Providing Software Library Selection Assistance By Using Metric-Based Comparisons

by

Fernando Lopez de la Mora

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Software libraries provide a set of reusable functionality, which helps developers write code in a systematic and timely manner. However, with the plethora of similar software libraries available in the market, selecting the appropriate one to use is often not a trivial task. Choosing libraries that are not well-suited for a project may lead to development delays or high maintenance. In this work, we investigate the idea of helping developers select libraries by providing them with a metric-based comparison of libraries in a given domain. Different developers care about different aspects of a library and two developers looking for a library in a given domain may not necessarily choose the same library. Thus, instead of directly recommending them a library to use, we empower developers with the information they need to make an informed decision. To evaluate this idea in practice, we presented software developers with a survey containing an initial implementation of a comparison of libraries, while obtaining feedback about our proposed approach and understanding the metrics developers care about. Our results show that developers find that the proposed technique provides useful information when selecting libraries. Based on the collected feedback from developers, we made enhancements to our metrics and comparisons and implemented a website whose objective is to serve as a source of library information, as well as a continuous surveying and crowd-sourcing mechanism to uncover metrics that influence developers' decisions when choosing software libraries.

Preface

The survey contained in this thesis received ethics approval from the University of Alberta Research Ethics Board, Project Name "Task-based Code Recommender Systems", No. Pro00074107, August 24th, 2017.

A version of Chapter 2 and Chapter 3 of this thesis was published as "Which library should I use? A metric-based comparison of software libraries" [31] at the International Conference on Software Engineering New Ideas and Emerging Results Track (ICSE NIER) 2018. Dr. Sarah Nadi was the supervisory author involved with concept formation and manuscript composition.

A version of Chapter 4 and Chapter 5 of this thesis has been accepted as "An empirical study of metric-based comparisons of software libraries" for review at the International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE) 2018. Dr. Sarah Nadi was the supervisory author involved with concept formation and manuscript composition.

To my parents

For their love and support.

Acknowledgements

I would like to thank my supervisor Dr. Sarah Nadi for her great and helpful supervision. Without her quality guidance and endless patience this thesis would not had been possible.

I am also grateful to Dr. Carrie Demmans Epp and her CMPUT 302 students, for showing us how to visualize our work in an intuitive and user-friendly manner.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software libraries facilitate development tasks by allowing client developers to reuse existing code that provides ready-to-use functionality, such as encrypting files or connecting to a database. Libraries expose their functionality through *Application Programming Interfaces* (APIs), which developers can make use of to access the promised operations of a library.

Nowadays, it is possible to find several libraries that were created to accomplish similar duties. For example, *junit* and *testng* are two libraries that enable users to create code tests. With the rising popularity of social coding websites such as Github, the availability of software libraries has expanded in recent years. However, with this proliferation of software, it is increasingly becoming a daunting task for developers to search and select a library that fits their needs. Furthermore, choosing an unsuitable library for a software project could result in the need to replace the library with a better suited one, for reasons such as performance gains or additional features [18]. These replacements translate into maintenance tasks which can consume up to 80% of software costs [9], making the selection of a suitable library for the job a duty that should not be neglected.

Developers often use Question and Answer (Q&A) websites to ask for library suggestions or compare similar libraries. For instance, question #11707976 [1] on Stack Overflow shows a developer looking for a cryptography library in Java. This developer has been considering using the *cryptix* library, but men-

[1] https://stackoverflow.com/questions/11707976/cryptography-in-java

tions that it was last updated in 2005, so she is looking for recommendations. One of the answers to this question recommends against using an old library: *"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."* Additionally, this user suggests the *jasypt* library as an option, to which the original poster responds: "*Yeah I tried jasypt but it has some bug in the binary decryption, I really don't want to deal with them right now.*"

The previous discussion shows that libraries that suffer from antiquity and fault-proneness can deter users from using the software. Furthermore, past research has shown that developers care about other aspects such as performance and community [49]. While it is possible to find information about these aspects in discussion forums, the data is unstructured, scattered across different threads or posts, in textual form, and is based on users' opinions or personal experiences, which hinders the process of comparing similar libraries. Analogous to how customers can compare specifications of hardware products such as computer monitors when shopping online, we propose that presenting quantifiable and comparable data of similar libraries, as shown in Figure 1.1, could allow developers to make better informed decisions with their library selections.

Previous research has focused on measuring different attributes of software systems [23], [32], [34], [47], [52]. These include, for example, number of changes/deletions, release frequency, fault-proneness, and time to close issues. 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).

In this thesis, we present a methodology to compare libraries using *metrics*, that is, quantifiable data that describes characteristics of the software.

mockito	easymock	powermock
Popularity		
5,380	1,998	716
Release Frequency		
Every 9.87 days	Every 238.28 days	Every 63.59 days
Date Of Last Code Update		
2018-01-16	2018-01-02	2017-11-03
Performance		
2.14% of all bug reports are related to Performance	3.28% of all bug reports are related to Performance	2.23% of all bug reports are related to Performance
Security		
0% of all bug reports are related to Security	0% of all bug reports are related to Security	0.13% of all bug reports are related to Security
Issue Response Time		
18.56 days	41.45 days	11.29 days

Figure 1.1: A metric-based comparison of mocking libraries

By using data obtained from different sources, namely open source software repositories, issue tracking systems, version control systems, and Q&A websites, we are able to automate the extraction process of these metrics and present recent information to users who are looking for libraries in a specific domain. Additionally, we conduct a developer survey with a preliminary version of our metric-based comparison of libraries to evaluate the usefulness of our approach, to find the metrics that matter the most to developers, and obtain valuable feedback, which we considered to make further enhancements to our technique. Finally, we show a complete implementation of our proposed methodology in a website where the metric data is automatically updated and users can compare libraries from different domains, as well as submit feedback for the usefulness of metrics.

1.1 Contributions

The main contributions of this thesis are:

- A methodology to compare software libraries using extractable metrics obtained from different sources such as open source repositories, issue tracking systems, version control systems and Q&A websites.
- An empirical study consisting of a survey of 61 software developers as a

means of determining the general usefulness of our proposed approach, the most important metrics for developers when choosing libraries, and the importance of metrics in different domains.

- A concrete implementation of our proposed technique consisting of a website where users can compare libraries from a variety of different domains, which also serves as a continuous surveying and crowd-sourced mechanism to find out the metrics that influence developers' decisions when choosing libraries.

- A discussion of possible improvements that can be made to the proposed comparison methodology.

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 discusses related research. Chapter 3 presents the metrics that we use in our comparison of libraries, along with the extraction methodologies that we use. In Chapter 4, we show the developer survey that was conducted as a way to obtain feedback to our approach. Chapter 5 details the results that we gathered from the survey, as well as a discussion based on the feedback provided by participants. In Chapter 6, we present our final website that contains a variety of libraries taken from different domains and their respective metric-based comparisons. We explain the threats of validity to our work in Chapter 7. Finally, Chapter 8 contains the conclusions and future directions for this research.

Chapter 2

Related Work

Researchers have extracted software data from sources such as open source software repositories, version control systems, Q&A websites, and issue tracking systems for a variety of purposes, such as recommending APIs or finding correlations between successful mobile applications and changes in APIs. In this chapter, we group related literature in two categories. Section 2.1 mentions work in which software data has been extracted with the objective of assisting developers in selecting APIs or libraries. On the other hand, Section 2.2 cites work that has used software metrics or data for reasons not necessarily related to the recommendation of libraries. Finally, we include information about relevant mining tools in Section 2.1, while Section 2.4 mentions available websites that show information about APIs or libraries.

2.1 Library or API Selection Assistance

Mining data from software repositories and Q&A websites has been used as a method to obtain information to rank or recommend APIs. For instance, Uddin et al. [49] measured and rated aspects of APIs such as documentation and performance by mining opinions (e.g. sentences that have a positive or negative sentiment) from Stack Overflow posts and counting the number of positive sentences using sentiment analysis. To accomplish this, the authors grouped opinions based on the aspects being discussed in sentences and rated aspects based on the overall sentiment of the relevant opinions.

Rahman et al. [40] mined questions and answers from Stack Overflow

to associate keywords to relevant APIs with the objective of recommending APIs for a given task described in natural language. Thung et al. [48] used association rule mining and collaborative filtering techniques to, given a set of libraries that a project is currently using, recommend libraries based on what other projects have used. Teyton et al. [46] proposed a mining technique to identify appropriate alternatives to replace existing libraries in a project. The technique is based on leveraging library dependency information in project management tools such as Maven to find library migrations in the revision history of projects.

Hora et al. [14] ranked API elements based on popularity and migration data mined from open-source repositories. The authors obtained popularity metrics by counting the additions and removals of Java import statements found in the diff of source code files. Similarly, they used the diff of source code files to obtain migration metrics by analyzing import statements that were replaced with other import statements, and thus, determining that a project migrated from an API to another one.

Mileva et al. [27] mined popularity information about API elements (classes and interfaces) from Java open source software repositories. The authors measured popularity trends by mining the history of repositories and performing a monthly count of the number of projects using Java import statements on a given class or interface in the source code over a period of time. In earlier work, Mileva et al. [26] created a tool that assists developers in selecting the most stable version of a library based on usage trends of library versions, including the number of projects using a given library version.

Similar to the related work in this section, we take advantage of open source software repositories and Q&A websites to extract relevant data for library selection assistance. However, unlike the research cited above, we do not rely on opinions from Q&A websites, and instead of using one source of information we combine and quantify data from several sources to help developers make an informed decision when selecting a library.

2.2 Software Metrics

There has been related work that has extracted metrics in isolation with purposes not necessarily related to recommending libraries or APIs. We group these metrics in the following categories: Popularity, Migrations, Change and Fault-Proneness, Bug Reports, Reliability, Opinions, Sentiment Analysis, Release Cycles, Bug-fixing Times, and Backwards Compatibility, Most Used Functionalities of APIs, and Development Activities.

2.2.1 Popularity

Popularity of APIs, specific library versions, and complete libraries, has been largely researched using different methodologies. Most recently, Borges et al. [3] performed a study on the popularity of Github projects using the number of stars on Github repositories as a proxy for popularity. Among their results, they observed an increase in the number of stars gained in a repository after new releases and found that the number of stars in repositories are correlated with their number of forks, contributors and commits.

Mileva et al. [27] collected popularity information about API elements (classes and interfaces) from Java open source software projects. Their notion of popularity of an API element is the number of projects using it. The authors measured the popularity trends of these API elements by mining the history of repositories and performing a monthly count of the number of projects that were using Java import statements on a given class or interface in the source code over a period of time.

Previously, Mileva et al. [26] investigated usage trends of library versions, including popularity, that is, the number of projects using a given library version. They accomplished this by mining the history of 250 open source projects managed by Maven and performing a monthly analysis of their meta information files in order to extract library version usage information over a period of two years.

Sawant et al. [43] found popular APIs by parsing the library dependencies in POM files from Java Github projects that use Maven.

