Which library should I use?

A metric-based comparison of software libraries

Fernando López de la Mora
University of Alberta, Canada
lopezdel@ualberta.ca

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

ABSTRACT

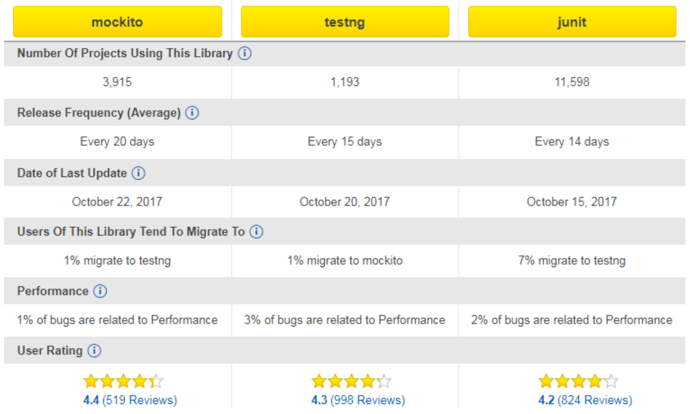
Software libraries ease development tasks by allowing client developers to reuse code written by third parties. To perform a specific task, there is usually a large number of libraries that offer the desired functionality. Unfortunately, selecting the appropriate library to use is not straightforward since developers are often unaware of the advantages and disadvantages of each library, and may also care about different characteristics in different situations. In this paper, we introduce the idea of using software metrics to help developers choose the libraries most suited to their needs. We propose creating library comparisons based on several metrics extracted from multiple sources such as software repositories, issue tracking systems, and Q&A websites. By consolidating all of this information in a single website, we enable developers to make informed decisions by comparing metric data belonging to libraries from several domains. Additionally, we will use this website to survey developers about which metrics are the most valuable to them, helping us answer the broader question of what determines library quality. In this short paper, we describe the metrics we propose in our work and present preliminary results, as well as faced challenges.

1 INTRODUCTION

Software libraries provide ready-to-use functionality by exposing *Application Programming Interfaces* (APIs) to client developers. To accomplish a specific task (e.g. encrypting a file), there is usually a repertoire of libraries available that can achieve the desired objective. However, with so many libraries to choose from and different factors to consider, picking the appropriate one to use is usually not an easy decision. For example, selecting a library that is bug-prone could result in unexpected program behavior. Similarly, a library that lacks community support could mean that the software is no longer maintained and that there is no one available to answer questions about it. Such problems could make a client project completely abandon a library in favor of a similar one [5, 14].

Similar to previous work [15], we use term *aspect* to refer to features or characteristics of software libraries. Additionally, we refer to a software *metric* as a quantifiable measurement that describes an aspect. For instance, release frequency and average time to close a bug report are both metrics related to the software support aspect.

ICSE-NIER'18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5662-6/18/05…$15.00
https://doi.org/10.1145/3183399.3183418



	mockito	testng	junit
Number Of Projects Using This Library ⓘ	3,915	1,193	11,598
Release Frequency (Average) ⓘ	Every 20 days	Every 15 days	Every 14 days
Date of Last Update ⓘ	October 22, 2017	October 20, 2017	October 15, 2017
Users Of This Library Tend To Migrate To ⓘ	1% migrate to testng	1% migrate to mockito	7% migrate to testng
Performance ⓘ	1% of bugs are related to Performance	3% of bugs are related to Performance	2% of bugs are related to Performance
User Rating ⓘ	★★★★☆ 4.4 (519 Reviews)	★★★★☆ 4.3 (998 Reviews)	★★★★☆ 4.2 (824 Reviews)

Figure 1: Comparison of testing libraries

Uddin et al. [15] showed that developers care about API aspects such as performance, usability, security, documentation, compatibility, community, and bugs. Such aspects could help determine whether or not a specific library is a good option for a given task when compared to similar libraries. While it is possible to find some questions and answers about API aspects on websites such as Stack Overflow, this information usually focuses on a limited number of aspects and does not provide the readers with the full picture of the library being discussed. Additionally, most of the information is text-based in the form of users' opinions and users need to sift through many posts to extract useful information. Similar to how customers can compare metrics of different products that belong to the same category when shopping online (e.g., comparing monitors in terms of resolution and screen size), we argue that presenting metrics that help provide a quantifiable comparison of API aspects could help developers make an informed decision about which library in a given domain is most suited to their project needs. The challenge is determining which software metrics make sense in the context of software libraries, and which of these metrics can be used to assess library quality.

