is why such a significant number of libraries are depended on once, because non-isolated nodes are depended on at least once. This histogram is only up to 30 usages and 3,624 libraries in the data set have usages above that. Those packages could be considered popular, an argument strengthened by the fact that the sum of their usage counts accounts for 66.7% of all usage relationships. This is in direct correlation to what we expect after *mvnrepository.com*. Both show that the popular libraries are used to an incredible extend while the rest is on average used rarely. Table 2 gives the 10 most popular packages. Note that our data set does not contain test dependencies. This is quite clear from the fact that JUnit is not on our list, while it is vastly more used than any other library, as shown on Mvnrepository. It is also noticeable how Mvnrepository gives a rather steep decline of usages on the top 10 most popular libraries page, which is not the case for us. Furthermore, our list contains packages that do not occur in the Mvnrepository top 10. It is not documented how Mvnrepository calculates dependent count, but we calculate the count transitively, which is likely the reason for both disparities. A good example is the 'aopalliance' package, which is the most popular in our dataset. It is used heavily because it is depended on by the Spring framework packages, among others, which are used heavily used themselves. However, Mvnrepository only reports it to be used by 925 packages, as of Januari 28[7]. A final note to make is that 8 of these 10 most popular libraries have status incomplete. This is due to the fact that the 47 thousand complete libraries have been picked at random from the 267 thousand packages that maven-explorer output.

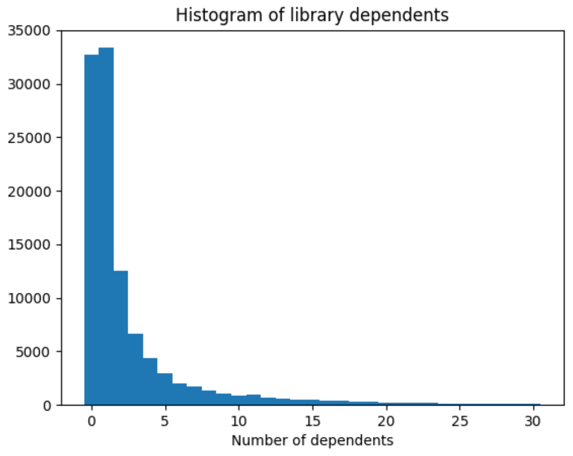

Figure 2: Histogram of library dependents. It only includes all libraries with less than 30 usages.

Group	Artifact	Version	Dependents
aopalliance	aopalliance	1.0	5,456
javax.inject	javax.inject	1	5,299
commons-logging	commons-logging	1.1.1	5,109
org.slf4j	slf4j-api	1.7.12	4,444
commons-io	commons-io	2.4	4,256
com.google.guava	guava	18.0	3,731
commons-codec	commons-codec	1.6	3,363
commons-collections	commons-collections	3.2.1	3,332
org.slf4j	slf4j-api	1.7.7	3,098
commons-lang	commons-lang	2.6	3,024

Table 2: The 10 most used libraries in the data set. Includes all nodes.

3.2 Isolated libraries

Table 3 shows the number of libraries that are not isolated, meaning, used at least once by libraries outside their group. It is noticeable how only 7,232 of the 47,152 complete packages are not isolated, this is about 15.3%. The complete packages are chosen at random, so this statistic reflects on the Maven ecosystem as a whole. It supports the 'toy project' hypothesis, mentioned in the introduction.

3.3 Usage threshold filter

Table 4 shows the effect of a usage threshold filter on the data set. The chosen thresholds are quite low because the lower

	Regardless of status	Complete
Regardless of isolation	*109,429*	*47,152*
Not isolated	*40,257*	7,232

Table 3: This table shows how many nodes are **not** isolated. The middle row is purely for reference, the numbers it contains can be found in table 1.

thresholds already remove many libraries. Furthermore, the data set is not large enough for larger thresholds to give valuable results; the discussion section will elaborate on this. As mentioned, the usage threshold filter only counts different-group-usages. The core will contain incomplete packages as well, but the table also gives the filter results for only complete packages. It is noticeable how a threshold of 1 gives such a steep decline for complete packages. Again, this is representative of the whole Maven ecosystem. Using this table, the threshold to construct the core is chosen in a later section.

