Library Utilization Metrics for Maven Projects

Maria Kolyda[1], Eirini Kostoglou[1], Nikolaos Nikolaidis[1[0000-0002-7958-9393]], Apostolos Ampatzoglou[1[0000-0002-5764-7302]], Alexander Chatzigeorgiou[1[0000-0002-5381-8418]]

[1] Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

Abstract. In modern software development, usually, reuse takes place by invoking in the codebase, methods that are deployed and imported into projects as 3rd party libraries. The ease with which one can take benefit of reuse of libraries has been simplified lately, by platforms such as Maven, Gradle, etc. However, this convenient choice in many cases leads to an overwhelming number of libraries being packed in the final executable, even when not needed (e.g., the code that uses originally invoked a library is removed, or it is dead). In this paper, we propose 5 metrics that capture the extent to which each library is utilized in the codebase, providing information to the software engineers on the actual utility of the library in the final product. To automate the calculation of these metrics we have developed a corresponding tool that can be used for quality monitoring purposes.
Video: https://youtu.be/m1N22F5mbHI
Code Frontend: https://github.com/kostoglou/LibraryUtilization
Code Backend: https://github.com/MariaKolyda/javaLibraryUtilization
Running Instance: http://195.251.210.147:3005

Keywords: libraries, metrics, maven, library utilization, app evaluation.

1 Introduction

The usage of third-party libraries in software development is a wildly used practice to speed-up the development process, and in turn reduce costs [7]. This practice only gets more popular with the rise of easy-to-use build automation tools and library repositories like Maven, Gradle, NPM, etc. Libraries provide already created and tested functionalities, so developers do not need to write code from scratch, but rather find an appropriate library. However, as any other benefit, the reuse of libraries comes with a cost, or at least with a threat of a cost. Based on the literature, the use of 3rd party libraries is a living part of software development, in the sense that libraries can be added, upgraded, or removed along evolution, and similarly does the code around them [17].

The excessive and unnecessary use of third-party libraries can lead to three main problems: (a) the size of the target system grows larger in size, hurting the performance and resource utilization of the software; (b) the third party library might bring vulnerabilities into the target system; and (c) the external quality of the library cannot be controlled, in the sense that in the majority of the cases, third-party library reuse is black-box. As examples of excessive and unnecessary use of third-party libraries, the following cases can be considered. First, along with evolution, there is a chance that some

libraries become ***unused*** after some source code update. In other words, either the code that was invoking a method declared from a library is removed, or it becomes dead code. However, if the development team does not remove the library from the build automation system, the library will remain a part of the build process. Therefore, it is important to keep track of the libraries that are used in the target system, and the extent of their utilization—in some cases, it might be beneficial to implement something from scratch, if a very small fraction of a library is utilized. Additionally, as an ***unnecessary upgrade of the library***, we can consider an upgrade to a newer version of a library that does not offer any functional or non-functional benefit. Therefore, it is important to monitor along evolution if the level of library utilization is not decreasing over time—i.e., importing larger libraries that are not used more.

In this paper, we propose five novel metrics that assess the level of library utilization in a software project (see Section 3) and develop a tool (presented in Section 4) for automating their calculation, to boost their adoption in practice. Finally, in Section 5, we present an initial validation of the tool.

2 Related Work

A lot of information about library reuse and metrics can be found in the literature, thus we try to present some of those studies in this section. Firstly, Mora et al. [4] [5] provided a way that compares libraries to help the developers with selecting the most suitable each time. To achieve this comparison, they used 9 metrics for each library and asked a total of 61 developers to evaluate them. They found out that developers are more interested in metrics related to the popularity, security, and performance of libraries. But this can change a bit depending on the domain of the application under development. A similar study was conducted by Vargas et al. [10], where they studied the factors that influence the selection process of libraries. They asked 115 developers for feedback on a total of 26 factors, which in turn could be used as metrics. Also, they grouped these factors into three categories namely: technical, human, and economic. Finally, similar types of metrics with an emphasis on performance, usability, documentation, and popularity were proposed in other studies as well [1] [8] [9] [12].

