LibComp: An IntelliJ Plugin for Comparing Java Libraries

Rehab El-Hajj
relhajj@ualberta.ca
University of Alberta
Edmonton, Alberta, Canada

Sarah Nadi
nadi@ualberta.ca
University of Alberta
Edmonton, Alberta, Canada

ABSTRACT

Software developers heavily rely on third-party libraries to accomplish their programming tasks. Since many libraries offer similar functionality, it can be difficult and tedious for developers differentiate similar libraries in order to select the most suitable one. In our previous work, we proposed the idea of metric-based library comparisons that allow developers to compare various aspects of libraries within the same domain, empowering them with information to aid with their decision. In this paper we present an IntelliJ plugin, LibComp, that provides this library metric-based comparison technique right within the developer's IDE. As soon as a developer adds a library dependency that LibComp has information about, LibComp will highlight this dependency to let the developer know that there are alternatives available. Once the user triggers the comparison for that library, they can view various metrics about the library and its alternatives and decide if they want to use one of the alternatives. In the process, LibComp also records the number of times the developer invokes the tool and any completed replacements. Such feedback, if optionally sent to us by the developer, provides us valuable insights into developers' replacement decisions as well as information on how we can improve the tool. A video demonstrating the usage of LibComp can be found at https://youtu.be/YtEEdJan77A

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Software maintenance tools**.

KEYWORDS

IntelliJ plugin, software library comparisons, software aspects

ACM Reference Format:
Rehab El-Hajj and Sarah Nadi. 2020. LibComp: An IntelliJ Plugin for Comparing Java Libraries. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3368089.3417922

1 INTRODUCTION

Most software applications rely on third-party components, such as software libraries, to implement various functionality. Choosing the right library is an important decision; relying on a library that is, for example, not maintained or that has known problems can lead to high maintenance costs down the line when developers are forced to replace this library or deal with rippling effects of the problems. However, it is not always clear how to compare and select the best library from a seemingly similar set of available libraries. Thus, developers often spend time to understand differences between available libraries by, for example, reading the libraries' documentation or searching for blogs or posts on Stack Overflow [17, 22]. To save developers' time and address their need to compare libraries, we previously proposed the idea of *metric-based library comparisons* [7, 8] where we compiled a set of metrics that can be used to compare libraries. After surveying 61 developers and gathering feedback about the metrics and comparison [8], we built a website that developers can use to compare libraries in the same domain [6].

While such a website is useful, it also means that every time a developer wants to compare libraries, they are forced to leave their Integrated Development Environment (IDE) to a completely separate environment. Thus, to make our library comparisons more accessible to developers, this tool paper contributes our metric-based Java library comparisons as an IntelliJ plugin, LibComp.

To help developers compare similar libraries, LibComp displays several libraries from the same domain (such as testing, databases, logging etc.) via a visual and interactive dialog shown in Figure 1. This dialog compares libraries across various metrics such as popularity, release frequency, as well as issue response and closing rates. LibComp empowers developers with information that allows them to make informative decisions about the libraries they use.

While our existing website allows users to upvote/downvote metrics, it does not capture the developer's final decision or if they were trying to replace an existing library. To overcome this, LibComp also gathers feedback about how the metric-based comparison is being used by developers. Whenever the user triggers LibComp, we record the domain being compared, the libraries being compared, and if any library replacement took place. If the developer

Libraries from the Mocking domain

Metrics	mockito (Current Library)	easymock	jmock-library	powermock
Popularity (Repos)	282/1000 Repos	27/1000 Repos	10/1000 Repos	59/1000 Repos
Release Frequency (Days)	8 Days	205 Days	33 Days	52 Days
Issue Closing Time (Days)	93 Days	66 Days	193 Days	46 Days
Issue Response Time (Days)	27 Days	92 Days	85 Days	174 Days
Backwards Compatibility	31 Breaking Changes	46 Breaking Changes	72 Breaking Changes	130 Breaking Changes
Security (Percent)	0.00 %	0.00 %	0.00 %	0.11 %
Performance (Percent)	1.59 %	1.36 %	0.00 %	0.99 %
Last Stack Overflow Post	2020-08-26	2020-08-07	2019-12-25	2020-08-25
Last Modification Date	2020-08-25	2020-04-10	2020-08-20	2020-03-30
License	MIT License	None	BSD 3-Clause "New" or "R...	Apache License 2.0
Overall Score	4.40/5	3.29/5	3.54/5	3.46/5

Cancel | Replace org.mockito Package with org.jmock

Figure 1: LibComp's main metric-based library comparison dialog. It displays data on various metrics for the selected library and all alternative libraries within the same domain.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7043-1/20/11...$15.00
https://doi.org/10.1145/3368089.3417922

authorizes sharing this data, they can send the collected feedback to our server[1]. The collected feedback can help us understand how developers compare libraries and what common replacements are, and also help us improve LibComp in the future.