	Regardless of status	Complete
0	*109,429* *100%*	*47,152* *100%*
1	*40,144* *37%*	*7,119* *15%*
2	*25,559* *23%*	*4,656* *10%*
5	*14,124* *13%*	*2,542* *5%*
10	*8,747* *8%*	*1,527* *3%*
50	*2,390* *2%*	*415* *1%*

Table 4: The number of libraries remaining after applying a usage threshold filter. The top-most threshold of 0 means no filter. The percentages in a column are relative to the 0 threshold value in the column.

Table 5 shows how many of the 1,000 most popular *complete* packages make it through the various usage threshold filters, this excludes test dependencies. This answers research sub-question 4, because it shows how many of the 1,000 most popular packages rely on irrelevant packages. At every threshold, the 1,000 most popular packages are not reduces as much as the rest of the data set, when comparing with table 4. This tells us that it is common for popular packages to mostly rely on other popular packages. The most noticeable facet of this table is the results for a threshold of 50; it is a rather steep decline from a threshold of 10. It shows that while popular packages rely on often used packages, they do not always rely on the most popular packages.

Threshold	0	1	2	5	10	50
Remaining packages	*1,000*	*980*	*974*	*963*	*949*	*415*

Table 5: This table gives the usage threshold filter results for the 1,000 most used complete packages.

3.4 A relevant, available and self-contained core

For our purposes, we opt for a usage threshold of 10. According to table 4 and 5, a threshold of 10 reduces the data set by 92%, but only reduces the 1,000 most popular packages by 5%. It leaves 8,747 packages, containing almost all popular ones. It is a number that is small enough for qualitative analysis, but large enough for developers to have all dependencies they need. Table 6 shows the top 10 most used packages that are discarded because of the current threshold, per threshold. It shows that with a threshold of 50, the discarded packages are much more popular than for a threshold of 10. Therefore, we think that for our use-case, a threshold of 10 is reasonable. But as explained, the core we create is not of much significance. The optimal threshold is different each each use-case.

4 Responsible Research

Producing results in a reproducible manner is crucial to research. This paper is mostly about the methodology to create a core, but the results presented in the section after this should still be reproducible. Firstly, maven-explorer gives consistent results, because it functions in a deterministic manner. Knowing the starting index as well as the number of processed packages gives a consistent data set. Secondly, the dependency:tree command is conversely not deterministic. Besides the fact that its input is a random set from the maven-explorer output, it also processed version ranges. This can result in disparities, which will be discussed further in section 5. The creation of the dependency graph is deterministic, as well as the results extracted from it.

This research requires the storage and analysis of much data. The data is publicly available and most projects contain a license in some way. Many have a reference to a license in the POM file and/or in the code repository. However, this is irrelevant to us. We do not incorporate anything into a project of our own much less publish it. It is therefore legal to use in this research.

5 Discussion and future work

5.1 Threats to validity

We wish for the methods discussed in this paper to be generalizable to the whole Maven ecosystem, but there some impediments to discuss first. Starting with the packages' release times, which are concentrated around 2015/2016, as visible in figure 3. This is due to maven-explorer, which starts with an index in 2015 and slowly progresses chronologically. This

