An exploratory study about extent of use of released packages in the Maven Central Repository

Horia Zaharia
Supervisors: Sebastian Proksch, Mehdi Keshani
EEMCS, Delft University of Technology, The Netherlands
22-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Maven Central Repository hosts over 9 million repositories which ease software reuse. Since its appearance, Maven has been studied and characterized using different popularity and quality metrics, in order to identify defining patterns and possible improvements. This study aims to analyze extent of use of packages released in the Maven Central Repository. Extent of use is a metric that quantifies how much of a package is used. The findings of this study show that on average 73.2% of available methods and 77.1% modules in a package are used by dependents of the package. The study also shows that extent of use at method level is highly correlated with extent of use at module level (Pearson correlation coefficient r = 0.829).

Keywords: Maven, Exten of use, Metrics

1 Introduction

Software reuse is a necessary practice especially in the enterprise application environment. This lead to the appearance of centralised repositories that provide developers with package management systems such as Maven, npm, pip, yarn. This study analyzes the Maven Central Repository at method level with focus on the extent of use of packages using the FASTEN software.

Maven Central provides developers with over 9 million repositories that can be used as third party libraries in their own productions. The reuse of these libraries in new releases lead to the creation of an entangled software ecosystem. Since this ecosystem evolved naturally, studying its performance is necessary to develop and adopt better practices. The existing literature proposes multiple methods to quantify characteristics of packages in Maven Central.

[Raemaekers *et al.*, 2012] presents four different metrics centered around method level changes which can be used in exploratory studies in order to reveal patterns within the Maven ecosystem. The metrics presented can be used to choose an appropriate design of own projects. However, this study and its proposed metrics only account for the changes that appear throughout the evolution of the package.

[Sajnani *et al.*, 2014] studies whether or not popularity is a measure of quality. "One of the perceived values of open source software is the idea that many eyes can increase code quality and reduce the amount of bugs" ([Sajnani *et al.*, 2014]). However, the cited paper challenges this wide-spread view: there are virtually no correlations between popularity and software quality metrics. This study proposes a metric that could be used to identify behaviours and patterns present in the Maven Central Repository: Extent of use. Extent of use quantifies what part of a package is used by its dependents. This metric is used to analyze Maven at method level and module level.

This paper aims to answer three related research questions. The first research question posed is "On average, what extent of a released package version is used by others in Maven Central?". After computing the extent of use, we use the results to investigate the second research question "Is extent of use correlated with the number of dependents?". The third research question focuses on the relation between extent of use at method level and module level: "Is extent of use at method level correlated with extent of use at module level?". The answers to the questions should serve as the first steps in analyzing package management systems through the lens of extent of use. Further research could prove a correlation between code quality and high extent of use, since developers avoid non-qualitative solutions.

This study heavily relies on functionalities provided by the FASTEN framework. FASTEN provides tools to analyze dependency relationships present in the Maven ecosystem at method level and a database with information about the ingested packages. These tools help create callgraphs which describe interactions between methods. The generated callgraphs and the information from the database are used to compute the extent of use.

This paper is structured as follows. Section 2 provides the reader with background information needed to tackle this paper. Section 3 discusses the goals and structure of the study, before describing the steps taken to derive the results. Section 4 shows the results derived from the study. Section 5 covers the responsible research and the replication package. Section 6 ends the paper with discussions and conclusions.

2 Background

Current literature provides extensive research about the Maven Central Repository. Continuing studying this topic is necessary because the ecosystem is continuously changing and evolving. The following subsections provide the reader with a glossary of terms discussed throughout the paper and an overview of the existent related work.

2.1 Glossary

Maven Central Repository - Repository that holds over 9 million packages that can be imported in own projects. This is the scope of this study.

Package - jar file that provides functionalities which can be used in own projects. This study focuses on packages present in the Maven Central Repository. Packages published in the Maven Central Repository are named and uniquely identified in the format groupId:artifactId:version. In the literature packages may be encountered under the names artefact or libraries.