LibComp is open-source on GitHub under an MIT license [12], with instructions on how to install and use the plugin. It currently supports 50 Java libraries from 10 domains, but can be easily extended to other languages in the future. LibComp currently detects usages of these supported libraries through import statements as well as dependencies in gradle or maven build files.

2 RELATED WORK

There is a lot of existing work that investigates the aspects developers take into account while selecting a library [17, 21, 23]; however, most of that work does not present any tool for end users. Many of our metrics are inspired by these discovered aspects; our previous work detailed these relationships and differences [7, 8]. In this section, we focus on the tooling perspective of existing information related to library comparison. Many comparisons, similar to how we started, are presented in a website form. For example, Hora and Valente [16] show Application Programming Interface (API) popularity and API migration information on a website. Similarly, Uddin and Komh present their Opiner tool [24], which mines developers' opinions on libraries from Stack Overflow, as a website. Lin et al. [18] also develop an Stack Overflow opinion mining approach, but do not build an actual tool/website for end users to use. In contrast to the above work and other existing work, our current metric-based comparison technique provides developers with factual information on various library aspects, and is based on multiple information sources. More importantly, to the best of our knowledge, none of the existing comparisons provide integrated support tools. To make research techniques more accessible to developers, various kinds of software development support tools have been integrated as IDE plugins (e.g., [3, 15, 20]). Motivated by making our metric-based comparisons more accessible, we also design LibComp as a plugin for the popular Java IDE, IntelliJ.

3 METRIC-BASED LIBRARY COMPARISON

LibComp delivers the metrics we developed in our previous work [7, 8]. Note that all these metrics rely on mining information from the library's version-control system, issue-tracking system, or Stack Overflow questions. Thus, we can collect information only about open-source libraries with these repositories available. The details of how each metric is calculated is explained in our previous work and all the scripts are open source [5]. In this section, we provide a brief summary of these metrics. We also clarify if any metrics or information has changed since our original publications.

- ***Popularity*** is the number of projects using a library. In our previous implementation [8], we used BOA [11] to mine such usage based on import statements. However, since the BOA data set is not frequently updated, we changed the calculation to search for import usage in the top 1000 starred GitHub repositories.
- ***Release Frequency*** is the average time in days between two consecutive releases of a library.

[1]Note that we received ethics clearance for this tool from our university

- ***Last Modification Date*** is the date of the last commit in a library's repository.
- **Performance** is measured with a heuristic calculating the percentage of performance-related issues of a library (as determined by a machine-learning classifier).
- ***Security*** is measured with a heuristic calculating the percentage of security related issues of a library (as determined by a machine-learning classifier).
- ***Issue Response Time*** is the average time, in days, to get the first response to a reported issue.
- ***Issue Closing Time*** is the average time, in days, to close issues.
- ***Backwards Compatibility*** is the average number of breaking API changes between two consecutive releases, calculated using Xavier et. al's work [25].
- ***Last Discussed on Stack Overflow*** is the time since a question tagged with this library has been asked on Stack Overflow.
- ***License*** is the library's license on GitHub. This is new information we include based on feedback from our previous survey [8].
- ***Overall Score*** is a also a new metric, which provides an overall rating out of 5 stars for each library [19]. Briefly, each of the above metrics gets a normalized weight between 0 and 1 depending on its semantics, where 1 is the maximum best value. We then calculate the overall library score as the sum of each metric score divided by the number of metrics and scaled to 5.

A scheduled cron job that automatically calculates the above metrics runs on a monthly basis and updates the data displayed on our website [6]. We also provide charts to compare the libraries and show evolution trends. For example, the popularity chart shows the evolution of a library's popularity, based on our monthly recalculation of the metrics. On the other hand, the release frequency chart shows the dates of the previous releases. Such charts were requested by our previous survey participants [8] as a way to help them better interpret the metrics.