Moreover, Washizaki et al. [16] viewed the libraries as black-box reuse and focused on metrics based on the limited information that can be obtained from outside of the components without any source code. They defined five metrics and through evaluation experiments, it was found that these metrics can effectively identify black-box components with high reusability. These metrics are: EMI (Existence of Meta-Information), RCO (Rate of Component Observability), RCC (Rate of Component Customizability), SCCr (Self-Completeness of Component's Return Value), and SCCp (Self-Completeness of Component's Parameter). A similar black-box approach was selected by Shatnawi et al. [14], where they proposed a model consisting of three metrics. These metrics are related more to the business side and are the library investment ratio, the library investment level, and program simplicity.

In contrast to the previously mentioned studies that aim at assessing the quality of the libraries per se, to aid developers in library selection, in this work, we focus on the

target system, and we assess the effectiveness of reuse—i.e., the level to which a library is utilized in each system. Therefore, even for exactly the same library, the metric scores would be different for different systems, since the way that the library is used is being assessed, rather than the library per se.

3 Proposed Metrics

In this section, we present the proposed metrics that can be used to assess the level of library utilization in a specific project. Most of these metrics rely on the entry points of a library, used in a specific project, as well as the call-tree that is parsed by invoking these methods (i.e., the subsequent series of method calls made inside the library to provide the needed functionality). Similar approaches can be found in other studies that calculate the call-tree, e.g., for assessing the Technical Debt (TD) of service, based on the entry points of services (end-points) and the methods that are subsequently invoked by the API call [11]. To explain the proposed metrics, we provide an illustrative example in Figure 1. We should note that each circle represents a class, while the number inside the circle represents the methods of that class. Finally, the connection between the classes represents the called methods from one class to another, and the different colors are used for each call-tree.

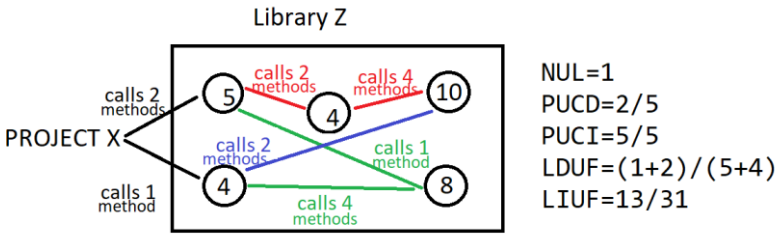

Fig. 1. Illustrative example for all metrics

Number of Used Libraries (NUL). The first proposed metric is calculated at the project level, and as the name implies is the number of the used libraries from one project. In our example, we can see that Project X used classes only from one library (Library Z). So, the value of NUL is 1. This number provides an indication of how much the project depends on third-party code—related to the performance and resource utilization of the executable.

Percentage of Used Classes Directly (PUCD). To measure the utilization of a library we proposed the PUCD metric, which calculates the percentage of used classes from a given project. We should note that for the calculation of this metric, we consider only the classes that are being used directly from the given target projects. So, in our example since Project X uses only two classes out of 5, the PUCD is 2/5 or 40%. This metric is related to the extent to which the quality of the target system might be affected by the third-party code.

Percentage of Used Classes Indirectly (PUCI). To measure the usability of the whole library, by considering all the classes that are being used, we proposed the PUCI metric. For the calculation of this metric, we consider all the classes that are being accessed, even indirectly from the examined project. So, the value of PUCI is 5/5 or 100%, because in our example all 5 classes are being used. This metric is related to the extent to which the quality of the target system might be affected by the third-party code. The same discrimination between direct and indirect dependencies, can be found in traditional coupling metrics, as well (e.g., TCC and LCC [3]).