Dependent - Package A is a dependent of package B if package A needs package B to build, run or compile.

Dependency - Package A is a dependency of package B if package B is a dependent of package A.

Callgraph - A callgraph is a directed graph that describes the interactions between methods of different packages. The nodes present in the directed graph represent methods that are called or make calls towards other methods. An edge from A to B in the graph illustrates that method A calls method B.

Dependent resolution - The process that yields the dependents of a package.

Dependency resolution - The process that yields the dependencies of a package.

FASTEN - FASTEN [FASTEN, 2022] is an intelligent software capable of performing package management analysis.

Functionalities of FASTEN include callgraph generation, dependents/dependency resolution, vulnerability analysis etc. FASTEN also provides a database containing information about the methods and packages analyzed.

Extent of use - The metric introduced by this paper. This metric quantifies the natural idea of "how much of this package is being used?".

2.2 Related Work

[Kula *et al.*, 2017a] present a way to decide when a library needs to be updated based on its usage level. The authors empirically explore library usage as a means to describe the library's age. Their results show that changes in library usage are not random, since 81.7% of the popular libraries fit the polynomial models.

[Kula *et al.*, 2017b] propose the Software Universe Graph Model as means to study evolving software systems and their dependencies over time. The researchers studied the Maven Central and the CRAN repositories using intelligent mining to reveal trends within software ecosystems. Their findings show that Maven ecosystems have a more conservative approach to updating dependencies while their CRAN counterparts are more adept to adopting changes.

[Kula *et al.*, 2018] wants to answer a hot topic regarding dependency management. Do maintainers update their dependencies? In order to answer this question the researchers conducted an empirical study that covers over 4600 GitHub software projects and 2700 library dependencies. The findings of this study show that 81.5% of the studied systems keep outdated dependencies. The adoption behaviours of maintainers greatly influences how the Maven ecosystem evolves.

[Raemaekers *et al.*, 2012] evaluates method level changes between different releases of the same libraries making use of four different metrics that offer insights into the stability of the said libraries. It assesses the level of change between releases in a quantifiable manner.

[Raemaekers *et al.*, 2017] is interested in how new releases of a library impact the client libraries and their semantic versioning. The findings of the authors show that on average 1 in 3 releases introduce breaking changes that produce compilation errors which need to be addressed. This study is however more interested in how developers should tackle the problem of breaking changes and deprecation.

[Soto-Valero *et al.*, 2020] studies the effect of bloated dependencies in systems. Bloated dependencies are defined as libraries that are packaged and compiled within the application but are not required to compile and run the said application. The findings of the study reveal that most bloated dependencies are the result of transitive dependencies indicating that the default dependency selection of Maven might not be optimal.

[Kula *et al.*, 2015] studies the adoption habits and trust of maintainers towards new releases of an existing library. The study concludes that maintainers are becoming more trusting of new releases being inclined to update their existing systems to the latest release.

[Raemaekers *et al.*, 2013] presents the Maven Dependency Dataset containing information on 148253 Java libraries. The researchers used a supercomputer to calculate metrics and changes over multiple levels (package, class, method) and to generate a complete call graph which includes call, inheritance, containment and historical relationships.

3 Methodology

This section describes the approach, objectives and processes undertaken to arrive at the results. The following section presents the high-level approach of the study. Section 3.2 restates the research questions and the incentives to answer them. Section 3.3 illustrates the data selection process and explains the choices made to derive a representative subset of packages from the Maven Central Repository. Section 3.4 gives the mathematical definition of extent of use. Sections 3.5 and 3.6 demonstrate the callgraph generation, respectively the analysis performed on the callgraphs to compute extent of use.

3.1 High-Level Approach

The process starts with the selection of a representative sample of packages from the Maven Central Repository. Using functionalities provided by FASTEN, the dependents and the dependencies of the dependents are computed for each selected package. Using the identified artifacts we generate a callgraph of the method calls present in the package, dependent and dependencies of dependent. Together with information obtained from the FASTEN database, these callgraphs represent the input for the Extent of Use analysis.