Previous research has focused on measuring different attributes of software systems [7, 10, 12, 14, 16]. These include, for example, number of changes/deletions, release frequency, fault-proneness, and time to close issues. In this paper, we argue that many of these attributes can be used as quantifiable metrics for comparing software libraries. We propose collecting relevant metrics from diverse data sources, such as version-control and issue-tracking systems, and consolidating them in a single website that developers can use to compare libraries from the same domain (e.g., cryptography or testing). Figure 1 shows a mockup of this website.

Our goal with creating this library metric comparison website is two-fold: (1) we aim to provide developers with a single source of information that will assist them in selecting the appropriate libraries for their projects, and (2) we will use the website as a continuous surveying and crowd-sourced mechanism to find out which metrics influence developers' decisions when choosing libraries, and also which metrics can be used to assess library quality.

2 MOTIVATION

In this section, we present a real motivational example and define our problem statement and goals. We also discuss previous work that had similar goals and how they differ from ours.

Motivational Example. Q&A websites, such as Stackoverflow, show that developers often care about certain library characteristics, but may not be aware of others. For example, question #11707976 on Stackoverflow is asked by user *user568021*, who needs a Java cryptography library to encrypt files for her application. She mentions that the library *cryptix* has all the functionality that she needs. However, she adds that development of the library was halted since 2005. Due to this reason, she asks about alternative libraries. Among the received answers, user *fvu* also recommends against using *cryptix* as she thinks that the software is abandoned: *"I would seriously think twice before going this route. The development of the software was halted because standard alternatives exist, and have a look at the mailing list, there's been no significant activity since 2009. In my book that means that the software is abandoned, and abandoned software means you're more or less on your own."* In the same answer, user *fvu* suggests using the library *jasypt*, to which the original poster comments that she has already tried such suggestion, but has found a bug which deterred her from continuing to use that library: *"Yeah I tried jasypt but it has some bug in the binary decryption, I really don't want to deal with them right now."*

Problem Statement. As shown in the motivational example, library aspects, such as lack of community support and defective functionality, can be decisive factors when it comes to choosing a software library. The example discussed two metrics for measuring the community support aspect, last development date and last activity date. There may be additional library aspects that are important for client developers, as well as various metrics that can be used to infer information about these aspects. We formulate the following problem statement: **Library metrics are important to developers when choosing libraries. Which library metrics matter most to developers? Which metrics can we associate to library quality? How can we extract such metrics?**

Goals. Our goal is to help developers choose software libraries by providing them with a single place where they can compare library metrics. We believe that *user568021* would have been able to more easily choose a library if she had a resource similar to that shown in Figure 1. In the process of creating this resource, we will determine the metrics that can be used to assess library quality.

Related Work. The work most related to ours is that by Uddin et al. [15], who created a website that provides API summaries based on aspects such as documentation and performance, mined from Stack Overflow discussions. As opposed to focusing on a single source of text-based information and focusing on opinions, our work proposes extracting quantifiable metrics from different sources such as software repositories, issue tracking systems, *and* Q&A websites. Using a combination of various information sources allows us to create a larger set of metrics that span different perspectives.

Hora et al. [3] and [8, 9] used popularity and migration data, to rank APIs and provide recommendations to developers respectively. Unlike the work by these authors, which compares libraries through a limited set of metrics, our work consists of using several metrics associated with different aspects of a library.

There has also been related research that focused on extracting metrics from software projects with purposes other than providing help in library selection [7, 10, 12, 14, 16]. We mention such related work in Section 3.