2.2.2 Migrations

Library migrations have been studied in software projects to identify the libraries being replaced and their respective replacements, as well as the drivers behind these migrations. Teyton et al. [47] studied library migrations in Java open source software by performing experiments on the source code of over 8,000 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 of the source code, eliminating the need for Maven information.

Kabinna et al. [18] studied logging library migrations and their motivations within Apache Software Foundation Java projects. They manually analyzed JIRA issue reports containing keywords related to migrations and the respective Git commit history of such reports. Among their findings, the authors noticed that flexibility and performance improvement are the main drivers for migrating to another library.

2.2.3 Change and Fault-proneness

Linares-Vazquez et al. [23] studied the relation of the success of mobile applications, determined by its user ratings, with the change and fault-proneness of the Android APIs used by such apps. To do this, they computed the number of bug fixes and changes to the interfaces, implementation, and exception handling of the analyzed APIs used by 7,097 apps. Their findings show that successful apps use APIs that are significantly less fault-prone, and less change-prone in terms of method signatures and public methods, than the APIs used by unsuccessful apps.

2.2.4 Classification and Severity of Bug Reports

Ohira et al. [33] built a dataset of four thousand manually-classified issue reports taken from four open source projects and assigned labels to bug reports such as performance and security. This dataset could be used to analyze concerns of performance and security in open source projects. Along the same

lines, Lamkanfi et al. [21] created machine learning classifiers to categorize bug reports into severe or non-severe classes.

2.2.5 Reliability

The reliability of specific library versions or releases has also been a topic of interest in the literature. Mileva et al. [26] investigated the reliability of library versions by mining the history of 250 Maven-managed software projects and counting the number of times they switched back to a previous library version. The notion of reliability by the authors is that the more a library version is switched back to a previous version by developers, the less reliable the library version is.

Raemaekers et al. [51] introduced four metrics that calculate the stability of the public interface and implementation of a library by analyzing its version history and considering the number of removed methods, the amount of change in existing methods, the ratio of change in new to old methods, and the percentage of new methods. They provide scenarios where these metrics could be used to determine the stability of a library. However, they do not provide an evaluation of the usefulness of these proposed metrics.

2.2.6 Opinions

Uddin et al. [28] presented techniques to automatically detect opinionated sentences of API aspects that developers care about when reading forum posts. In order to do this, they created a benchmark of sentences taken from Stack Overflow posts and labeled each sentence manually with a specific aspect. To detect the polarity (negative, positive or neutral sentiment) of sentences, they used an implementation of the Sentiment Orientation algorithm. To automatically detect opinionated sentences they used machine learning classifiers that provided a precision of up to 80% percent.

2.2.7 Sentiment Analysis

Sentiment Analysis has been applied on text found in commit messages and discussion forums for various different goals. Sinha et al. [44] applied sentiment analysis on developer commit logs from Github projects. To accomplish this, they used the Boa framework to extract over 2 million commit messages from 28K projects and the SentiStrength tool to determine sentiment polarity of such messages. They noticed an overwhelming neutral sentiment across the commit messages. Additionally, they found strong correlations between the negative, neutral, and positive sentiments and the average number of modified files per commit.

Guzman et al. [13] applied sentiment analysis using SentiStrength on 60K commit comments from different Github projects. Among their results, they found that commits made in Mondays tend to have more negative emotions, Java projects tend to have more negative comments, as well as projects with distributed teams tend to have positive content in their commits.

Ortu et al. [35] created a dataset of JIRA issue comments extracted from four popular open source communities and manually labeled such comments with different emotions such as love, joy, surprise, anger, sadness, and fear. They claim that such dataset can be used to investigate the role of emotions in software engineering.

Pletea et al. [39] mined expressions of emotions within security-related discussions on Github using sentiment analysis with the Natural Language Text Processing toolkit (NLTK). To accomplish this, they searched for keywords related to security in the discussions of commits and pull requests from software projects presented in the MSR 2014 Mining Challenge Dataset and used NLTK to classify each discussion with a negative, positive or neutral sentiment. They evaluated the sentiment scores by manually reviewing a subset of the results, however, they disagreed on some of the sentiments produced by the tool.

2.2.8 Release Cycles

Release cycles have been studied for purposes such as comparing shorter release cycles with longer ones. Khomh et al. [19] investigated the relation between shorter release cycles of software and its respective quality defined by metrics such as crash rates and post-release bugs. Their subject system was Mozilla Firefox and found that while bugs take less time to fix with a shorter life cycle, the difference of post-release bugs is negligible when compared to the traditional life cycle.

Costa et al. [5] analyzed bug reports of Firefox releases with traditional cycles as well as rapid releases of the same software, with the intent of studying if rapid release cycles deliver addressed issues at a faster pace. Among their results, they found that there are not major differences in these two types of releases in terms of the time between the issue report date and the integration of the fixed issues into a release.

2.2.9 Bug-Fixing Times

Previous research has tried to predict bug-fixing times, as well as finding factors that affect the fixing time of bug reports. For instance, Giger et al. [7] used attributes such as severity, priority, and assignee from bug reports of open-source projects to create prediction models for bug fixing times. Panjer et al. [37] also experimented predicting bug fix-times with data mining techniques and found that factors such as commenting activity, severity, product, component, and version affect the length of bug lifetimes.

Ortu et al. [34] analyzed the relation between sentiment, emotions and politeness of developers in JIRA comments taken from Apache projects with the time that it takes to fix an issue. They used available tools to measure sentiment and politeness and created their own classifier for emotion measurement. They found that emotions such as joy and love are linked with shorter fixing times, whereas negative emotions are more related with longer times.

Lamkanfi [20] found that removing outliers in bug fix-time datasets can improve the prediction results for the amount of time that it takes to fix bugs.

2.2.10 Backwards Compatibility

Code compatibility between releases of libraries have been studied in the literature. Mostafa et al. [32] detected backwards compatibility problems in Java libraries by performing regression tests on version pairs, and by inspecting bug reports related to version upgrades.

Xavier et al. [52] measured breaking changes in APIs, that is, changes found in new versions of an API that result in compilation errors or behavioral differences in code that was using a previous version of the same API. To accomplish this, they analyzed 317 Java libraries and 260K projects using them, and implemented a diff tool that extracted breaking changes between two versions of a Java library.

2.2.11 Development Activities

Mauczka et al. [25] produced a dataset of commits classified by whether they address functional and non-functional requirements. To do so, the authors surveyed developers of open source projects and asked them to label their own commits based on classification schemes provided by the authors. Such dataset could be applied to provide an insight on the kinds of activities occurring in an open source project.

2.3 Mining Tools

In this thesis, we refer to *mining* as the process of extracting data from sources such as software repositories, issue tracking systems, or Q&A websites. Mining software repositories is a necessary task in our work, as repositories are one of the sources that provide us with metric data for our library comparisons. In this section, we briefly introduce tools that have been designed specifically for mining software repositories and that we cite in this thesis, namely Boa and GHTorrent.

2.3.1 Boa

Boa is an infrastructure created specifically for mining software repositories [6]. It consists of a web interface, a domain-specific language, and large datasets of software repositories taken from Github and Sourceforge, which ease extraction tasks by writing small programs that can be executed on datasets of different sizes in a timely manner. While Boa facilitates mining tasks, its datasets have not been updated since 2015. We use Boa in the initial implementation of our metric-based comparisons.

2.3.2 GHTorrent

GHTorrent [11] is a dataset that provides an offline mirror of Github repository data. GHTorrent datasets are more frequently updated than those of Boa, with new versions available often on a monthly basis. Although we currently extract our data through the Github API, GHTorrent is one option that we consider in our future work to avoid Github query limits.

2.4 API or Library Comparison Websites

Websites that compare APIs based on metrics are currently available. For example, `npms.io` is a website that allows to search for Javascript node packages and compare them on three metrics: quality, popularity, and maintenance. Unlike our proposed website, npms only provides numeric scores for their metrics without showing users the actual extracted data. `Apiwave.com` presents Java API rankings based on popularity and migration data extracted from Github [14]. Our work differs in terms of the set of metrics we employ and how we compare libraries.

2.5 Chapter Summary

In this chapter, we presented literature related to this thesis. This previous research was viewed from two perspectives: Library or API Selection Assistance and Software Metrics. The first category consists of work that has extracted

data from software repositories and Q&A websites with the objective of providing recommendations or selection assistance to developers. While this thesis shares the same goal, our work is different in that we combine several sources of information from which we extract metric data. Additionally, we provide visualization of the data. The second category focuses focuses on extracting metrics for a variety of intentions. We discuss this category as part of related work as we leverage some of these metrics for library selection assistance purposes. Finally, we mention tools used to mine software repositories and related websites used for API comparison purposes.

Chapter 3

Metrics

As a first step for helping developers make informed decisions about library selections, we identified an initial set of metrics that we use for a first implementation of our methodology. For our comparison of libraries, we require metrics which, as per our definition of this term, have to provide comparable and quantifiable data that describes a characteristic of a library. For example, a metric that counts the number of client projects of a library meets this criteria as this data can be used to compare the popularity levels of similar libraries, that is, how many projects depend on this software. As another requirement, intrinsic to our idea of creating a website that constantly updates information for comparison of libraries, we need metrics whose extraction methodology could be automated. Furthermore, many of our metrics are related to *Non-Functional Requirements* (NFR). In contrast to Functional Requirements which specify the functionality of libraries, Non-Functional Requirements can describe quality aspects of a library. Examples of NFR include reliability, performance, and security of libraries. Since our idea is to compare libraries with alike functionality, that is, similar Functional Requirements, showing metrics related to NFR allows developers to see differences in libraries despite their domain similarities.

Based on the above criteria, we choose the following library metrics: Popularity, Release Frequency, Issue Response Time, Issue Closing Time, Backwards Compatibility, Performance, Security, and Last Discussed on Stack Overflow. In the rest of this chapter, we present the definition, intuition,

and extraction methodology of these metrics. An example of each metric is provided with data taken from the Java library *mockito*.

3.1 Popularity

Definition and Intuition.

Developers may simply want to use what the majority of developers are using. We define *Popularity* as the number of client projects using a given library. Instead of using popularity of API elements such as classes of a library, we focus on the number of projects using any API element of a given library.

Extraction Methodology.

To obtain Popularity data, we use BOA [15], a large-scale mining infrastructure. To count the number of client projects of a given library, we wrote a BOA script that searches the latest snapshot of Java files in a project and looks for Java import statements that include the general package of a library in these files. If this requirement is met, we assume that the project uses the target library. As detailed in Chapter 2, this is a common methodology used to extract popularity information. We ran our script on the latest available full September 2015 dataset on BOA, which consists of a snapshot of more than 7 million Github projects. As an example for this metric, mockito's popularity is 5,380, which represents its number of client projects in this dataset.

3.2 Release Frequency

Definition and Intuition.

Developers may be interested in knowing how often a library is updated, as new releases usually contain bug fixes and added functionality. We define the *Release Frequency* of a library as the average time between two consecutive releases of a library, more formally:

$$ReleaseFrequency = \frac{\sum_{i=1}^{N-1} ReleaseDate_i, ReleaseDate_{i+1}}{NumberOfReleases - 1} \tag{3.1}$$

Extraction Methodology.