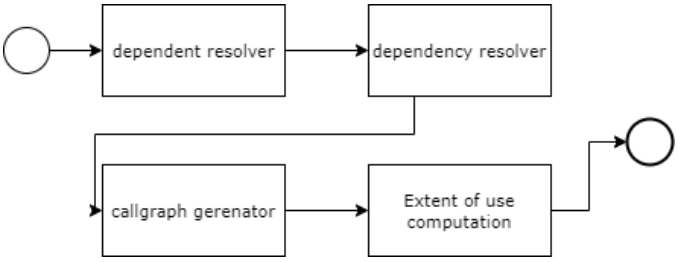

Figure 1: Study Work Flow

3.2 Research Questions

This section describes the research questions related to the metrics defined in section 3.4 and their importance for the literature.

Research Question 1:

On average, what extent of a released package version is used by others in Maven Central? The answer to this question provides a baseline for the new metric defined as extent of use. In future works, extent of use could provide a new tool for identifying patterns present in the Maven Central Repository.

Research Question 2:

Is extent of use correlated with the number of dependents? Answering this question will reveal if extent of use is related with the number of dependents a package has.

Research Question 3:

Is extent of use at method level correlated with extent of use at module level? The answer to this question will provide information on the relation between extent of use at method and module level. If the two are uncorrelated it means only certain modules are used, while their counterparts in the same package are not. Future works could provide an improved dependency management strategy based on used modules, rather than whole packages.

3.3 Data Selection

Given the immensity of the Maven Central Repository it is not feasible to conduct the analysis over the whole repository. To reduce the number of packages to be analyzed a time window is chosen: only packages added to FASTEN between 01/10/2021 and 31/03/2022 will be considered.

Test related packages are removed from the dataset, since releases in Maven Central Repository are shipped without the testing code available. Given that different versions of the same package will likely yield similar results, only one version per package will be considered. Furthermore, the design of large scale products is often to release multiple packages that depend on one another, under the same groupId; to avoid data skewness generated by this design choice only one package per groupId is considered.

After filtering weighted random sampling is performed, using the number of dependents a package has as weight for sampling. The number of dependents a package has is a natural choice for sampling, since packages with more dependents have more influence over the Maven ecosystem.

3.4 Metrics

Extent of use for a package at method level in a dependent is defined as:

Extent of use = $\frac{n_c}{n_a}$

Where:

n_c = number of non-private methods called by dependent and the dependencies of the dependent

n_a = number of non-private methods available in the package

Only non-private methods are considered to avoid artificially inflated extent of use generated by the package design. This metric maps the extent of use to a real number in the interval [0,1], where 1 corresponds to full extent of use and 0 corresponds to no use at all. n_a is identified by a series of SQL queries performed on the FASTEN database. n_c is found by comparing the methods available retrieved from the database, with the ones present in the callgraph of the package, dependent and its dependencies. Further explanations about the computations of these variables are given in section 3.6.

Thus extent of use for a package at method level is defined as the mean of the extents of use in its dependents:

ΔExtent Of Use = $\frac{1}{n}\sum_{i=1}^{n} EoF_i$

Where:

n = number of dependents the package has
EoF_i = Extent of use for the package in depenent i

Extent of use for a package at module level in a dependent is defined as:

Extent of use = $\frac{m_u}{m_a}$

Where:

m_u = number of modules that contain at least a method called by the dependent

m_a = number of modules in the package

This metric similarly maps extent of use to a real number in the interval [0,1], but considering usage of modules instead of method calls. m_a is retrieved from the FASTEN database. m_u is computed by creating the set of modules containing the methods called by the dependent, which are found in the callgraph. Extent of use for a package at module level is the mean of the extents of use at module level over all dependents, just like at method level.

3.5 Callgraph generation

Callgraphs describe the relations between packages, under the form of a directed graph. Nodes in the graph represent methods and the directed edge from a node to another represents that the first method calls the second. For each selected package the callgraph of the package, dependents and dependencies of dependents is generated.