Library Direct Utilization Factor (LDUF). To measure the utilization of a library we proposed the LDUF metric, which calculates the percentage of used methods out of the total number of methods that the used classes have. We should note that in this metric we do not consider the indirect methods that are being used. In the given example, Project X calls in total 3 methods (2 from the first class, and 1 from the second one) of Library Z, and these classes have 9 methods in total (the first one has 5, and the second one has 4). So, the value of LDUF is 3/9 or 33.3%. This metric can act as an indicator of the "*worth*" of reusing the library, based on its fraction that is reused in practice.

Library Indirect Utilization Factor (LIUF). Finally, we created the LIUF metric, which considers the indirect utilization of a library. To achieve this, we trace all the method calls that take place, and we find the number of used methods of each class. In the same example from Library Z 13 methods are being used out of the total 31 methods, so the value of LIUF is 13/31 or 41.9%. In more detail, we can see that the red call tree calls 6 methods, the blue one calls 2 methods, and the green calls 5 methods. Also, we should note that we do not count more than once a used method. This metric can act as an indicator of the "*worth*" of reusing the library, based on the fraction that is reused.

4 Library Utilization Tool

For the calculation of the proposed metrics, we created the Library Utilization tool. This tool was created as a web application, with a front-end, written in React and a back-end written in Java and the Spring framework. The web service exposes all the necessary functionalities through a RESTful API, whereas the web app makes the appropriate requests and demonstrates the appropriate results and views to the user. We should note that both the frontend[1] and backend[2] projects can be found online, along with a video[3], which presents all the functionalities. The main functionalities of the application are the following: (a) analyze a project, (b) inspect the metrics scores, (c) inspect the call-tree of a method call, and (d) analyze the history of a project.

Analyze a project. When the users open the web application, they are greeted with the screen of Figure 2, from where they can start a new analysis. By providing the Git URL of the project they want to analyze, they can start a new static analysis in the last commit of the project or get the last already analyzed commit (if any exist). The analysis of a

[1] https://github.com/kostoglou/LibraryUtilization
[2] https://github.com/MariaKolyda/javaLibraryUtilization
[3] https://youtu.be/m1N22F5mbHI

project is time-consuming, especially for big projects with a lot of libraries. To calculate all the proposed metrics, we must analyze the code of the project and all the libraries that are being used. To be able to get the code of each library we had to limit our application to analyze only Maven project (at least for a first release). Moreover, we had to analyze the code of the project, to get the used methods of each library, and the code of each library, to get the call-tree of the methods. To this end, we used the *JavaParser* library [15], which is a very well-known parsing library for Java projects. Finally, we should note that once a project is analyzed it is saved in a database, so in case a user asks for an already analyzed project the results can be provided almost instantly.

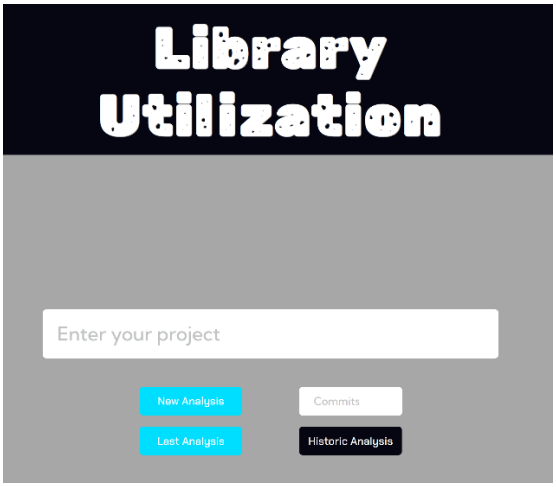

Fig. 2. Analyze Project Screen

Inspect the Metric Scores. Once the project analysis is finished, or the results are retrieved from the database, the user is presented with the metric scores (see Figure 3). The user can see the NUL of the project and for each library the values of the other four metrics. This feature provides a basic and bird-eye view on the analysis.