3 METRICS

We now present software metrics we believe are relevant for library client developers. Several of these metrics were extracted from the literature, even if originally proposed for different purposes.

Popularity. Library *popularity* represents the number of its client projects. The more projects using a library, the more popular it is. Related research has focused on mining software repositories to obtain popularity information of API elements, such as classes [3, 9]. In our work, we are interested in obtaining popularity at the granularity level of the whole library; that is, the number of client projects using any of the library's APIs. We believe this is a relevant metric since it informs interested users about which libraries the majority of developers are using in their projects.

Release Frequency. We refer to *Release Frequency* as the average time difference between two consecutive releases of the same library. In related work, Khomh et al. [6] investigated the relation between shorter release cycles of software and its respective quality defined by metrics such as crash rates and post-release bugs. To calculate release cycles, the authors extracted the starting date of the development of Firefox versions and its release date from Mozilla release notes, and computed the difference between these two dates. We are interested in how often a library has new releases, since client developers may want to know if a library receives constant or infrequent updates before committing to use it; such updates usually contain new features or bug fixes.

Issue Response And Closing Times. *Issue Response Time* indicates the average time that it takes to get a response back when a bug report is opened. On the other hand, *Issue Closing Time* refers to the average time between the closing date and the creation date of an issue report. Work by Ortu et al. [12] analyzed the relation between sentiment, emotions, and politeness of Apache developers in their JIRA comments with the time that it takes to fix an issue. Giger et al. [2] used attributes such as severity, priority, and assignee from bug reports of open-source projects to create prediction models for bug fixing times. For our work, we care only about the averages of these two metrics. We believe that these metrics provide information about developer and community support in terms of estimations for both the waiting time that library users might expect in order to receive any response about a recently opened bug report, and for the resolution time of an issue.

Recency. This metric refers to the last time a library was updated, either with new code added to its repository or with a new version release. Based on our motivational example, we believe that recency could be useful to determine if the development or maintenance of a library remains active.

Backwards Compatibility. A library that is not *backwards compatible* with its previous versions results in compilation errors or modified program behavior compared to code that was written using past versions of the library. This represents problems to the client developers as they have to modify code that used to previously work before updating the library. Xavier et al. [16] extracted breaking changes from APIs through a diff tool that collected such changes between two versions of a Java library. Mostafa et al. [10] detected backwards compatibility problems in Java libraries by performing regression tests on version pairs, and by inspecting bug reports related to version upgrades.

Similar to the work by Xavier et al. [16], we plan to show the average number of breaking changes per release for a library as a way to measure backwards compatibility. We believe that this metric is relevant as developers might want to avoid libraries that often have a large number of breaking changes with each release.

Migration. Library migration occurs when a client project replaces a library in favor of a different library in order to accomplish a similar task. Teyton et al. [14] detected library migrations in over 8,000 Java open-source software projects. Their approach consists of analyzing the changes (additions and removals) in the library dependencies of different versions of a project using static analysis. Kabinna et al. [5] studied the logging of library migrations by manually analyzing JIRA issue reports containing keywords related to migrations and the respective Git commit history of such reports.

We believe that developers may want to be aware of common library migrations before they commit to a particular library and invest time in using it. Accordingly, we are mainly interested in pairwise migrations of libraries in the same domain (e.g. projects using *testng* often migrate to *junit*). To determine this information, we will use the migration dataset provided to us by Teyton et al.[14]. We will follow their same methodology to extract information about libraries not already included in the dataset.

Fault-proneness. Fault-proneness refers to how likely a library might result in unexpected behavior caused by software defects. Linares-Vazquez et al. [7] measured fault-proneness by calculating the number of bug fixes of a library. They accomplished this by looking at commit messages related to these fixes.

We also plan to count the number of bug fixes of a library as a way to measure fault-proneness. This metric may provide an estimation on how buggy a library is compared to similar ones. To enable comparison of libraries, we will present a normalized measure of these bug fixes.