We use the Github API to extract the git tag information found in the repository of the library, since each tag usually represents a release version of the software [8]. For each tag, we extract the commit associated with it and use the commit date as the release date of the library version. To find the Release Frequency, we calculate the average of the time difference between each two consecutive releases of the library. As an example, the Release Frequency of mockito is 9.87 days, which represents the average time between consecutive releases.

3.3 Issue Response Time and Closing Time

Definition and Intuition.

Potential clients may want to know if a given library's developers are helpful and if there is an active community around it. One quantifiable way to measure this idea is to see how quickly reported issues are replied to and resolved. We refer to *Issue Response Time* as the average time that it takes to receive a comment once a bug report has been opened. Similarly, we define *Issue Closing Time* as the average time that it takes to close a bug report since it was originally opened. More formally:

$$IssueResponseTime = \frac{\sum_{i=1}^{N} IssueFirstCommentDate_i, IssueCreationDate_i}{N} \tag{3.2}$$

$$IssueClosingTime = \frac{\sum_{i=1}^{M} IssueClosingDate_i, IssueCreationDate_i)}{M} \tag{3.3}$$

where N is the total number of issues with comments and M is the total number of closed issues.

Extraction Methodology for Issue Response Time.

We used the Github API to obtain all issues found in a given library whose issue tracking system is hosted on Github. If a library used JIRA to track

issues, we downloaded all issues as an XML file by querying the URL provided by the Export XML option in the JIRA tracking system of each library and parsed the contents. For each issue, we extracted its creation date, as well as the date of the first comment on the issue. The issue response time is then calculated as the difference between the creation date and the date of the first comment. We then calculated the issue response time of the library as the average of response times for all considered issues. Note that we discarded issues that had no comments as we care about the average time the library community took to reply to issues. We used the first comment made by any user other than the original poster to calculate response time, as knowledgeable users who may not be major contributors to the project often provide useful suggestions or feedback on the issue. Our extracted Issue Response Time for mockito is 18.56 days.

Extraction Methodology for Issue Closing Time.

We again used the Github API to collect all issues posted in the library's repository for Github-hosted tracking systems. Similar to the previous extraction methodology, we exported all issues as an XML file for libraries whose issue tracking systems are on JIRA by querying the URL provided by the Export XML option. As this metric focuses on closing times, we discarded issues which were not in a closed state. For each issue, we extracted its creation and closing dates. For bug reports hosted in JIRA, we used the resolved date as the closing date for issues that had a closed status, as we found that JIRA does not provide an explicit closing date in its issue reports. The closing time of each issue is the difference between the creation and closing dates. Finally, to calculate the issue closing time metric for the library, we used the average of the closing time of all closed issues. We extracted an Issue Closing Time of 70.84 days for mockito.

3.4 Backwards Compatibility

Definition and Intuition.

A library is said to be backwards compatible if client projects can upgrade to a more recent version of the library without having to modify code that used the library's APIs. Libraries that often break existing code may result in more maintenance work from client developers, so we believe that developers may want to see information about backwards compatibility when choosing a library. We define Backwards Compatibility as the average number of breaking changes found in consecutive releases of a library, more formally:

$$BackwardsCompatibility = \frac{\sum_{i=1}^{N-1} BreakingChanges(Release_i, Release_{i+1})}{N-1} \tag{3.4}$$

where N is the total number of releases of a library and $BreakingChanges$ is a function that counts the total number of breaking changes between two releases.

Extraction Methodology.

We used Xavier et al. [52]'s diff tool that analyzes two source code versions of a given library to detect changes for three types of API elements: type, field, and method. For types, breaking changes consist of type removal, visibility loss of the type (e.g. from public to private), and changes in its base type. For fields, field removals, modifications in the field's type, visibility loss, or different default values are considered breaking changes. Breaking changes for methods can be due to method removals, visibility loss, changes in its return type, parameter list changes, and exception list changes. We used this diff tool to count the number of breaking changes between two consecutive versions R_i and R_{i+1} of a library for all releases R_1 to R_{n-1}, where n is the number of releases of a library. Similar to our Release Frequency methodology, we obtained releases of a library by collecting git tags. Finally, we calculated the average of the resulting number of breaking changes across all analyzed

releases. We obtained an average of 165.16 breaking changes per release for mockito.

3.5 Performance and Security

Definition and Intuition.

Performance of a library refers to how efficient and optimized its code is. Security of a library shows its ability to handle sensitive information without compromising data and its robustness against attacks. Developers may want to avoid libraries with many performance and/or security problems. As proxies to measure performance and security of libraries, we use the proportion of bug reports classified as performance-related and security related. More formally, these metrics are defined as follows:

$$Performance = \frac{\#PerformanceBugs}{\#TotalBugs} \tag{3.5}$$

$$Security = \frac{\#SecurityBugs}{\#TotalBugs} \tag{3.6}$$

Extraction Methodology.

Past work by Uddin et al. [49] extracted information about performance and security aspects of APIs by mining opinions from discussion forums. However, our goal is to extract quantifiable information about these problems. As a medium to obtain quantifiable data about non-functional attributes of a library such as performance or security, we use bug reports related to these problems as our source of information. Bug reports and their associated fix commit messages have been previously used to extract general metrics or infer information about software [23], [32], [34]. Textual features of bug reports have also been used to provide classifications using machine learning techniques [21], [36]. Similar to how background checks, school transcripts, or a history of car accidents are used to determine current reliability aspects of individuals, bug reports can tell valuable information about the current performance and security of a library.

Since issue tracking systems such as Github do not provide uniform issue labels (e.g. performance) for bug reports across different repositories, we resort to a different method to extract this information. Specifically, we use a combination of a keyword-based approach and a classifier to label bug reports as performance or security related. For simplicity, we will only refer to Performance as the metric being discussed; however, we use the same exact methodology to extract Security data.

To extract Performance data for our list of libraries, we first collected the titles of all bug reports from Github or JIRA, based on where the issues are tracked for each library. Our performance extraction methodology is then divided into two steps. The first step consists of filtering bug reports based on keywords. The second step relies on automatically classifying the resulting bug reports using a machine learning classifier. We use a two-step methodology, as using only machine learning classifications resulted in a large number of false positives. We detail these steps in the following paragraphs.

Bug Report Filtering. We filter bug reports of a given library by searching the titles of these reports for keywords related to general performance problems. We manually obtained these keywords from the titles and descriptions of bug reports in the training dataset described in the following paragraphs. The keyword selection process consisted of looking for terms which, in isolation, can describe general performance problems independently of the library domain or component. If a term met this criteria, we added it to our set of keywords. The final list of keywords can be seen in Table 3.1.

Performance	*deadlock, efficient, fast, freeze, hang, inefficient, memory, leak, minimize, optimize, outofmemory, overflow, perform, , scalable, slow, unresponsive*
Security	*attack, authenticate, authorize, availability, confidential, cipher, crack, decrypt, encrypt, secure, vulnerable*

Table 3.1: Performance and Security keywords used to filter bug reports.

Training Dataset. In order to use machine learning classifiers, we use a training dataset of 1,000 titles of bug reports, 500 labeled as performance bug

	Precision	Recall	F1-Measure
Multinomial Naive Bayes	0.79	0.87	0.83
Support Vector Machine	0.67	0.92	0.77
Random Forest	0.62	0.87	0.73

Table 3.2: Recall, Precision, and F1-Measure numbers for the Performance classifier

reports and the other 500 as non-performance. To create this dataset, we first use existing manually classified bug reports from the dataset by Ohira et al. [33] which contains 320 performance reports and 161 security reports. Additionally, in order to have 500 positive examples, we complement the dataset by adding bug reports that two researchers, including the author of this thesis, manually classified. Our manual classification was done as follows: we first search in the Bugzilla issue tracking system for bug reports containing the keyword *performance*. As projects, we used Tomcat, Eclipse, Ant, Thunderbird, and Firefox to perform our search as their issue tracking systems have a large number of bug reports and do not overlap with any of our target libraries. We manually classified the titles of bug reports as performance or non-performance and agreed on the labels. We then collected the titles of the bug reports which were classified as performance.

Bug Report Classification. Using the training data set mentioned in the last paragraph, we train a machine learning classifier to provide performance or non-performance classifications for bug reports, using only the title of the bug report as the input data. We omit bug descriptions from our input since we found that they often consist of steps on how to reproduce an issue, which do not provide valuable information to the classifier, and often add noise. For each bug report title, we eliminate stop words, and stem the remaining words from the resulting text. We then calculate the inverse term frequency of the resulting text and used it as input to a Multinomial Naive Bayes classifier. We use Multinomial Naive Bayes as it produced better F1-score numbers when compared to other classifiers that we tried, as seen in Tables 3.2 and 3.3.

We validate our classifier using a stratified 10-fold cross-validation and achieve a recall of 79% and a precision of 87% (recall of 89% and precision of

	Precision	Recall	F1-Measure
Multinomial Naive Bayes	0.88	0.89	0.88
Support Vector Machine	0.99	0.57	0.72
Random Forest	0.71	0.80	0.75

Table 3.3: Recall, Precision, and F1-Measure numbers for the Security classifier

88% for our security classifier). For our final performance metric for a given library, we report the percentage of bug reports classified as performance-related out of all issues found in the issue tracking system of the library. For this metric, we consider all issues regardless of state, as closed issues can reveal past problems of a library, while open reports may reveal current problems. Using this methodology, we calculated that mockito has 2.14% performance-related issues and 0% security-related issues in its issue tracking system.

3.6 Last Modification Date

Definition and Intuition

We refer to the *Last Modification Date* of a library as the last time changes were done to its code repository. While the Release Frequency of a library provides an estimation on how often the software is updated, it does not provide information about the recency of the library. Last Modification Date provides an indication of whether the development of a library is still active.

Extraction Methodology

We extract the date of the last commit made in the Github repository of the library. When extracting data for our survey, the Last Modification Date for mockito was January 15, 2018.

3.7 Last Discussed on Stack Overflow

Definition and Intuition

This metric refers to the last time a question was posted about a given library in the popular Q&A website Stack Overflow. Similar to the Last Modifica-

tion Date, we believe that libraries with active communities are constantly discussed in Q&A websites such as Stack Overflow.

Extraction Methodology

For this metric, we collected the Stack Overflow tags corresponding to the libraries in our list. We identified the tags for each library by using the *search by tag* functionality on the Stack Overflow website. Using this feature, we searched for the name of the library and selected the tag with the largest number of questions. Using the Stack Overflow API, we then looked for the most recent questions containing the identified tag of a given library and extracted the date of latest posted question. As an example for this metric, mockito was last discussed on January 15, 2018 at the time we prepared the data for our survey.