GenerateCallGraph $(package)$
  $dependents \leftarrow resolveDependents(package)$
  for $d_i \in$ *dependents* **do**
    $dependencies_i \leftarrow$ *resolveDependencies*(d_i)
    $callGraph \leftarrow$
    $generateCallgraph(p, d_i, dependencies_i)$
  end
  return $callgraphs$

Algorithm 1: Algorithm that generates a callgraph

Algorithm 1 is a pseudocode representation of the Java algorithm written to generate the callgraphs.

Dependent and dependency resolution are functionalities provided by the FASTEN suite. After generation callgraphs are stored for analysis.

3.6 Analysis

In order to compute the extent of use, the non-private methods of each package need to be identified; this is done by SQL queries performed on the FASTEN database. The tables from the FASTEN database that are use are: packages - a table that

contains an unique id for each package, package_versions - a table that contains an unique id for each version of each package, modules - a table that contains information such as which modules belong to which package release and callables - a table which hosts the information of the methods analyzed by FASTEN. First, the packages table is queried to find the id of the package. Then, the package id is used as a foreign key to identify the id of the package version that is being analyzed. The id of the version is used as a foreign key to identify the ids of the modules in the package. Then methods are selected from the callables table, if the module that contains the method is found in the list of modules previously identified and the method's access is not "private" or "packagePrivate".

This process yields the wanted list of methods. This list of methods is then compared to the list of methods in each callgraph generated for each dependent of a package. The methods found in the list returned by the SQL query and not in the list of methods present in the callgraph are the methods that the dependent does not use. These methods are save after identification. The list of used method is the union between the set of methods identified by SQL queries and the set of methods in the callgraph. The list of used methods is used to create the set of modules that contain at least a function that is called.

ExtentOfUse $(package)$
- $dependents \leftarrow resolveDependents(package)$
- $nonPrivateMethods \leftarrow getMethods(package)$
- $modules \leftarrow getModules(package)$
- $EoF \leftarrow 0$
- $moduleEoF \leftarrow 0$
- **for** $d_i \in dependents$ **do**
  - $methodsInCallgraph \leftarrow getMethodsFromCallgraph(package, d_i)$
  - $unusedMethods \leftarrow nonPrivateMethods - methodsInCallgraph$
  - $usedModules \leftarrow getModule(nonPrivateMethods \cup methodsInCallgraph)$
  - $EoF \leftarrow EoF + \frac{nonPrivateMethods.size() - unusedMethods.size()}{nonPrivateMethods.size()}$
  - $moduleEoF \leftarrow moduleEoF + \frac{usedModules.size()}{modules.size()}$
- **end**
- $moduleEoF \leftarrow \frac{moduleEoF}{dependents.size()}$
- $EoF \leftarrow \frac{EoF}{dependents.size()}$
- **return** $(EoF, moduleEoF)$

Algorithm 2: Algorithm that computes extent of use

Then, the extent of use is calculated in each dependent at method and module level. After calculating the extent of use in each dependent, extent of use of the package is computed at both levels.

Algorithm 2 represents the pseudocode interpretation of the Java application implemented for this computation. EoF is an abbreviation for extent of use.

4 Results

After computing extent of use for the selected packages results can be derived. The following subsections present the results needed to answer the research questions.

To answer the first question, the distributions of extent of use (method and module level) are used. To identify relations for the last two questions, the Pearson correlation coefficient is used.

"Pearson's correlation coefficient (r) is a measure of the linear association of two variables. Correlation analysis usually starts with a graphical representation of the relation of data pairs using a scatter diagram. The values of correlation coefficient vary from –1 to +1. Positive values of correlation coefficient indicate a tendency of one variable to increase or decrease together with another variable. Negative values of correlation coefficient indicate a tendency that the increase of values of one variable is associated with the decrease of values of the other variable and vice versa. Values of correlation coefficient close to zero indicate a low association between variables, and those close to –1 or +1 indicate a strong linear association between two variables" [Kirch, 2008]. Generally, a correlation is considered strong when r $\geq$ 0.7.