Performance & Security. Non-functional properties such as performance and security are important factors for developers looking to incorporate libraries in their projects. Performance of a software library refers to the efficiency of its underlying code. Libraries with performance issues may result in unexpectedly slow execution, inefficient behavior, or crashes caused by memory leaks. On the other hand, libraries with security problems have code vulnerabilities that users might exploit for malicious purposes. Research by Uddin et al. [15] has measured performance and security of APIs by mining opinions from discussion forums.

As a less subjective source of information, we plan to use bug reports to reveal any performance or security problems that a library may have. We think that developers want to be aware of libraries with a high number of security vulnerabilities or performance issues. In the future, we will consider extracting information about additional non-functional properties from bug reports.

4 PRELIMINARY RESULTS & CHALLENGES

In this section, we report on some interesting preliminary results from two of the metrics we explored, as well as the challenges we have faced. For our preliminary results, we are using a set of 60 Java libraries taken from a variety of domains such as cryptography and testing. As selection criteria for these libraries, we require that they are open source and have available issue tracking systems either on Github or JIRA to allow automating the collection of certain metrics. For our final list of subject systems, we plan to cover a variety of domains, with several libraries per domain.

Popularity. We obtain popularity information using BOA [1], which allows us to mine a large dataset of over 9 million Github projects. We assume that a project uses a library in our dataset if any of its Java files contain an *import* statement to any of the library APIs. As BOA's latest dataset dates to September 2015, we plan to either use an updated dataset if released or implement the same search criteria ourselves to be able to query the latest data from all libraries. From our results, *bouncycastle* has the largest number of client projects in the cryptography domain, with *apache shiro* and *spongycastle* claiming the second and third ranks respectively. *JUnit* is the most popular library in the testing domain, followed by *mockito* and *testng*.

Migration. So far, we have used the migration dataset from the work by Teyton et al. [14] to find migration data for the libraries on our list. Interestingly, while our popularity results above show that *junit* is the most popular testing library, the migration dataset shows that the third most common library migration consists of projects replacing *junit* with *testng*. This suggests that while some developers might initially think of picking *junit* as their library of choice for testing purposes, they may hesitate after reading the migration data. This particular example shows how the combination of different metrics can shape the final decision of a developer.

Performance and Security. Since we want to use the number of reported security- and performance-related bugs as an objective metric for the security and performance of a library, we need to find a way to uniformly extract this data from the various library repositories. This is challenging, because libraries use different issue tracking systems. Furthermore, issue tracking systems such as Github may optionally contain customized information such as labels or tags for bug reports. These challenges led us to resort to using more heuristic-based approaches, published in the literature, to automatically identify performance and security bugs.

As learning algorithms have been successfully applied for text classification purposes (e.g.,[4]), we initially wrote machine learning classifiers trained with a dataset of manually labeled performance and security bug reports created by Ohira et al. [11]. We created a supervised classifier for performance classifications trained with a collection of 636 bug reports, half of them labeled as performance bug reports. Similarly, we created a second classifier for security classifications trained with 318 bug reports, half of them labeled as security reports. For each bug report, we concatenated its summary and description, eliminated stop words, and stemmed the remaining words from the resulting text. We calculated the inverse term frequency of the resulting text and used it as input to a classifier. We are currently tuning our classifiers to obtain acceptable recall and precision numbers. Recently, Pandey et al. [13] classified bug reports using machine learning algorithms. However, their objective consisted of classifying bug reports into "true bugs" or not, which is a more general classification goal than ours. In the context of performance and security classifications we aim to do, we believe that bug reports are difficult to classify as their descriptions may contain explanations with multiple terms that are not necessarily related to performance or security issues. Furthermore, there are words whose interpretation depends on the domain. For example, *block* is usually used to describe performance issues (e.g. blocking resources), but in cryptography libraries, this word is a noun commonly found when referring to ciphers.

As an alternative to using classifiers, we are also experimenting with a keyword-search approach. Keywords are often used as filtering criteria in bug reports or messages in version control systems to detect information of interest [7, 10]. By analyzing the summaries and descriptions of the performance and security bug reports found in the Ohira et al.'s dataset [11], we have created a keyword list that we believe can describe performance or security problems independent of the library domain. While this is still work in progress, we have noticed fewer bug reports incorrectly labeled as performance or security issues using this approach.