3.8 Chapter Summary

In this Chapter, we presented the metrics that we employed for a first implementation of a metric-based comparison of software libraries. The selected metrics were: Popularity, Release Frequency, Issue Response Time, Issue Closing Time, Backwards Compatibility, Performance, Security, and Last Discussed on Stack Overflow. These metrics have the objective of providing quantifiable information describing different characteristics of libraries. These metrics were extracted from open source repositories, issue tracking systems, version control systems, and Q&A websites.

Chapter 4

Developer Survey

To evaluate our proposed methodology of comparing libraries using metrics, we implemented a preliminary version of our technique. Using the 9 metrics described in Chapter 3, we extracted their metric data for a set of libraries and used this information to survey software developers about the usefulness of our employed metrics and our approach for comparing libraries. More specifically, we designed the survey with the following 3 research questions in mind:

RQ_1 Is a metric-based comparison of libraries useful for developers when selecting a software library to use?

RQ_2 Which metrics influence developers' library selections when comparing software libraries?

RQ_3 Does the library domain affect metric usefulness?

The rest of this chapter is organized as follows: Section 4.1 mentions the subject library domains and the criteria used to select libraries for our survey. Section 4.2 describes how the survey was presented to participants and Section 4.3 contains the questions and possible answers of the survey. Finally, 4.4 explains the recruitment strategies employed to contact potential survey participants.

4.1 Surveyed Libraries

To compare similar libraries, we chose 10 of the most popular Java library domains in MVNRepository [16], a website that categorizes Java libraries based

on domains. These domains were testing, database, utilities, xml processing, logging, object relational mapping, json processing, mocking, security, and cryptography. For each domain, we investigated the available Java libraries by consulting the same website. For this implementation, we focused on libraries that have available open software repositories on Github and issue tracking systems, either hosted on Github or Jira. We discarded libraries whose projects did not meet these requirements. Additionally, since some of our metrics depend on text analysis, we ignored libraries whose bug reports were written in languages other than English. For instance, we found that this was the case with the *alibaba/fastjson* repository, whose main language for issues was Chinese. Our final subject list consisted of 34 Java libraries.

4.2 Survey Overview

After gathering the metric data for our chosen libraries, we implemented a website to conduct our developer survey. Our website first presented participants with a set of background questions about their professions and Java proficiency levels. Then, the website proceeded to display a list of library domains available for surveying. To obtain a balanced set of responses for each domain, our website dynamically hid domains whose number of responses were too high compared to the least responded domain, and thus, encouraging users to evaluate the least popular domains. After selecting a domain, participants were able to see a table containing a metric-based comparison of libraries from the selected domain, similar to the table in Figure 1.1. Each metric presented in the table contained an information icon that users were able to hover over for a simple description of the metric. Below the table, a set of domain-specific questions were shown to participants. Finally, a set of exit questions were displayed to participants.

4.3 Survey Questions

We now detail the complete set of questions in our survey along with their respective possible answers.

Background Questions

Q_{B1} *What is your current occupation?* Undergraduate student, graduate student, academic researcher, industrial researcher, industrial developer, freelance developer, or other.

Q_{B2} *How many years of Java development experience do you have?* <1 year, 1-2, 2-5, 6-10, 11+ years.

Domain-specific Questions

Q_{D1} *Based on the presented information, which of the above libraries would you select if you were looking for a [domain] library to use?* A dropdown list containing the names of the libraries shown in the table. Only one library could be selected. Note that the choice of library per se is not important for our survey objectives. The purpose of this question, as well as the next two, is to ensure that participants think about the presented metric information.

Q_{D2} *Which of the above libraries have you used before?* A checklist containing the names of the libraries shown in the table. Multiple libraries could be selected.

Q_{D3} *Name any other libraries from this domain that you have used before.* Free-text.

Q_{D4} *Explain your reasons for your choice in Q_{D1}. Which metrics, if any, influenced your decision?* Free-text.

Exit Survey Questions

Q_{E1} *On a scale of 1 to 5, with 1 being not useful at all to 5 being very useful, how would you rate the usefulness of the following metrics?* A list of the 9 metrics is presented, with a Likert scale from 1 to 5 for each metric.

Q_{E2} *On a scale of 1 to 5, with 1 being not useful at all to 5 being very useful, how useful do you find the above metric-based comparison for selecting a library to use from a given domain?* Likert scale from 1 to 5.

Q_{E3} *Are there additional metrics that you think might be helpful for comparing libraries in a given domain?* Free-text.

Q_{E4} *Please let us know if you have any further comments about the above metric-based comparison of libraries.* Free-text.

4.4 Participant Recruitment

To obtain valuable feedback for our approach, we needed to survey Java developers who use libraries. We employed three recruitment strategies to target this audience. The e-mail templates used to recruit participants can be found in the Appendix A.1, A.2, and A.3 of this thesis.

Github Recruitment. To recruit developers who have used one of our subject libraries before, we searched for Github users who contributed code to client projects of our subject libraries. As it is not possible to contact Github users directly, we gathered e-mail addresses as follows. We used the Github API to search for Java repositories that included code that references API elements of our subject libraries as the only filtering criteria. From these results, we looked at the files containing our searched API elements and collected the git commits associated with them. Finally, we discarded git commits that were older than 6 months and obtained the e-mail addresses of the authors of the remaining commits. We collected a total of 298 e-mail addresses.

Stack Overflow Recruitment. Our second recruitment strategy consisted of recruiting Stack Overflow users who have been involved in discussions of two subject libraries of the same domain. Such a discussion indicates that they were previously comparing libraries and thus getting the input of such users is important. Since Stack Overflow does not offer functionality to contact users directly, we searched for questions containing at least two tags of our subject libraries and collected e-mail addresses by visiting the personal websites listed in the profiles of the users involved in those questions. Given the manual

nature of this methodology and the fact that not all users have websites on their profiles, we collected only a total of 20 e-mail addresses in this step.

Snowball Sampling Our third strategy consisted of snowball sampling [10] where we simply asked any Java developer to complete the survey. We sent email invitations to the undergraduate and graduate mailing lists of the Computer Science department at our university, who may had forwarded the email to others. We also advertised the survey on social media accounts and invited others to promote the survey.

4.5 Chapter Summary

In this chapter, we explained the details of the survey that was presented to developers with the objective of evaluating the usefulness of a metric-based comparison of software libraries. Our survey consisted of an initial implementation of our comparison methodology, which was used to compare libraries from 10 popular domains, and a set of questions whose objective was to make developers think about the metrics and the usefulness of the presented information. To recruit developers for surveying purposes, we used three strategies which targeted users of Java libraries.

Chapter 5

Results

The previous chapter introduced the developer survey designed to obtain feedback about our first implementation of a metric-based comparison of software libraries using the metrics specified in Chapter 3. In this chapter, we present the results of this survey, information about the response rate and participants' background, statistics about evaluated domains and participants' answers, conclusions to the research questions posed in the previous chapter, as well as a discussion involving the free-form answers from the survey.

All of the reported statistics in this chapter use a pairwise Wilcoxon sum rank test to observe any statistically significant differences between metric ratings, using $\alpha = 0.05$. Since we perform multiple tests, we use the Holm's adjustment method for our p values. To estimate effect sizes of any significant differences, we use Cliff's delta with the following ranges [12], [23]: small for $d < 0.33$, medium for $0.33 <= d < 0.474$ and large for $d >= 0.474$.

The organization of the rest of this chapter is as follows: Section 5.1 contains basic statistics related to the survey participation, while Section 5.2 has information about the background of participants. In Section 5.3, details about the evaluated domains and selected libraries by participants are mentioned. Sections 5.4, 5.5, and 5.6 provide answers to the first, second and third research question, respectively. Finally, Section 5.7 discusses the free-form answers and comments provided by developers.

Figure 5.1: Occupation

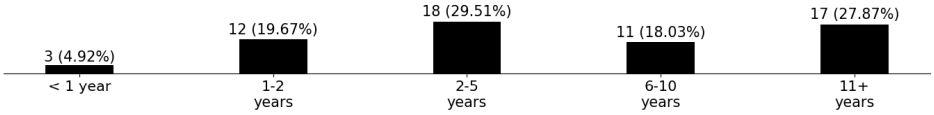


Figure 5.2: Years of experience using Java

5.1 Participation Breakdown

We received a total of 61 responses for our survey. From the 318 e-mails that we sent to Github and Stack Overflow users, we collected 24 responses (7.5% response rate), with 21 participants recruited from Github and 3 from Stack Overflow. The remaining 37 responses were obtained with the snowball sampling recruitment. Out of the 61 total responses, 53 participants completed all sections of the survey, while 8 participants answered only the background and domain-specific questions but did not answer the exit questions. A total of 22 participants evaluated at least two domains, while the remaining participants assessed only one domain.

5.2 Background of Participants

Figure 5.1 shows the distribution of participants' backgrounds. For simplicity and better visualization, we group undergraduate and graduate students under the category Student. Similarly, we use the category Researcher for both industrial researcher and academic researcher, and Professional Developer for both industrial developer and freelance developer. Students present the highest percentage of participants (49.18%), followed by Professional Developers (39.34%). Note that there was one participant who picked the other category in occupation and indicated that they are a business analyst. Figure 5.2 shows the distribution of years of Java experience among participants. Three

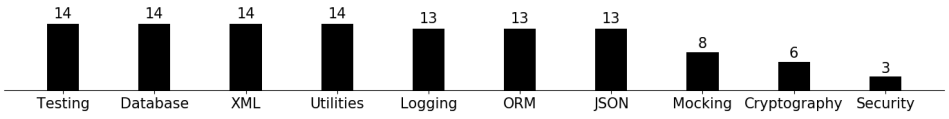


Figure 5.3: Number of evaluations for each domain

Table 5.1: Most selected libraries per domain.

Domain	**Most Selected Library**	**% of Participants with experience using the library**
Logging	qos-ch/slf4j	53.85%
Utilities	google/guava	71.43%
Mocking	mockito/mockito	100%
Cryptography	bcgit/bc-java	66.67%
JSON	google/gson	61.54%
Database	h2database/h2database	50%
ORM	hibernate/hibernate-orm	76.92%
Security	spring-projects/spring-security	33.33%
XML	apache/xerces2-j	42.86%
Testing	junit-team/junit4	92.86%

quarters of the participants (75.41%) had at least 2 years of Java experience.

5.3 Domain and Library Breakdown

The number of evaluations per library domain are shown in Figure 5.3. Note that the same participant may have evaluated more than one domain. Testing, Database, XML, and Utilities are the most evaluated domains, each consisting of 14 domain-specific survey responses, while Security is the least evaluated with 3 responses. Table 5.1 shows a detailed breakdown of the selected libraries. In 29.46% of responses to Q_{D1} and Q_{D2} of the survey, participants chose a library they had not previously used, which supports the idea that familiarity with the library was not the deciding selection factor for all participants.