Research Question 1

Figure 2 showcases the distribution of the packages over extent of use at method level.

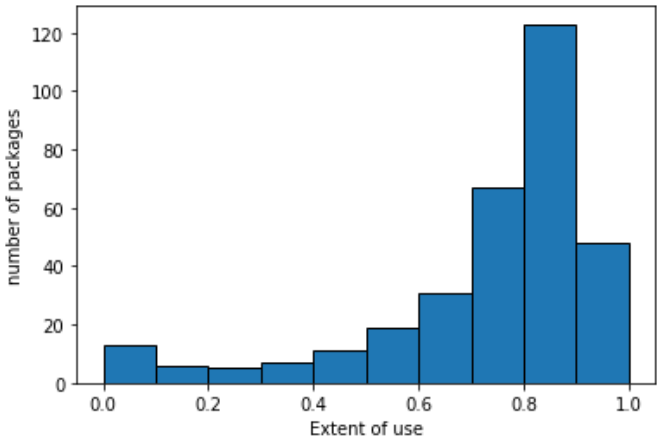

Figure 2: Extent Of Use Distribution At Method Level

It can be observed from the distribution that there exists a concentration around the 0.8-0.9 extent of use mark. The table below gives a description of the identified distribution.

Extent of use distribution at method level description	
Minimum extent of use	0
1st quantile	0.873
median	0.804
3rd quantile	0.685
Maximum extent of use	1
Mean extent of use	0.732

The average extent of use at method level is 0.732, meaning that in general, 3 out of 4 public methods are being

used in dependents. Since the median is 0.804, for half of the dataset, 4 out of 5 public methods are being used. For a package to be in the 1st quantile, it should have an extent of use of at least 0.873 (nearly 9 out of 10 public methods used).

Figure 3 showcases the distribution of the packages over extent of use at module level.

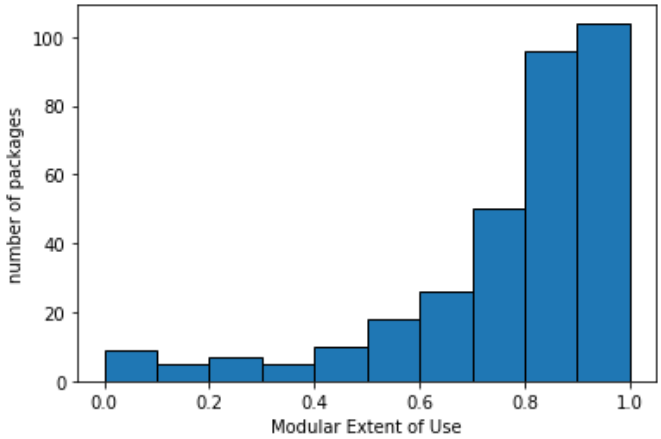

Figure 3: Extent Of Use Distribution At Module Level

The figure clearly shows that the bulk of packages in the dataset have an extent of use at module level over 0.8. The table below describes the distribution found.

Extent of use distribution at module level description	
Minimum extent of use	0
1st quantile	0.915
median	0.845
3rd quantile	0.707
Maximum extent of use	1
Mean extent of use	0.771

The average extent of use at module level is 0.771. This means that three quarters of modules in a package are generally used. The median of the distribution sits at 0.845, suggesting that for half of the packages in the dataset, over 84% of their modules are used.

Research Question 2

Figure 4 presents extent of use at method level plotted against the number of dependents the package has.