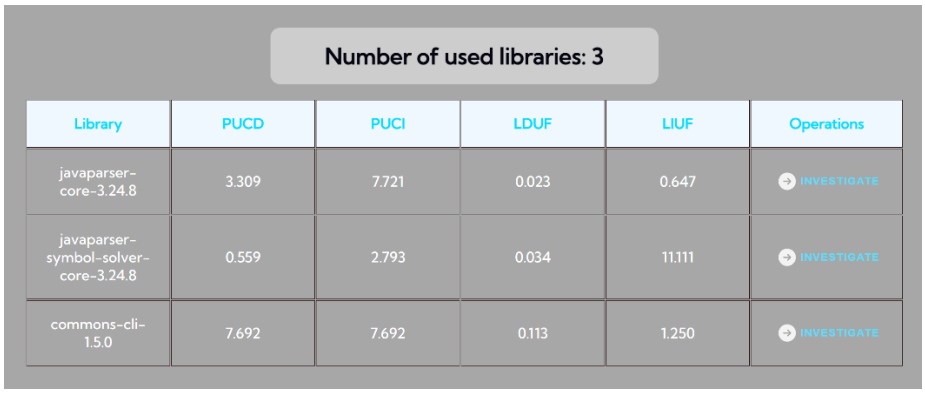

Fig. 3. Metrics Results

Inspect the call-tree of a method call. By selecting the “Investigate” button for one library, the user can see all the methods that were used from this library. And by selecting one, the user can see the call-tree of that method (see Figure 4). Moreover, a slider is provided from which the user can specify the number of nodes they want to see for a given call-tree. The nodes are limited to a max number of 800 since after that the graph is hard to read and maybe not so useful. This function can be useful for inspecting out of which method calls, tentatively malicious or low-quality methods are being invoked, affecting the external behavior of the target system. Through this feature, the engineer can get a hint of which functionalities might need to be re-written from scratch, in case of a run-time quality problem.

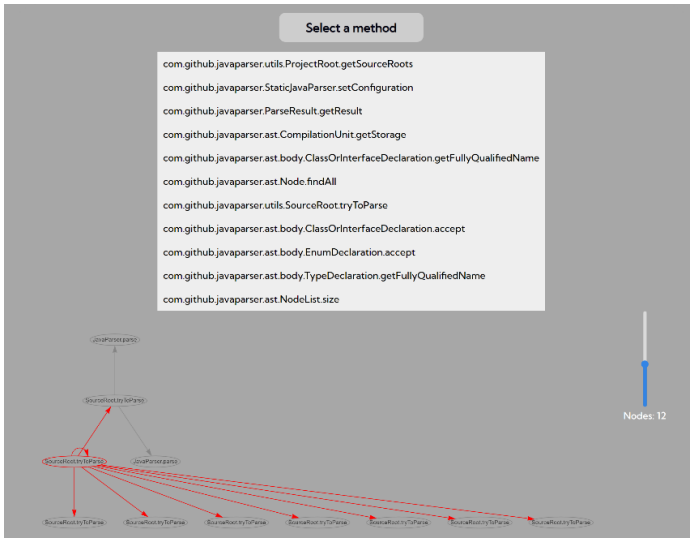

Fig. 4. Call-Graph Representation

Fig. 5. Evolution Analysis

Analyze Project History. The analysis of all the commits of a project is not recommended due to time constraints, and since we do not expect there to be a big change in every commit from the aspect of library utilization. For this reason, in the historical analysis, the user should provide the number of commits they want to analyze along with the Git URL. The commits that are going to be analyzed will be spread out to the history of the project according to the provided number. Once the historic analysis is completed, the user is presented with the results like in Figure 5. The users can see the evolution of NUL in a line chart and a table with all the libraries that were used along with their commits. Finally, by selecting a specific library they can see the evolution of the four-remaining metrics. This feature can be interesting for seeing if the level of library utilization stays constant along evolution, or if the library grows or shrinks, but no additional features are being exploited.