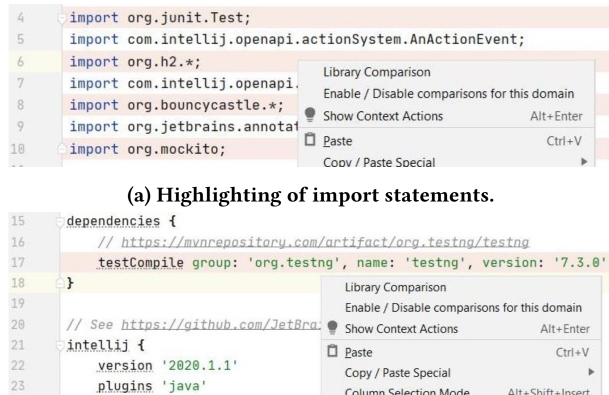
4 LIBCOMP OVERVIEW

In this section, we present LibComp from the end-user's perspective, that is the developer who is developing some application in IntelliJ.

4.1 Usage Overview

LibComp can be installed as an IntelliJ plugin using the package we provide on our GitHub repository [12]. After installation, when a user opens an IntelliJ project, LibComp automatically starts analyzing all open Java files and, if available, the build.gradle or pom.xml file in the project. Any import statements or gradle/maven library dependencies we have information about in our database get highlighted in pink, as shown in Figure 2. The developer can right click on any of these highlighted statements and select "Library Comparison" from the menu. A dialog that displays the metric-based comparison for all the libraries in the domain corresponding to the highlighted library pops up in the centre of the editor.

Figure 1 shows the dialog that will appear when the developer clicks on the highlighted `import org.mockito.*` statement. The comparison in the dialog displays all the libraries we have information on in the Mocking domain, which is the domain the clicked `import org.mockito.*` statement (seen in the bottom of Figure 2a) belongs to. The already imported library is specified as the



(a) Highlighting of import statements.

(b) Dependency highlighting in build.gradle (similarly for pom.xml).

Figure 2: LibComp's dependency highlighting. User right clicks on highlighted lines and clicks "Library Comparison" to open the comparison dialog shown in Figure 1.

"Current Library" as seen in the first column in Figure 1. While this dialog is open, the user has the ability to compare the metric data for the alternative libraries with the selected library, open a chart to see a visualization of the data by clicking the chart icon on the far left of the dialog, or sort the libraries by a specific metric by clicking either the red or green downward/upward arrows for the metric of their choice. Based on the provided information, the developer is able to compare the metric data and come to an informed decision on whether or not they would like to change the current library. The user can hover over any of the metric columns to get a short description about them. If the developer chooses to replace the current library, she can choose one of the alternative libraries and click on the "Replace" button. In our example in Figure 1, the developer chose the jmock-library library (as indicated with the blue highlighting) as a replacement.

Once the replace button is clicked, the comparison dialog will close and the originally selected import statement is replaced with an import statement corresponding to the newly chosen library. If the developer instead decides against replacing the selected library, they can click cancel, which will close the dialog and nothing in the editor will change. LibComp is always analyzing the Java files in the background. If any new import statements are added which we have information on, they will also be highlighted. The same flow applies for build.gradle/pom.xml files, but instead of highlighting import statements, LibComp highlights added dependencies, as shown in Figure 2b. The user can open the same dialog and replace the library if they choose to do so.

4.2 LibComp Interaction Data at a Glance

LibComp tracks how the user interacts with the plugin. This allows us to collect feedback data in order to learn what types of library replacements occur and how our comparisons affect this. Each developer who uses LibComp can fill out a user profile (inspired by previous work [2]) in the form shown in Figure 3a, which can be accessed from the main LibComp menu. To then send us their collected feedback data, the developer must give us explicit permission by going to the "Send Feedback" dialog shown in Figure 3b. Using this dialog, the developer can read the terms and conditions of the collected data. The developer needs to explicitly acknowledge these terms before being allowed to send us the data. Section 5 provides more details on the exact feedback we collect.

5 IMPLEMENTATION DETAILS

To obtain the data we display in LibComp, we built a REST API that communicates with our server, which also hosts our comparison website. However, to avoid always relying on internet connectivity, LibComp saves the latest data it retrieves in a local JSON file. Whenever a project is opened, LibComp sends a request to check whether the data on our server has been updated; if it has, LibComp requests the new library metric data and stores it locally. Otherwise, it uses the local version of the data it already has. Currently, LibComp presents data for 50 Java libraries from 10 domains [8]. Each of these libraries already has a package name associated with it. For example, the package name associated with the Google Guava library from the Utilities domain is `com.google.common` while that associated with the Deeplearning4J library from the machine learning domain is `org.deeplearning4j`. This package name helps LibComp identify matching libraries. To extend LibComp with additional libraries in the future, a contributor only needs to add a new library entry to the LibraryData.json file in our scripts repository [13] with all the above relevant information such as the library package, domain, and GitHub URL. The new entry will then be included in the next monthly update of the data.