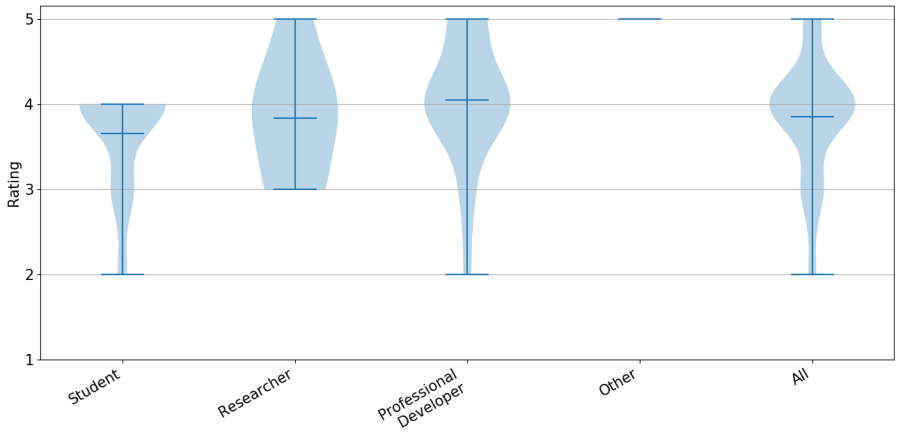


Figure 5.4: Answers to Q_{E2} by participant occupation

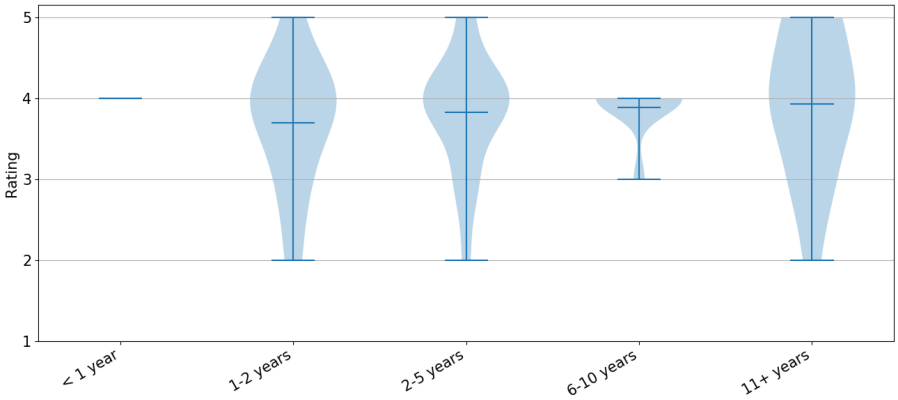


Figure 5.5: Answers to Q_{E2} by participant experience

5.4 RQ1: Is a metric-based comparison of libraries useful for developers when selecting a software library to use?

We start by first answering RQ1, which evaluates the overall usefulness of metric-based library comparisons. We do so by analyzing the 53 received answers for Q_{E2} from participants. Figures 5.4 and 5.5 show the distribution of ratings for Q_{E2} per occupation and experience groups respectively. The first four violin plots of Figure 5.4 present the distributions of ratings per occupation group, while the fifth plot shows the distribution of ratings by all participants. When considering all participants, the right-most plot in Figure

5.4 shows that the highest frequency of answers is concentrated near rating 4 (i.e., *Useful*) based on the Likert scale. The mean rating of all participants for Q_{E2} is 3.85, the median is 4, and the interquartile range is 0 (lower quartile = 4, upper quartile = 4). By observing the first three plots in the figure (we ignore the Other category as it only contains one participant), we can see that Professional Developers have rated the usefulness of a metric-based comparison the highest with a mean rating of 4.05 (median = 4, interquartile range = 0.75, lower quartile = 4, upper quartile = 4.75). Researchers have an average rating of 3.83 (median = 4, interquartile range = 1.25, lower quartile = 3, upper quartile = 4.25), and Students have an average rating of 3.65 (median = 4, interquartile range = 1, lower quartile = 3, upper quartile = 4). We observe medium effect sizes between the rating distributions of Students and Professionals (d = 0.46), between Researchers and Professionals (d = 0.37), and a large effect size between Students and Researchers (d = 0.78). Despite of this, we find no statistically significant differences between the ratings of the occupations. On the other hand, Figure 5.5 shows a small ascending trend of the ratings as experience increases starting from the 1-2 years group, but we also find no significant differences between the ratings of the Java experience groups. We can therefore conclude that based on our sample, background does not affect the rating provided by participants.

Finding.1: When comparing software libraries, our participants, regardless of background, find a metric-based comparison of libraries useful (mean rating = 3.85, median = 4)

5.5 RQ2: Which metrics influence developers' library selections when comparing software libraries?

Next, we aim to investigate which metrics influence developers' library selections when comparing software libraries. To answer this question, we use three sources of data from our survey. The first is the individual ratings of each metric in question Q_{E1}. The second is the free-form answers to Q_{D4} where participants explicitly mentioned which metrics affected their decision

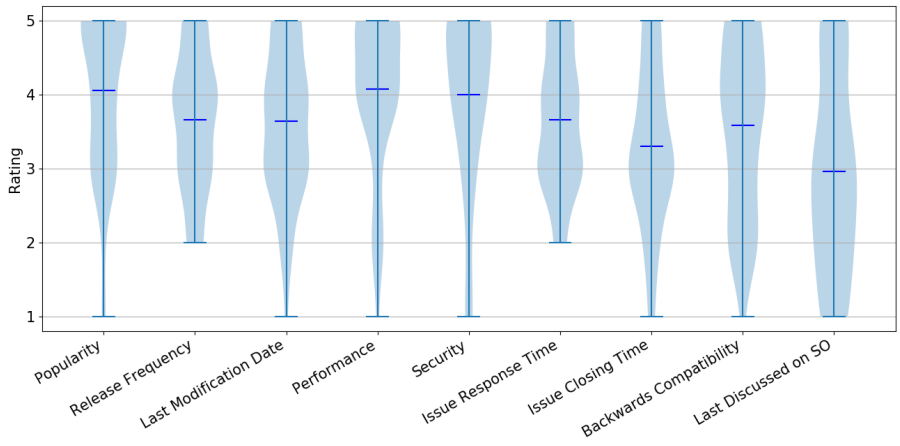


Figure 5.6: Rating distributions of Q_{E1} for all metrics.

when selecting a library and why. The third is the free-form question Q_{E3}, where participants mention additional metrics they would like to see.

Figure 5.6 shows the distribution of the ratings for each metric and also indicates the mean score. All metrics have a mean above 3.0 with the exception of the *Last Discussed On Stack Overflow* metric, whose mean is 2.96. *Performance*, *Popularity*, and *Security* were the 3 highest rated metrics by participants, with mean ratings 4.08 (median = 4, interquartile range = 1, lower quartile = 4, upper quartile = 5), 4.06 (median = 4, interquartile range = 2, lower quartile = 3, upper quartile = 5), and 4.00 (median = 4, interquartile range = 1.5, lower quartile = 3.5, upper quartile = 5) respectively. Additionally, we find that each of these 3 metrics has statistically significant differences with the metrics *Issue Closing Time* and *Last Discussed on Stack Overflow*. Analyzing the magnitude of these differences, we observe medium effect sizes ($d = 0.38$) between Popularity and Issue Closing Time, and between Popularity and Last Discussed on Stack Overflow ($d = 0.45$). For *Security*, the results show a medium effect size with Issue Closing Time and Last Discussed on Stack Overflow ($d = 0.38$ and 0.42 respectively). Finally, for Performance, the tests also reveal medium effect sizes with Issue Closing Time ($d = 0.41$) and with Last Discussed on Stack Overflow ($d = 0.45$). Thus, we can conclude that compared to the other metrics, Performance, Popularity and Security are indeed metrics that have more influence on developers' decisions.

To gain insights about the top-rated metrics, we analyze the free-form answers from Q_{D4}. We use P1 to PN when quoting different participants. We observe that participants are concerned about possible unnecessary overhead to their applications that libraries may add, which might serve as explanation for the high mean for Performance. For example, P1 said that *"[...] logging already added some overhead, I don't really want any performance issues with the library that I'm using"*.

For Popularity, we observe that participants associate this metric with library quality and support, as the comments of P2, P3, and P4 suggest: *"Popularity is a good proxy for quality"*, *"... These 2 metrics [Popularity and Last Modification Date] are a good indicator of the quality of a library"*, and *"It's popular so I guess it should have something good"*. Moreover, Popularity is also associated with library support, as we can infer from these comments by P4, P5, and P6: *"Popular libraries tend to have the most support"*, *"...usually more popular libraries are better developed and supported"*, and *"Popularity reflects the sustainability of the library"*.

Although Security is positioned as the third highest rated metric, we did not find explanations for this fact in our collected answers. However, since several of our subject domains deal with handling of data, we speculate that developers most probably consider integrity and confidentiality of data as an important factor when choosing libraries.

Finding.2: Performance, Popularity and Security are rated highest, and each had two statistically significant differences in their ratings w.r.t other metrics.

To find additional metrics that developers care about that we did not consider in our comparison, we analyze the free-form answers for Q_{E3} of the Exit Survey. This question explicitly asked participants about additional metrics they would like to see. In total, 39 of our participants left comments for this question. To analyze their responses, we use an open coding approach from grounded theory [4], specifically card sorting. Two researchers, including the author of this thesis, wrote each metric that a participant mentioned on a piece of paper. Then, we iteratively grouped related metrics, until categories of sim-

Table 5.2: Categories of additional metrics from Q_{E3} and the number of participants mentioning these metrics.

Category	**# of participants**	**Definition**
Documentation Quality	14	Recency and availability of library documentation and learning materials
Library Usability	8	Metrics related to the ease of use and learning curve of a library.
General Library Information	8	Statistics about a library and its public repository.
Library Functionality	5	Information about the functionality offered by a library.
Community Support	5	Statistics about community aspects from a library.
Legal	3	Information about licensing and ownership of a library
Compatibility	3	Information about the library's compatibility with other software such as libraries, platforms, and programming language versions.
Dependencies	2	Other software that needs to be installed to use a library.
Library Alternatives	2	Information about similar libraries including from other programming languages.
Crowd-sourced Opinions	2	Opinions and reviews found in Q&A websites
Robustness	2	Reliability of a library.
Quality Assurance	3	Information about testing suites and continuous integration statistics of a library.
Memory	1	Information about the memory usage of a library.

ilar metrics were formed. Table 5.2 shows the categories that emerged, along with the number of associated answers and the definition of the category. Note that two participants mentioned popularity metrics, which we do not report in the table since Popularity is a metric already covered in our comparison. Additionally, we did not understand the comments by 2 participants so we did not include them in our categorization.

Finding.3: Additional metrics related to documentation and usability of a library are highly desired by developers.

5.6 RQ3: Does the library domain affect metric usefulness?

Since our metric comparison is organized by domain, we are interested to see if the perceived usefulness of the metrics depends on the domain. To investigate this, we use two sources of data. The first is the individual metric ratings from Q_{E1} from the exit survey, and the second is the free-form answers to Q_{D4} from the domain-specific questions where participants explicitly mention the metrics that affected their decision.