5 Validation of the Metrics and Tool

To evaluate the proposed metrics and the tool, we have performed an initial exploratory empirical study, that was designed and reported based on the guidelines of Runeson et al. for case studies [13].

5.1 Study Design

To study the proposed metrics and tool, with respect to their tentative acceptance in the industry, in terms of real word systems and the relevance of the idea, we have formulated the following research question: "*Does the developed library utilization tool meet the expectations of the practitioners?*"
The validation of the tool was conducted by asking 13 senior software developers, from 5 different companies, to use and evaluate the tool in a 1-day workshop. First, the researchers have presented the tool, as well as the envisioned motivation and usage scenarios. Then, the practitioners were given a small task to familiarize themselves with the tool, and then some extra time to experiment independently. To assess the relevance and usability of the tool, we provided access to the participants to an online instance of the web application. They were asked to perform several tasks and interact with the application, to get hands-on experience. Each of the participants was asked to do the following: (a) create a new analysis; (b) inspect the results and the call tree; and (c) inspect the results of the evolution analysis. Then, they were asked to brainstorm on what they have learned from using the tool in the form of a focus group, using the whiteboard. The focus group and the discussion were moderated by the researchers.

The workshop closed with the participant filling in a small questionnaire at the end. The evaluation of the relevance of the metrics and the usability of the tool was performed based on the System Usability Scale (SUS) instrument [2].

5.2 Results

The results of the evaluation are presented in Figure 6 based on SUS. We can see that all of the questions received excellent responses, however the frequency of the application usage received a little bit more unfavorable feedback. The participants seem to understand the main disadvantage of the application, which is the time needed for a new analysis, but they were not displeased about it. In a Q&A that was followed with some of the participants, we could see the need for supporting more languages and library registries. Moreover, as for the frequency of use of the tool, the participants told us that they do not often add new libraries or change the methods that they use. So, it is normal to not have to use often a tool like this in their daily routines, but mostly as a complementary analysis during quality control processes (e.g., before releases or end of sprints).

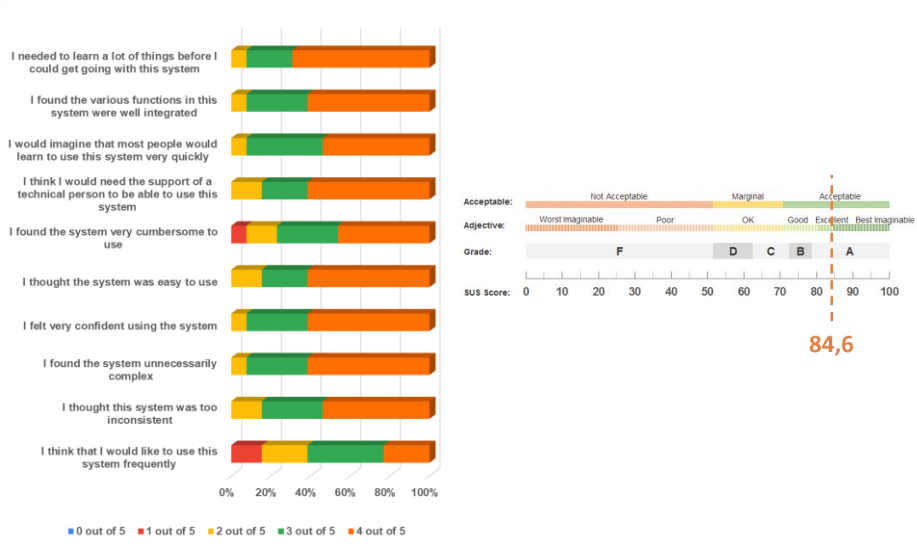

Fig. 6. Usability of Proposed Metrics and Tool

6 Conclusions

The development using libraries in software engineering is a widely adopted practice, but it must be monitored. The usage of many dependencies or wrong ones can lead to a lot of problems, so several metrics exist to measure some aspects of the libraries. In this paper, we have introduced five metrics to fill the gap in the utilization aspect of a library for a given project. We also created a tool for the calculation and presentation of these metrics, and we provided it as a web application. Industrial validation took place with 13 developers from 5 companies, to assess the usability of the created tool. The results showed that the tool is usable and liked by the participants, with the only concern being the frequency that they would use it.