To support library comparisons in Java files, LibComp uses IntelliJ's Project Structure Interface (PSI) to obtain imported packages. PSI provides us an abstract syntax tree representation of a source code file [10], which allows us to collect the set of import statements. For each detected import in a Java file, LibComp identifies the base package of that import statement (e.g., base package for `import org.junit.Test` is `org.junit`) and checks its local library data to identify if there are any libraries corresponding to this package. Such libraries will be highlighted as shown in Figure 2a.

To support library comparisons in gradle build files, LibComp parses the build.gradle file (or pom.xml file in maven projects) to obtain all dependencies. For each listed dependency, LibComp compares the package specified in the `group` attribute to its local library data. Any dependencies with matched libraries will be highlighted. If the developer triggers LibComp from within the build file and decides to replace the current library, LibComp will replace the group and name of the dependency with those for the alternate library. For the dependency version number, LibComp queries Maven Repository [1] to obtain the latest version for the given library. LibComp will also leave a commented link to the maven repository page which was queried for the user to be able to refer to.

Finally, LibComp again uses our server's REST API to communicate the user's interaction with the plugin, if the user provides the necessary permissions shown in Figure 3b. If such permission is granted, then whenever LibComp is triggered, a feedback record containing the following information will be sent to our server:

- **Time of Comparison:** current date and time at which the comparison was made.
- **Project ID:** the name of the current project. By keeping track of the project ID, we are then able to track how many times the developer used LibComp within a single project.

(a) LibComp's user profile dialog allows users to create a profile that will be associated with their feedback entries.

(b) LibComp's send feedback dialog explains the collected feedback/data. Checking these boxes means that the user has granted LibComp permission to send the plugin interaction data to our server.

Figure 3: LibComp's feedback system dialogs.

- **Previous Import List:** other import statements in the class in which the selected import statement was clicked. By keeping track of the previous import list, we can identify relationships between groups of libraries with common replacements.
- **Location of Replacement:** the line number where the replacement occurred.
- **Class Name:** the name of the class which the replace happened in. By keeping track of the class name, we can track how many comparisons the developer completes within a single class.
- **Selected Library:** the initial library which was selected for comparison. By keeping track of the selected library, we are then able to track what libraries developers are exploring most often.
- **Alternative Libraries:** the possible alternative libraries which belong to the same domain as the selected library. This is recorded in case future libraries are added to a domain.
- **Selected Alternative:** the library which the user chose to replace the selected library with. This value is null if the user decides against replacing their initial selected library and clicks cancel. By keeping track of the selected alternative, we are then able to track what library replacements are common.

Tracking how developers interact with LibComp allows us to find patterns of behaviour on frequent replacements. All feedback entries are associated with a user profile that each user creates, displayed in Figure 3a. For each user, we collect information such as their occupation, team and project sizes and level of programming experience. We do not collect any identifying information such as name or email, but instead generate a random user ID which associates any collected feedback with the profile information. We also allow the user to optionally rate LibComp and send any comments they may have regarding LibComp. This feedback can help us make improvements to the plugin in the future.

6 FUTURE WORK

This paper described LibComp's first release. We encourage users to report any bugs or desired feature requests on our GitHub repository. We also plan for the following future enhancements.

Functionality. When working at the dependency level in gradle/maven files, LibComp can precisely replace a library dependency with its alternative. However, when working at the import level, LibComp can recognize only the base package of a library. Thus, from LibComp's perspective, importing `org.junit.X` is no different than importing `org.junit.Y` and both these import statements would be replaced by `org.testng.*`, if the user chooses to do so. From a tooling perspective, we plan to explore better granularity levels for detecting and replacing import statements. From a research perspective, we will investigate code migration techniques to not only replace the import for the developer but also update any existing code snippets that use the old library.

Usability evaluation. We originally planned an evaluation of LibComp as follows: (1) use before/after perceived competence scales [9] to understand user's perceived competence to compare libraries, (2) use co-discovery learning [14] during the actual tasks participants need to perform to gather more information about their thought process, and (3) finally use the Quantitative evaluation of satisfaction w/ UI (QUIS) scale [4] to evaluate UI usability. Due to COVID-19 in-person meeting restrictions, we had to put this evaluation on hold, but we plan to investigate online alternatives.