For Q_{E1}, we analyze the metric ratings per domain by using survey responses that evaluated a single domain. This means that we do not consider

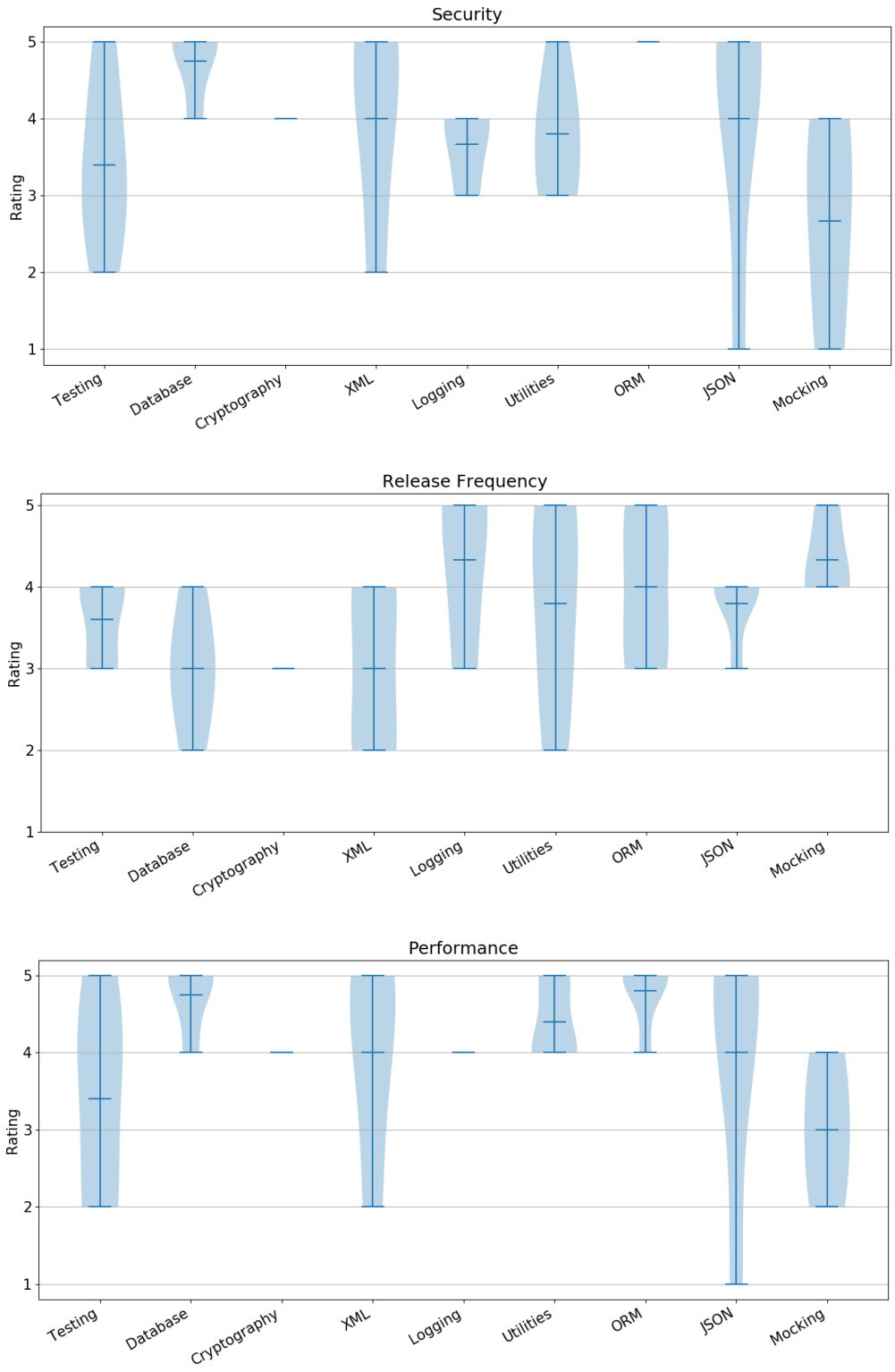


Figure 5.7: Usefulness ratings for the Security, Release Frequency and Performance metrics by domain

participants who evaluated more than one domain, such that we can get a one-to-one mapping between domain and metric rating. We have 35 such ratings to analyze.

We find that while there are no large differences in the violin plots among most domains for metrics such as Popularity and Last Modification Date, the plots for the Security metric ratings show high variability among domains. We visualize three metrics, including Security, that show lack of consistency in their violin plots among domains in Figure 5.7. Plots for the remaining metrics can be found in Appendix A.4 of this thesis. Note that we do not include a column for the Security domain in our plots since there are no responses that evaluated only this domain.

Given that the number of one-to-one domain-metric ratings, that is, the metric ratings belonging to a survey response where only a single domain was evaluated, was not that high (e.g., 1 rating in the Cryptography domain vs. 5 ratings in the JSON domain), we do not perform statistical tests to compare these populations since the results of the tests would be meaningless. Instead, we triangulate the descriptive statistics in Figure 5.7 with the answers to Q_{D4}. We have 108 comments in Q_{D4}, since many participants evaluated more than one domain. We analyze the comments and count the number of times each metric is mentioned. We present a heat map in Figure 5.8 based on these counts. Darker colors indicate more mentions by participants. The heat map confirms our results from the metric ratings in the previous paragraph. We can see that Popularity is frequently explicitly mentioned as a reason to choose a library for most domains. In contrast, the Last Discussed on Stack Overflow metric was barely mentioned by participants. As previously discussed, Security seems to be a metric that is only relevant in some domains, such as databases and cryptography, but completely irrelevant in other domains such as utilities or testing.

To understand why metric usefulness may vary among domains, we look at explanations provided by participants in Q_{D4} as well as any relevant comments from question Q_{E4}. We find that Security is intrinsic to certain domains: *"Since this is a crypto API, I'd like to use a library that has the least security*

issues" - P1. On the other hand, Security provides little use for specific domains to some participants, as P7 suggests: *"Importan[ce] of metrics depends on the library. I don't care about the security of JUnit."* We can observe this opinion reflected in the Security metric ratings of Figure 5.7, where the testing and mocking domains have low ratings. Similarly, Performance is seen as crucial for some domains, as P8 describes *"In ORM, the first thing I care about is performance in enterprise projects"*. For Release Frequency, certain domains may provide more stable library releases and therefore, the metric is less important, as P7 points out: *"Testing frameworks do not need to be released all the time. They are stable."*

Finding.4: Some metrics are more intrinsic to certain domains than others. Examples include Security, Performance, and Release Frequency.

5.7 Discussion of free-form comments

A metric-based comparison of software libraries has two main challenges: (1) knowing which metrics to include and (2) designing a method to fairly and accurately extract these metrics and present them in a useful and intuitive way. Our survey focused on the first challenge: it evaluates whether a metric-based comparison of software libraries is useful to developers, and helps uncover which metrics developers care about when making a choice. However, the comments provided by our participants provide us valuable suggestions and insights on how to address the second challenge. In the rest of this section, we discuss the most requested metrics that we did not include in our work as mentioned by our survey participants and we provide our insights on how these suggested metrics could be extracted. We also discuss improvements that participants wanted to see in the set of metrics employed for the survey.

Table 5.2 shows that documentation quality is by far the most demanded metric, even though it is hard to quantify, as mentioned by one participant. There is a vast amount of work on how to improve documentation, such as the research by Robillard et al. [42]. While concretely quantifying documentation is hard, simple techniques such as mining various information sources and presenting related links on our metric-based website, e.g., through tech-

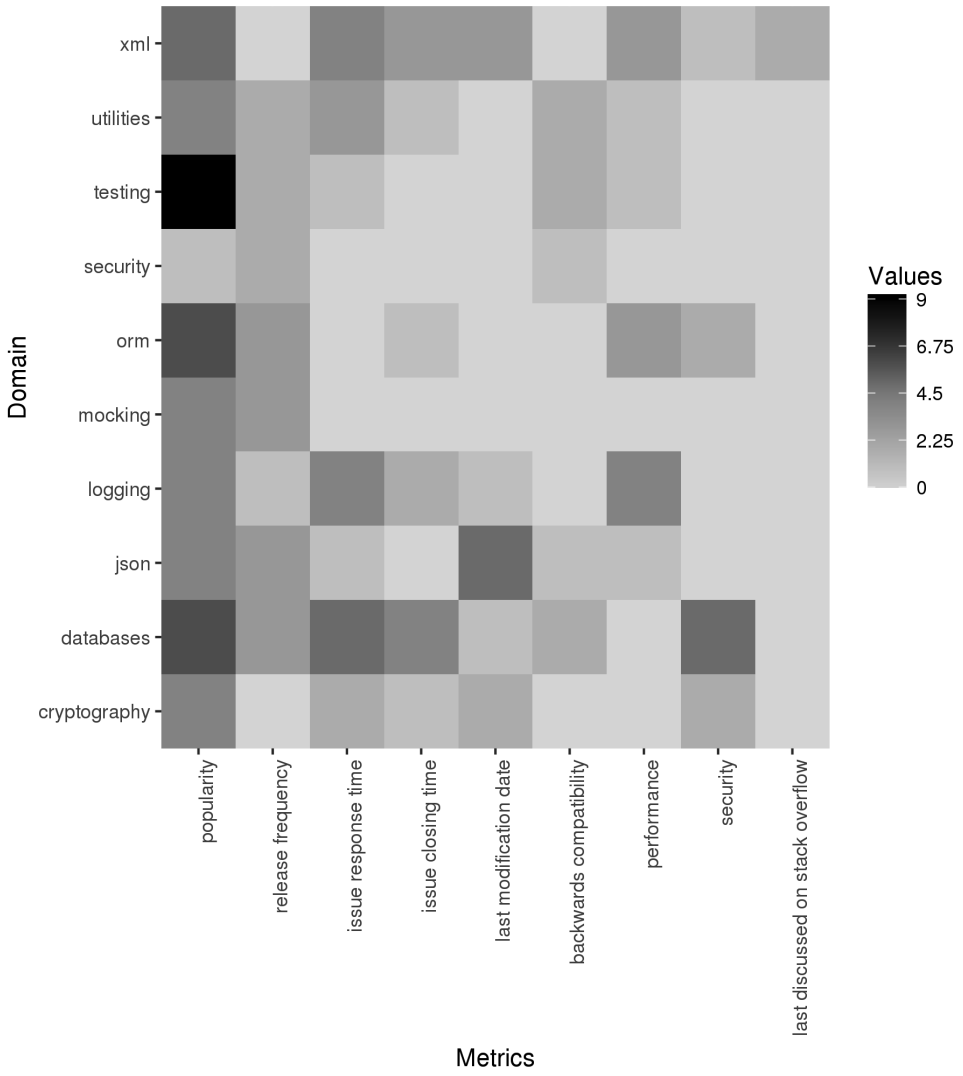


Figure 5.8: Heatmap of frequency of metrics mentioned by participants in Q_{D4}. Darker colors indicate higher frequency.

niques that link the API used in the documentation to the library [45], are feasible and could be useful to developers. Other options include automating documentation search techniques, such as those presented by Parnin and Treude [38], or using patterns of knowledge in API documentation as a means to assess their quality [24]. Future work could investigate how existing documentation techniques can be summarized and quantified in a way that quickly gives developers an indication of availability and quality.