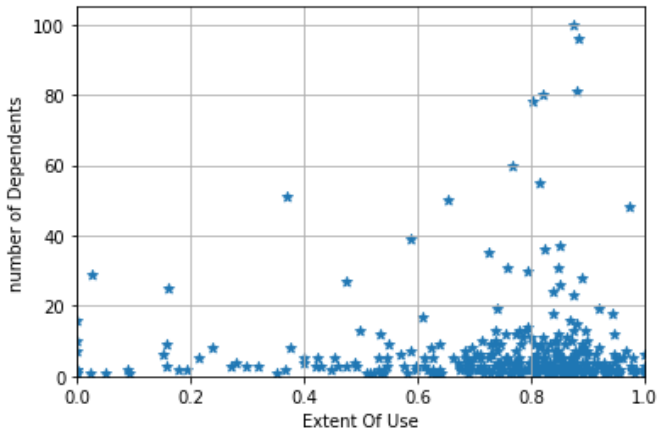

Figure 4: Extent of use at method level plotted against the number of dependents

It can be observed from the figure that most packages have an extent of use around 0.8 at method level regardless of the number of dependents. This observation is sustained by the low value of the Pearson correlation coefficient between number of dependents and extent of use at method level: **r = 0.038**.

Figure 5 presents extent of use at module level plotted against the number of dependents the package has.

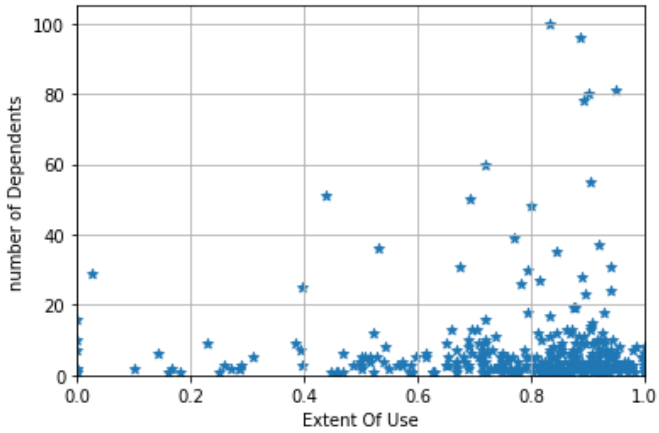

Figure 5: Extent of use at module level plotted against the number of dependents

Once again, the figure shows a grouping of packages, this time around 0.9, regardless of the number of dependents. This observation is validated by the low value of the Pearson correlation coefficient: **r = 0.241**.

Although the correlation at module level is bigger than the one at method level, because both values are so small, it can be concluded that these correlations arise because of the limited scope of this study. Thus, there is no relation between the number of dependents and extent of use at the studied levels.

Research Question 3

Figure 6 shows a plot with extent of use at module level on the y axis and extent of use at method level on the x axis.

The plot helps us visualise the relationship between extent of use at the two levels. The Pearson correlation coefficient for these two variables is **r = 0.829**. This value is high enough to conclude that extent of use at method level is related to the extent of use at module level.

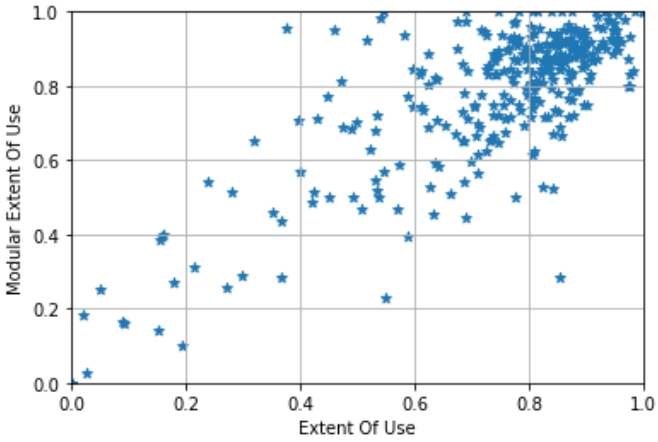

Figure 6: Extent of use at module level plotted against the extent of use at method level

5 Responsible Research

Meaningful advancements in science can be made only through responsible research. Results from this study can be replicated by one who has access to the FASTEN database. To aid with result replication, a replication package containing the analyzed packages and the analysis code was realised. Contact the author of the study to receive the replication package. The scope of this research does not pose critical ethical concerns. Data selection was made trying to avoid skewness or biases. The code written to accomplish the analysis was written to the best abilities of the author.