To threshold 1		To threshold 2		To threshold 5		To threshold 10		To threshold 50	
156	org.kaazing:community.license:2.17	128	org.glassfish.web:javax.servlet.jsp:2.3.2	108	org.opensaml:opensaml:2.6.1	97	com.force.api:force-partner-api:24.0.0	711	org.sonatype.plexus:plexus-sec-dispatcher:1.3
82	org.kaazing:net.tcp:1.1.0.9	96	org.eclipse.jetty:jetty-jsp:9.2.10.v20150310	67	org.apache.velocity:velocity-tools:2.0	70	com.101tec:zkclient:0.3	136	org.apache.maven.wagon:wagon-http-shared:2.8
82	org.kaazing:net.api:1.1.0.9	39	org.fusesource.leveldbjni:leveldbjni-linux64:1.5	65	sslext:sslext:1.2-0	57	org.webjars.bower:jquery:3.7.1	134	org.apache.maven.wagon:wagon-http-lightweight:2.8
74	org.codehaus.plexus:plexus-i18n:1.0-beta-7	27	org.json4s:json4s-native_2.10:3.2.5	52	org.apache.stanbol:org.apache.stanbol.enhancer.engines...	43	org.jboss.weld:weld-api:1.1.Final	132	com.sun.jersey:jersey-json:1.9
57	org.codehaus.plexus:plexus-velocity:1.1.7	27	org.http4s:http4s-websocket_2.11:0.1.3	42	org.apache.camel:camel-core:2.15.5	43	org.apache.camel:camel-core:2.14.4	130	org.apache.maven:maven-settings-builder:3.3.3
54	org.wildfly.swarm:config-api-runtime:0.3.27	25	org.apache.neethi:neethi:2.0.4	36	org.apache.openejb:javaee-api:6.0-6	39	io.airlift:slice:0.10	117	org.mobicents.diameter:jdiameter-ha-impl:1.7.0.58
52	org.wildfly.swarm:config-api:0.3.27	22	org.springframework:spring-beans:4.3.2.RELEASE	36	org.apache.camel:camel-core:2.14.3	27	com.vaadin:vaadin-sass-compiler:0.9.12	109	org.jboss.resteasy:resteasy-jaxrs:3.0.8.Final
52	org.wildfly.swarm:config-api-modules:0.3.27	22	com.amazonaws:aws-java-sdk-core:1.10.41	33	org.apache.openwebbeans:openwebbeans-impl:1.2.7	25	org.apache.metamodel:MetaModel-full:4.3.3	95	org.mobicents.cluster:cache:1.14.0.FINAL
51	org.wildfly.swarm:config-api-runtime:0.3.25	19	org.openrdf.sesame:sesame-util:2.8.3	29	org.apache.openwebbeans:openwebbeans-ee-common:1.2.7	24	com.amazonaws:aws-java-sdk-core:1.10.39	92	com.typesafe.play:twirl-api_2.11:1.1.1
51	org.webjars.bower:angular:1.8.3	18	org.jboss:jboss-common-core:2.5.0.Final	29	org.apache.openwebbeans:openwebbeans-web:1.2.7	23	com.thinkaurelius.titan:titan-core:0.5.0	84	org.apache.maven.reporting:maven-reporting-api:3.0

Table 6: The top 10 most used packages, discarded because of the usage threshold filter. Example: the ”To threshold 5” column displays the top 10 most used packages that passed the filter with a threshold of 2, but not with a threshold of 5. The number in the column left of the ID's are the number of usages.

poses the problem that our core is unusable for many purposes, because the packages it contains are dated. However, the methods used to extract our core are likely perfectly valid for the extraction of a modern core. There is a possibility that this is not the case, which is if the Maven ecosystem or standard practices regarding dependencies have vastly changed. For example, if the distribution of package usage as given in figure 2 has changed. While we cannot be certain of any such macro changes in the ecosystem, we feel it improbable of changes large enough to invalidate this research.

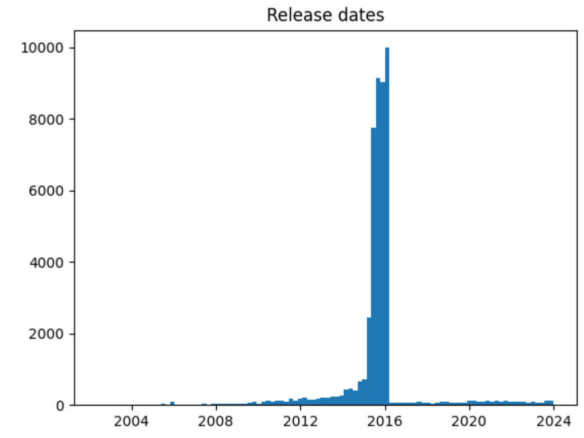

Figure 3: Release date histogram of the complete libraries in the dataset.

Many results given in this paper would be different if generated on the whole Maven ecosystem, rather than a subset. The inclusion of incomplete nodes into our data has skewed the statistics calculated on the data in a significant way. Incomplete nodes are referenced at least once, while complete nodes are not. This means the average usage count of libraries in the data set is relatively high, a phenomenon visible in table 4. There, complete nodes in the right-most column are rarely used by libraries outside their group, which is not true for the full data set, in the middle column. The created core includes such incomplete nodes because we want it to contain popular libraries, but should be left out if it is supposed to be generalizable to the Maven ecosystem. Another facet to take note of is the effect of data set size. Depending on the size of the desired core and its contents, the filtering threshold could be increased. The larger the size of the initial data set is, the higher the usage counts. Many results are given in percentages, which does mean that it should be generalizable to data sets of different sizes.