The next most demanded metric relates to library usability; specifically, how easy it would be to use the API of a library. Since this is also hard to

quantify, one option could be to leverage the vast amount of research that is able to mine API usage examples [41] and present some of these instances to developers. Developers could then assess whether these examples seem easy to use or not.

While our selected metrics do have some general information about the library (e.g., Last Modification Date), our participants provide many other concrete suggestions that pertain to the general information of the library. For example, surprisingly, two participants mention code size while we thought client developers would not necessarily care about low-level details of the library. Others mention more high-level general information metrics such as absolute number of releases and how old the library was.

Five participants mention that understanding the functionality offered by the library is important, while one participant wishes to see the overlap in functionality with other libraries. Another comment suggests having code examples for each of the library functionalities. The community support around a library seems to also be very relevant for users. While we try to capture that with the Issue Response Time and Last Discussed on Stack Overflow metrics, it seems that developers are looking for additional metrics such as the size of the community and how often the library is discussed in various blogs and resources. Such metrics could be automated, by mining the web, or through more advanced techniques such as looking at related developer networks.

Interestingly, one metric that we did not think of but that is very important in practice is the licensing of the library. To add this information, GitHub could be used as it already specifies the licenses used in each project, or license mining techniques could also be employed [50]. Similarly, it makes sense that developers are interested if a given library is compatible with libraries that they are using, as well as the list of dependencies they would need to include for this library to work correctly. Techniques for mining library dependencies can also be used to extract such information. The remaining categories are mentioned by two or less participants, but it is worth noting that crowd-sourced opinions are mentioned by two of the participants. Information from the summaries of API reviews created by Uddin and Komh [49] could be integrated in our

comparison to address this.

As suggestions for the set of metrics used for the survey, participants mentioned improving the presentation of the metric information, such as adding graphical representations of the data (e.g. using trend graphs to visualize metrics such as Issue Closing Time and Issue Response Time) or aggregated scores for each library (e.g. 5-star rating for each library summarizing all of its metrics data, as a quick way to observe which library is the highest rated.). Finally, based on Finding 4, we believe that displaying customizable metric-based comparisons based on the domain (e.g. displaying the most important metrics of the domain first) may be useful to users. The fact that distinct participants in our survey often rated the same metric differently supports our reasoning that developers may care about different characteristics of a library depending on their needs. This further sustains our belief that it is important to provide developers with all relevant data, in an easily comparable form, and leave them to decide which parts of this data they will use for their choices.

5.7.1 Chapter Summary

In this chapter, we presented the results of the survey conducted to software developers where our objective was to investigate the following points: the usefulness of a metric-based comparison of libraries when selecting software libraries, the metrics that influence developers' library selections when comparing software libraries, and whether the library domain affects the usefulness of metrics. The survey results show that most developers, regardless of background, found the metric-based comparison useful. Among the metrics included in our implementation, Performance, Popularity, and Security were rated the highest, while additional metrics related to documentation and usability were the most requested by developers. Finally, we found that some metrics such as Security, Performance, and Release Frequency are more intrinsic to specific domains.

Chapter 6

Enhancements and Website

One of the goals of this work is to create a website where comparisons of software libraries are shown in an intuitive manner, metric information is constantly being extracted in an automated fashion, and metric feedback is received as users interact with the website with the objective of knowing which information is most useful to software developers when choosing libraries. To reach this point, we presented an initial set of metrics and their respective methodologies to be used in our first implementation of a metric-based comparison of libraries in Chapter 3. We introduced this initial implementation of our approach to developers along with the survey detailed in Chapter 4 with the purpose of obtaining feedback. We analyzed the results of the survey in Chapter 5 and mentioned the suggestions made by participants. Using the collected feedback from our survey, our next step was to make improvements and enhancements to the metrics and build the final website that can be used by the community.

This chapter details the changes that we made to our metrics and gives an overview of the created website. The rest of this Chapter is organized as follows: Section 6.1 specifies the modifications made to the previous set of metrics as suggested by participants in our survey. In Section 6.2 we give an overview of the website application created to show comparisons of libraries. Finally, Section 6.3 explains the main features of the website. The website application can be visited at [29] and the source code for the extraction of metrics is available in a public Github repository [30].

6.1 Metric and Library Modifications

Based on the feedback collected from the free-form questions in our developer survey, we made changes to our set of metrics. Additionally, we added 3 more library domains, namely Machine Learning, Collections, and Mail Clients, which consist of a total of 50 libraries as our list of subject systems. We now discuss the details of the modifications we made.

6.1.1 Popularity

In order to present up-to-date information about the subject libraries in our website, we implemented the extraction methodology for the Popularity metric from scratch, as the latest available Boa dataset dated back to September 2015. Using the same idea as the methodology in Chapter 3, we query the Github API to obtain a list of Java repositories that have at least 1 fork and were updated within the last 2 years. We clone such repositories and search for Java import statements corresponding to target libraries in the Java files of the resulting repositories. Filtering repositories with at least 1 fork is a criteria that has been used previously by researchers with the aim to discard low quality projects [1].

6.1.2 License Information

License information about libraries was one of the requests made by participants. Although license information does not fit our definition of a metric (e.g. it is not quantifiable data), we decided to include it in our comparison of libraries as licensing may be a decisive factor when integrating a library in certain projects. For example, a software license that requires the disclosure of modified source code of a library may not appeal to developers not keen to share their code. We retrieve the license information using the Github API as hosted repositories on this platform already contain the license that is being used with each project.

6.1.3 Overall Score

A requested addition by participants was to include an overall score for each library as a way to quickly visualize the comparison of libraries. Although our objective with this work was not to provide a ranking of libraries since we leave developers to decide which libraries are best suited for them, we decided to implement a 5-point scale scoring system based on the metric data of each library in order to help developers with their comparison. The intuition behind this overall score is that libraries with well-rounded metric data yield high scores. The scoring system is defined as follows:

$$OverallScore = \frac{\sum_{i=1}^{N} MetricScore_i}{N} \times 5 \tag{6.1}$$

where

$$N = \|Metrics\|$$

and *MetricScore* is the following set of scores:

$$PopScore = \frac{LibraryPopularity}{max(LibraryPopularity_i) \forall i \in DomainLibraries} \tag{6.2}$$

$$RFScore = max(0, 1 - \frac{ReleaseFrequencyAverage}{365}) \tag{6.3}$$

$$PerfScore = \frac{\#NonPerformanceBugIssues}{\#Issues} \tag{6.4}$$

$$SecScore = \frac{\#NonSecurityIssues}{\#Issues} \tag{6.5}$$

$$ICTScore = max(0, 1 - \frac{AverageDaysToCloseIssues}{365}) \tag{6.6}$$

$$IRTScore = max(0, 1 - \frac{AverageDaysToRespondIssues}{365}) \tag{6.7}$$

$$LMDScore = max(0, 1 - \frac{\#DaysSinceLastModification}{365}) \tag{6.8}$$

$$BCSCore = \frac{\#NonBreakingChanges}{\#BreakingChanges + \#NonBreakingChanges} \tag{6.9}$$

The score for each metric is normalized to obtain a maximum best possible value of 1. The overall score is the sum of each metric score divided by the number of metrics and scaled to 5. *PopScore* assigns high scores to libraries whose popularity numbers are closer to the most popular library of the domain. *RFScore, ICTScore, IRTSCore,* and *LMDScore* penalize libraries with a large average of number of days in each of the respective metrics. On the other hand, *PerfScore, SecScore,* and *BCScore* assign high scores to libraries with low numbers of performance issues, security issues, and breaking changes, respectively.

6.2 Structure of Website Application

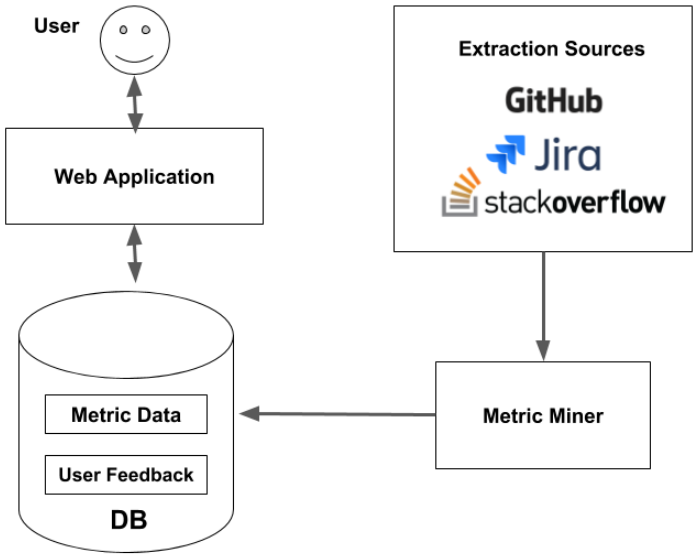


Figure 6.1: Overview of the website architecture. Line arrows indicate the direction of the flow of information.

The website application consists of three basic components: a metric miner, a database, and a presentation layer. Figure 6.1 shows the information flow within this application. The metric miner queries Github, JIRA, and Stack

Overflow to extract metric information for each of the subject libraries. The metric information is stored in temporary files and is copied to the main database once all the required data for all libraries has been extracted. Therefore, the metric data for all libraries can be updated by just executing the metric miner and additional metrics can be added by modifying this component. When a user wishes to see a comparison of libraries from a specific domain, the web application queries the database to retrieve the information necessary to show the metric-based comparisons. Finally, feedback is stored in the database when users upvote or downvote metrics when comparing libraries.

6.3 Website Features

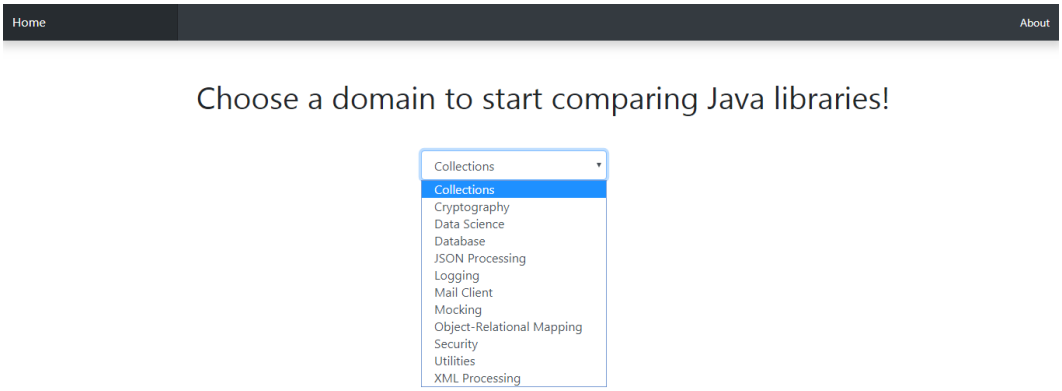


Figure 6.2: Landing page showing the available domains

We now explain the features that users would encounter when visiting this website. The main page of the website (Figure 6.2 shows a list of available software domains that contain comparisons of libraries. After selecting a domain, users are shown a table consisting of a metric-based comparison of libraries from the chosen domain, as pictured in Figure 6.3. In this table, each metric row contains a set of options which allow to obtain more information about the metric, view a graphical representation of related metric data, as well as to upvote or downvote a metric.