Acknowledgements

Work reported in this paper has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 780572 (project SDK4ED)

References

1. Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., Shihab, E.: Why do de- velopers use trivial packages? an empirical case study on npm. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering. pp. 385–395 (2017)
2. Brooke, J.: Sus: a "quick and dirty'usability. Usability evaluation in industry 189(3), 189–194 (1996)
3. Charalampidou, S., Arvanitou, E.M., Ampatzoglou, A., Avgeriou, P., Chatzi- georgiou, A., Stamelos, I.: Structural quality metrics as indicators of the long method bad smell: An empirical study. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 234–238 (2018). https://doi.org/10.1109/SEAA.2018.00046
4. De la Mora, F.L., Nadi, S.: An empirical study of metric-based comparisons of soft- ware libraries. In: Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 22–31 (2018)
5. De La Mora, F.L., Nadi, S.: Which library should i use? a metric-based compari- son of software libraries. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results. pp. 37–40 (2018)
6. Flov´en, K.F.: State management models impact on run-time performance in single page applications (2020)
7. Frakes, W.B., Kang, K.: Software reuse research: Status and future. IEEE trans- actions on Software Engineering 31(7), 529–536 (2005)
8. Gizas, A., Christodoulou, S., Papatheodorou, T.: Comparative evaluation of javascript frameworks. In: Proceedings of the 21st International Conference on World Wide Web. pp. 513–514 (2012)
9. Hora, A., Valente, M.T.: apiwave: Keeping track of api popularity and migration. In: 2015 IEEE international conference on software maintenance and evolution (ICSME). pp. 321–323. IEEE (2015)
10. Larios Vargas, E., Aniche, M., Treude, C., Bruntink, M., Gousios, G.: Selecting third-party libraries: The practitioners' perspective. In: Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. pp. 245–256 (2020)
11. Nikolaidis, N., Ampatzoglou, A., Chatzigeorgiou, A., Tsekeridou, S., Piperidis, A.: Technical debt in service-oriented software systems. In: International Conference on Product-Focused Software Process Improvement. pp. 265–281. Springer (2022)
12. Piccioni, M., Furia, C.A., Meyer, B.: An empirical study of api usability. In: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 5–14. IEEE (2013)
13. Runeson, P., Host, M., Rainer, A., Regnell, B.: Case study research in software engineering: Guidelines and examples. John Wiley & Sons (2012)
14. Shatnawi, M.Q., Hmeidi, I., Shatnawi, A.: Software library investment metrics: a new approach, issues and recommendations (2017)

15. Smith, N., Van Bruggen, D., Tomassetti, F.: Javaparser: visited. Leanpub, oct. de 10, 29–40 (2017)
16. Washizaki, H., Yamamoto, H., Fukazawa, Y.: A metrics suite for measuring reusability of software components. In: Proceedings. 5th International Workshop on enterprise networking and computing in healthcare industry (IEEE Cat. No. 03EX717). pp. 211–223. IEEE (2004)
17. Zaimi, A., Ampatzoglou, A., Triantafyllidou, N., Chatzigeorgiou, A., Mavridis, A., Chaikalis, T., Deligiannis, I., Sfetsos, P., Stamelos, I.: An empirical study on the reuse of third-party libraries in open-source soft- ware development. In: Proceedings of the 7th Balkan Conference on In- formatics Conference. BCI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2801081.2801087, https://doi.org/10.1145/2801081.2801087