A small note to make is why we use maven-explorer, and why it could be omitted. We use the application to analyze indices, download POM files and analyze them for errors. However, we could have downloaded the POM files ourselves, and the errors are also detected by dependency:resolve. We have used maven-explorer for simplicity reasons. Our time constraint does not incorporate a large margin for error, making it quite beneficial to have a section of the pipeline that we don't need to develop from scratch. However, the cost is that the release dates of the packages in the data set are quite far in the past. We recommend future research to avoid using maven-explorer, to allow for more flexibility.

The Maven ecosystem is continuously changing, which can complicate the reproduction of our results. Fortunately, Maven has a policy dictating that uploaded artifacts are immutable. In combination with the fact that the Maven Central indices do not change, one could relatively consistently reproduce the results presented in this paper. The only issue would be version ranges, or the depreciated 'LATEST' or 'RELEASE' version tags. The dependency:resolve command, resolves dependencies with such version inputs to the most recent version conforming to the range or tag. From this follows that a newly published version can change the dependency graph. This is certainly a threat to the reproducibility of our results and can regrettably not be countered.

A purpose of this paper is to find a core of popular packages. However, our analysis is based on the dependency relations between *public* Maven artifacts. There are often many more personal or profession projects that have not been published to a public repository. As example would be Github's 2022 study, which notes that many more developers public to private repositories rather than public ones[8]. The omittance of private repositories can certainly skew the data, which we actually noticed. According to a 2020 JVM ecosystem survey by Snyk, the Spring framework is utilized by 60% of the software professionals in the survey[9]. However, in the list of most used packages in our data set, the first Spring framework package is only in the 118th place, which is much lower than we expect after reading the Snyk survey. If such personal or professional project were included in the data set, it would certainly change table 2 and table 4. Unfortunately, we can only use public data for our research, so a solution to this problem does not exist.

5.2 Reflection

We have discussed many possibilities and edge cases in the creation of a core in Maven central. Most notably, the self-contained clause has many facets worth noting, and there does not exist a single correct interpretation. The choice on optional dependencies, exclusions and scope have a large impact of the final core, especially when taking transitively of dependencies into account.

We failed to include provided and system dependencies in the dependency graph, because of the semantics of the Maven dependency plugin as well as a lack of time. However, when performing analysis on packages in the Maven ecosystem, such dependencies are in fact relevant for analysis, because they are required to run the projects that depend on it. They do have a separate role in the ecosystem. They are meant to be added manually to the runtime classpath, often through OS environment variables. The exact setup is chosen by the user, which means it is not strictly part of the dependency tree. However, this is a somewhat ambitious assertion. We attempted to include the provided and system dependencies through analysis of the effective POM's. However, we abandoned the idea upon encountering discrepancies with the actual POM files. The time constraint did not allow us to further research the topic.

The answers to the research sub-questions have resulted in the statistics and information needed in finding a core. However, they show much more. We have seen that the gap between the popular and unpopular libraries is vast, 3,624 or 3.3% of the libraries in the data set account for two thirds of the total dependency relations. This figure supports the 'toy project' hypothesis, but note that it does not speak to the quality of the Maven ecosystem. One might be quick to think that the toy projects are likely of unsound quality, but this does not have to be the case. Lima and Hora found in their 2020 paper that popular projects have a higher code quality than others[10], but also found that they are more likely to introduce breaking changes. This poses the question, are popular libraries advantages to use?

Throughout the research we have depended on in-degree centrality to lead to answer. However, many more centrality metrics are possible, and some are just as- or more viable for this use-case. We have experimented with PageRank[11], and noticed some results are fairly similar. The top 1,000 of both metrics overlap with 67.5%. A PageRank threshold filter is certainly possible, but requires a study into the algorithm and the significance of its results. The metric most commonly referred to with libraries is usage count, such as on Mvnrepository and the Sonatype website[6]; so usage count is most tangible. This, along with a time constraint motivated our choice for usage count.