CONCLUSION

This paper presented LibComp, an open-source IntelliJ plugin [12] for comparing Java libraries. LibComp provides a metric-based comparison of libraries in the same domain to help developers decide about the best library to use. LibComp currently supports library comparisons within gradle and maven build files by analyzing declared dependencies and within Java files by analyzing import statements. If authorized by users, it also collects data about how developers interacted with the comparisons and sends us this data for later analysis. Using LibComp, developers are not only able to compare and switch between libraries within the same domain, they can do so without having to leave the comfort of their IDE.

REFERENCES

[1] [n. d.]. Maven Repository. https://mvnrepository.com/

[2] Sven Amann, Sebastian Proksch, and Sarah Nadi. 2016. FeedBaG: an interaction tracker for visual studio. In *24th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 1–3.

[3] Brock Angus Campbell and Christoph Treude. 2017. NLP2Code: Code snippet content assist via natural language tasks. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 628–632.

[4] John P. Chin, Virginia A. Diehl, and Kent L. Norman. 1988. Development of an Instrument Measuring User Satisfaction of the Human-Computer Interface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '88)*. Association for Computing Machinery, New York, NY, USA, 213–218. https://doi.org/10.1145/57167.57203

[5] Fernando López de la Mora, Rehab El-Hajj, and Sarah Nadi. [n. d.]. Metric-based Library Comparison Scripts. https://github.com/ualberta-smr/LibraryMetricScripts

[6] Fernando López de la Mora, Rehab El-Hajj, and Sarah Nadi. [n. d.]. Metric-based Library Comparison Website. http://smr.cs.ualberta.ca/comparelibraries/

[7] Fernando López de la Mora and Sarah Nadi. 2018. An Empirical Study of Metric-based Comparisons of Software Libraries. In *PROMISE*.

[8] Fernando López de la Mora and Sarah Nadi. 2018. Which Library Should I Use?: A Metric-based Comparison of Software Libraries. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '18)*. ACM, New York, NY, USA, 37–40. https://doi.org/10.1145/3183399.3183418

[9] Edward L. Deci and Richard M. Ryan. 2002. *Handbook of self-determination research*. University of Rochester Press.

[10] IntelliJ Platform SDK DevGuide. Last accessed: 2019. Program Structure Interface (PSI), https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html.

[11] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2015. Boa: Ultra-large-scale software repository and source-code mining. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1 (2015), 1–34.

[12] Rehab El-Hajj and Sarah Nadi. [n. d.]. LibComp plugin. https://github.com/ualberta-smr/LibCompPlugin

[13] ernando López de la Mora, Sarah Nadi, and Rehab El-Hajj. [n. d.]. Library Metric Scripts. https://github.com/ualberta-smr/LibraryMetricScripts

[14] Rachel L. Franz, Barbara Barbosa Neves, Carrie Demmans Epp, Ronald Baecker, and Jacob O. Wobbrock. 2019. *Why and How Think-Alouds with Older Adults Fail: Recommendations from a Study and Expert Interviews*. Springer International Publishing, Cham, 217–235. https://doi.org/10.1007/978-3-030-06076-3_14

[15] L. Hattori and M. Lanza. 2010. Syde: a tool for collaborative software development. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 2. 235–238. https://doi.org/10.1145/1810295.1810339

[16] Andre Hora and Marco Tulio Valente. 2015. apiwave: Keeping track of api popularity and migration. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 321–323.

[17] Enrique Larios-Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. 2020. Selecting third-party libraries: The practitioners' perspective. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[18] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, and M. Lanza. 2019. Pattern-Based Mining of Opinions in Q A Websites. In *41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 548–559.

[19] Fernando Lopez de la Mora. 2018. *Providing Software Library Selection Assistance By Using Metric-Based Comparisons*. Master's thesis. University of Alberta.

[20] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A Comparative Study of Manual and Automated Refactorings. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 552–576.

[21] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. 2013. An empirical study of API usability. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 5–14.

[22] Gias Uddin, Olga Baysal, Latifa Guerrouj, and Foutse Khomh. 2019. Understanding how and why developers seek and analyze api-related opinions. *IEEE Transactions on Software Engineering* (2019).

[23] Gias Uddin and Foutse Khomh. 2017. Mining API aspects in api reviews. In *Technical Report*.

[24] G. Uddin and F. Khomh. 2017. Opiner: An opinion search and summarization engine for APIs. In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 978–983.

[25] L. Xavier, A. Brito, A. Hora, and M. T. Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 138–147. https://doi.org/10.1109/SANER.2017.7884616