5 CONCLUSIONS AND FUTURE WORK

Summary. With the abundance of software libraries offering similar functionality, it is often not clear to developers which library best suits their needs. In this work, we introduced the idea of using metrics extracted from various information sources, such as revision history and bug reports, as a means of providing decision support to developers through useful comparisons. We believe that having one website that developers can go to and see a clear metric-based comparison of libraries, such as that shown in Figure 1 can help them choose the right library. We have already discussed several metrics we derived from the literature, often ones not used in the context of comparing libraries. We provided some preliminary results, and described the challenges faced due to comparing libraries coming from different domains and being developed under varying conventions and software processes.

Next steps. Our main next step is to collect the remaining metrics, while finding the right assumptions and heuristics to provide a fair comparison of varying libraries from different domains. Once this is done, we will launch our website that summarizes all this information in a way that is easy for developers to navigate through. We will also use the website as a means to survey developers on which metrics they find useful and why, as well as ask them to rate the presented libraries so we can collect the overall rating information shown in Figure 1. Our idea is to have a single website that developers can go to for information and at the same time continuously provide feedback about the usefulness of the metrics. The collected information will serve as a crowd-sourced answer to the question of which library metrics relate to library quality and affect developers' choices, and can be used to dynamically update the presented information on the website.

REFERENCES

[1] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A language and infrastructure for snalyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 422–431.

[2] Emanuel Giger, Martin Pinzger, and Harald Gall. 2010. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering (RSSE '10)*. ACM, New York, NY, USA, 52–56.

[3] Andre Hora and Marco Tulio Valente. 2015. Apiwave: keeping track of API popularity and migration. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME '15)*. IEEE Computer Society, Washington, DC, USA, 321–323.

[4] Andreas Hotho, Andreas Nürnberger, and Gerhard Paaß. 2005. A brief survey of text mining. *LDV Forum - GLDV Journal for Computational Linguistics and Language Technology* 20, 1 (May 2005), 19–62.

[5] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. Logging library migrations: a case study for the Apache software foundation projects. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 154–164.

[6] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. 2012. Do faster releases improve software quality?: an empirical case study of Mozilla Firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR '12)*. IEEE Press, Piscataway, NJ, USA, 179–188.

[7] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API change and fault proneness: a threat to the success of Android apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 477–487.

[8] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. 2009. Mining trends of library usage. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops (IWPSE-Evol '09)*. ACM, New York, NY, USA, 57–62.

[9] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. 2010. Mining API popularity. In *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques (TAIC PART'10)*. Springer-Verlag, Berlin, Heidelberg, 173–180.

[10] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. Experience paper: a study on behavioral backward incompatibilities of Java software libraries. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 215–225.

[11] Masao Ohira, Yutaro Kashiwa, Yosuke Yamatani, Hayato Yoshiyuki, Yoshiya Maeda, Nachai Limsettho, Keisuke Fujino, Hideaki Hata, Akinori Ihara, and Kenichi Matsumoto. 2015. A dataset of high impact bugs: manually-classified issue reports. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 518–521.

[12] Marco Ortu, Bram Adams, Giuseppe Destefanis, Parastou Tourani, Michele Marchesi, and Roberto Tonelli. 2015. Are bullies more productive?: empirical study of affectiveness vs. issue fixing time. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 303–313.

[13] Nitish Pandey, Abir Hudait, Debarshi Kumar Sanyal, and Amitava Sen. 2018. *Automated classification of issue reports from a software issue tracker.* Springer Singapore, Singapore, 423–430.

[14] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. 2014. A study of library migrations in Java. *J. Softw. Evol. Process* 26, 11 (Nov. 2014), 1030–1052.

[15] Gias Uddin and Foutse Khomh. 2017. Automatic summarization of API reviews. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*.

[16] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: a large scale study. In *24th International Conference on Software Analysis, Evolution and Reengineering (SANER '17)*. 138–147.