5.3 Future research

Usage count sufficed for this paper, but is still quite limiting. It is possible that other centrality metrics uncover interesting anomalies regarding dependencies. A future research topic would be mapping the Maven ecosystem in terms of various centrality metrics. Using graph theory to uncover the multi-faceted dependency networks underlying the Maven ecosystem. This paper touches the surface, but there is likely much more to discover.

This paper has worked on deducing a set of libraries for qualitative analysis on dependencies. However, we have never defined qualitative analysis. Numerous tools are available for assessing code cleanliness, documentation, and bugs. However, the outcomes of such analyses do not provide insight into whether incorporating it as a dependency is a wise decision. An interesting question for future research would be: what software measurements are relevant to decide on the inclusion of a dependency? The topic of software quality is already widely explored, but the aforementioned research question is after the quality and evolution of dependencies, in order to decide on the inclusion of a dependency.

Another interesting subject of research would be whether the methods presented in this paper are equally valid for other code ecosystems; NPM for example. NPM also functions based on a build file, so dependency tree generation is possible, but there are some important differences. Due to the nature of Javascript, it does allow multiple versions of the same package in the dependency tree, and does not support provided or system dependencies. Differences such as these require the core creating methodology to be adapted. Analyzing and comparing its results to the results in this paper would be an interesting inquiry.

6 Conclusion

The Maven ecosystem, with an emphasis on Maven Central, contains a plethora of toy-projects. Developers want the dependencies they use to be reliable and continuously maintained, which means toy-projects should often be avoided. This paper addressed this problem by formulating a methodology to create a core of the Maven ecosystem such that it can be analyzed for library quality and consequently used by developers to have guarantees about the quality of their dependencies.

In our dependency graph of compile and runtime dependencies, we found that two thirds of all dependency relations are towards 3.3% of the libraries. We found that 85% of the libraries are isolated to their own group and only 5% are used more than 5 times by libraries outside their group.

A self-contained clause for the creation of a core is a complicated process when using existing tools. Creating a new tool is even more complicated, because the Maven dependency mechanism is no trivial system. To create a core, analyze the use-cases and simplify the process accordingly.

[6]https://central.sonatype.com/

References

[1] S. M. Blackburn, R. Garner, C. Hoffman, *et al.*, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[2] E. Tempero, C. Anslow, J. Dietrich, *et al.*, “The qualitas corpus: A curated collection of java code for empirical studies,” Dec. 2010, pp. 336–345. DOI: 10.1109/APSEC.2010.46.

[3] J. Dietrich, H. Schole, L. Sui, and E. Tempero, “Xcorpus - an executable corpus of java programs,” *JOURNAL OF OBJECT TECHNOLOGY*, vol. 16, 4 Aug. 2017, ISSN: 1660-1769. DOI: 10.5381/jot.2017.16.4.a1..

[4] Mvnrepository, *Maven repository: Top projects at maven repository*, Accessed: 2023-12-09. [Online]. Available: https://mvnrepository.com/popular.

[5] T. A. S. Foundation, *Introduction to the dependency mechanism*, Accessed: 2023-12-24, 2002. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.

[6] A. S. Foundation, *Maven*, https://github.com/apache/maven.

[7] F. R. Olivera, *Introduction to the dependency mechanism*, Accessed: 2024-01-28, 2006. [Online]. Available: https://mvnrepository.com/artifact/aopalliance/aopalliance.

[8] GitHub, *The global developer community*, Accessed: 2023-12-09, 2022. [Online]. Available: https://octoverse.github.com/2022/developer-community.

[9] Snyk, *Jvm ecosystem report 2021*, Accessed: 2023-12-09, 2021. [Online]. Available: https://snyk.io/reports/jvm-ecosystem-report-2021/.

[10] C. Lima and A. Hora, “What are the characteristics of popular apis? a large-scale study on java, android, and 165 libraries,” *SOFTWARE QUALITY JOURNAL*, vol. 28, no. 2, pp. 425–458, Jun. 2020, ISSN: 0963-9314. DOI: 10.1007/s11219-019-09476-z.

[11] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” *COMPUTER NETWORKS AND ISDN SYSTEMS*, vol. 30, no. 1-7, pp. 107–117, Apr. 1998, 7th International World Wide Web Conference, BRISBANE, AUSTRALIA, APR 14-18, 1998, ISSN: 0169-7552. DOI: 10.1016/S0169-7552(98)00110-X.