6 Discussion and conclusions

Based on the results presented in section 4 we can draw conclusions about extent of use in the Maven Central Repository. At both method level and module level, extent of use is high: three quarters of available methods and three quarters of modules are in use. Since these are used in productions, one expects that this code is bug-free.

Still, it seems that, on average, one in four modules remains unused in each dependent. This leads to the idea that instead of importing whole packages, there may exist a better strategy based on only importing used modules. [Pashchenko *et al.*, 2018] presents the problem of vulnerabilities in open-source software. [Soto-Valero *et al.*, 2020] is a study focused on identifying causes for bloated dependencies. Bloated dependencies in an application are defined as dependencies that are not needed to compile, run or build the application. The cited study concludes that 81.7% of the bloated dependencies are transitive dependencies. Reducing the import volume could be a strategy in mitigating bloated dependencies and inherited vulnerabilities.

Results have shown that there is no relation between extent of use at method/module level and the number of dependents. This is not a surprising result, considering that it is normal for dependents to use the package in similar ways, with similar extents.

The correlation between extent of use at method level and extent of use at module level is strong, based on the Pearson correlation coefficient calculated in the results section. This means that used methods are generally spread between modules, instead of having a single module hosting most used methods. This is a sign of qualitative design of packages in the Maven Central Repository. Future works could prove correlations between quality of code metrics and high extent of use.

Extent of use can be a metric used to decide whether or not to use an available solution, instead of creating one. When considering to use an already available solution from a package, one can compute in which extent they will use that package. If extent of use is very low (i.e. under 0.05), it might be better to write an own solution to avoid unnecessary bloating the code.

This study is subject to a number of limitations. Even though the dataset is considered representative, the limited scope of this research (384 packages) makes it hard to draw hard conclusions for the whole of Maven Central Repository. The Maven ecosystem is constantly evolving so data derived within this study might become redundant. Future works could study the evolution of extent of use in package management systems.

In conclusion, extent of use in the representative sample of the Maven Central Repository is elevated: 0.732 at method level and 0.771 at module level. The strong relation identified between extent of use at method and module level is natural and a sign of good package design.

References

[FASTEN, 2022] FASTEN. https://github.com/fasten-project, 2022.

[Kirch, 2008] Wilhelm Kirch, editor. *Pearson's Correlation Coefficient*, pages 1090–1091. Springer Netherlands, Dordrecht, 2008.

[Kula *et al.*, 2015] Raula Kula, Daniel German, Takashi Ishio, and Katsuro Inoue. Trusting a library: A study of the latency to adopt the latest maven release. *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

[Kula *et al.*, 2017a] Raula Kula, Daniel German, Takashi Ishio, Ali Ouni, and Katsuro Inoue. An exploratory study on library aging by monitoring client usage in a software ecosystem. *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.

[Kula *et al.*, 2017b] Raula Gaikovina Kula, Coen De Roover, Daniel M. Germán, Takashi Ishio, and Katsuro Inoue.

Modeling library popularity within a software ecosystem. *Tech. Rep.*, 2017.

[Kula *et al.*, 2018] Raula Kula, Daniel German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 2018.

[Pashchenko *et al.*, 2018] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, New York, NY, USA, 2018. Association for Computing Machinery.

[Raemaekers *et al.*, 2012] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. *28th IEEE International Conference on Software Maintenance (ICSM)*, 2012.

[Raemaekers *et al.*, 2013] Steven Raemaekers, Arie van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. *10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[Raemaekers *et al.*, 2017] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, 2017.

[Sajnani *et al.*, 2014] Hitesh Sajnani, Vaibhav Saini, Joel Ossher, and Cristina V. Lopes. Is popularity a measure of quality? an analysis of maven components. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 231–240, 2014.

[Soto-Valero *et al.*, 2020] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, 2020.