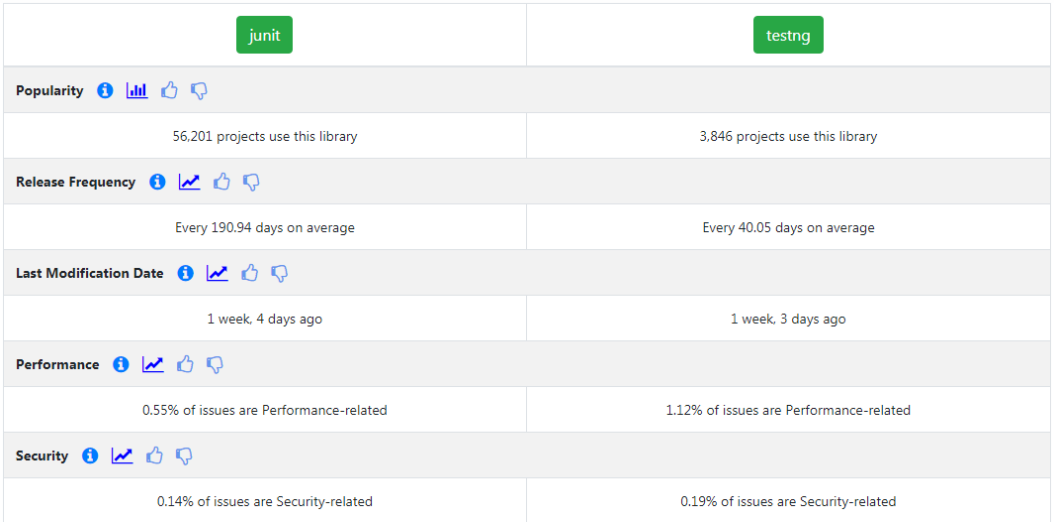


Figure 6.3: In addition to seeing the metric information, users have the option to read additional information about metrics, see graphs related to metric data, and approve or disapprove the usefulness of metrics.

A common idea suggested in the survey feedback was related to the presentation of the metric data. Some participants suggested that graphical representations to visualize data would help to know more information about a specific metric in cases where, for example, the distribution of data is skewed, and therefore, a numerical average would give a false impression of the metric data. To obtain additional opinions about graphical visualizations that were most suitable for our metrics, we resorted to CMPUT 302 (Human-Computer Interaction) students, which implemented their own graphical visualization of our metric data as part of a project class where we observed similarities to what our survey participants suggested. To address these recommendations, we now complement metric information with graphs, which users can see optionally. We now present the available visualizations for each metric:

6.3.1 Bar Charts

We use non-stacked bar charts only for the Popularity metric. Figure 6.4 shows the popularity visualization option, where each bar represents the number of client projects for each library.

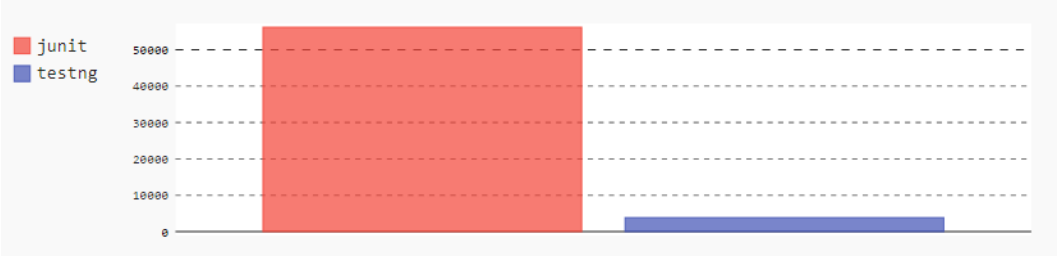


Figure 6.4: Visualization for the Popularity metric.

6.3.2 Timeline Charts

We use timeline charts for the Release Frequency, Last Modification Date, and Last Discussed on Stack Overflow metrics. For Release Frequency, a timeline of all the library releases can be seen as pictured in Figure 6.5. In this graph, a point represents a release of the corresponding library on the y-axis. Users can hover over points to see exact release dates. Similarly, the timelines for the Last Modification Date and Last Discussed on Stack Overflow metrics show points corresponding to the dates of the last 10 modifications made in the library, and the dates of the last 10 created questions in Stack Overflow related to the library, respectively.

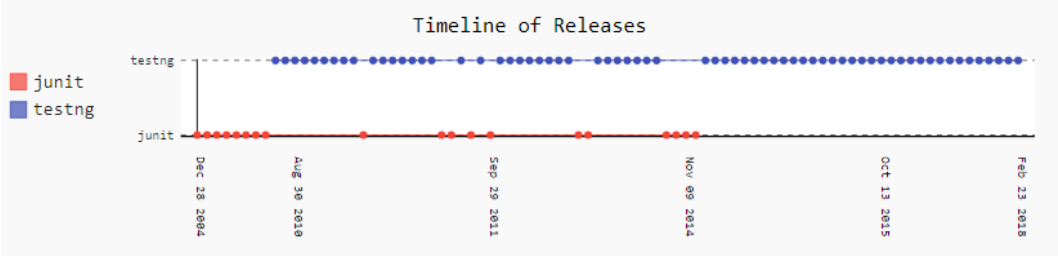


Figure 6.5: Visualization for the Release Frequency metric.

6.3.3 Stacked Bar Charts

Stacked bar charts are available for the Performance and Security metrics. In these graphs, users can observe the total number of issues classified as either related to performance, security, both performance and security, or none of these classifications in the stacked bars of this graphical representation as seen in Figure 6.6

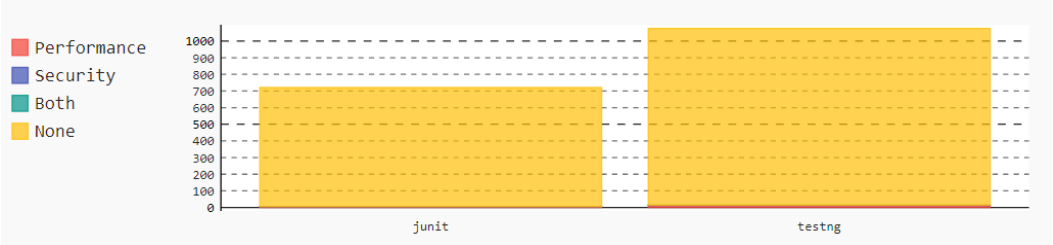


Figure 6.6: Visualization for Performance and Security metrics.

6.3.4 Line Graphs

Line graph visualizations are available for the Issue Closing Time, Issue Response Time, and Backwards Compatibility metrics, as Figure 6.7 exemplifies. The Issue Closing Time visualization shows a line graph where each point represents an issue, and the position of the point in the y-axis represents the number of days that it took to close such issue since it was first created. The Issue Response Time visualization is similar with the exception that the position of the point in the y-axis shows the number of days that occurred from the creation date of the issue to the date of its first comment. Finally, the Backwards Compatibility graph shows a line chart where each point represents a release, and the position on the y-axis reflects the number of breaking changes found in such release. Points in all graphs can be hovered over to see exact X and Y values.

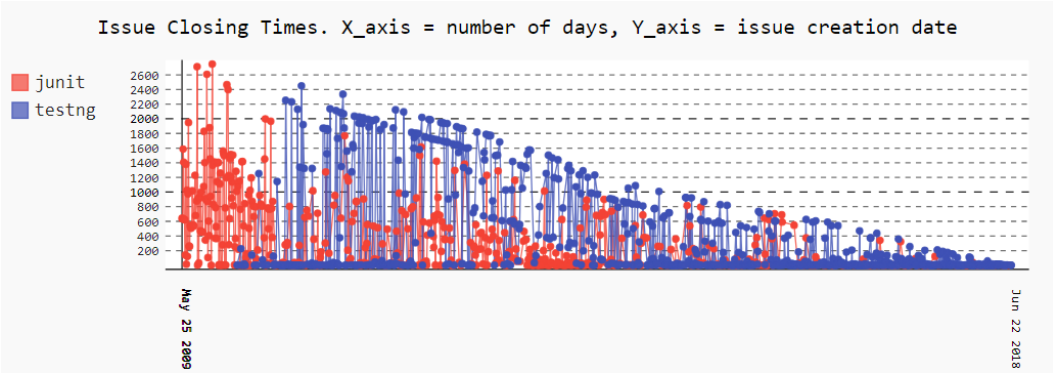


Figure 6.7: Sample line graph visualization of the Issue Closing Time metric.

6.3.5 User Feedback

In order to obtain information about the metrics that users find more useful as they observe library comparisons in each domain, we implemented a feedback system accessible from the comparison tables, as seen in Figure 6.3, where users can upvote or downvote the usefulness of the presented metric information. The feedback data is used to arrange the order in which the metric rows of comparison tables are shown, that is, metrics with the highest number of upvotes are shown at the top of tables, while the least upvoted metrics are shown at the bottom rows of tables, and ties are broken arbitrarily. Each domain has its own separate feedback data. The intuition behind this idea is that users would be able to see the most relevant metrics first, based on what previous users thought was more useful in a specific domain. Additionally, it is possible to receive open feedback in the form of comments in the *About* section of the website.

6.4 Chapter Summary

In this chapter, we introduced the website which contains the metric enhancements made to our technique based on the feedback provided by surveyed participants. We also explained basic architecture of the application, the main features of the website, namely the mechanism created to receive user feedback, as well as the available graphical visualizations available for the metric data.

Chapter 7

Threats to Validity and Limitations

In this chapter we discuss threats to validity and limitations related to the work presented in this thesis. We mention threats related to the Construct, Internal, and External validity of this work. *Construct Validity* refers to how well concepts and tools defined in this work measure what they claim to measure. *Internal Validity* refers to the degree in which the results of the experiments are attributable to the factors presented and measured in the study and not affected by third factors unknown to the researcher. Finally, *External Validity* refers to the generalization of our findings [17].

7.1 Construct Validity

Metric Names One possible threat to this study refers to the names we assigned to each one of our metrics, which may create confusion among users. For example, we used the name Popularity as the number of client projects that each library has, however, some users may think that this metric refers to the number of stars in the Github repository of a library. To mitigate this threat, we included explanations for each of the metrics in our survey in the form of information icons. Furthermore, for our website, we are including a Frequently Asked Questions section that provides more in-depth definitions of each our metrics to avoid any confusion.

Popularity Relying on `import` statements to collect popularity information may not reflect actual usage of the imported API elements. However, it provides an upper bound estimation of the popularity of a library. Another threat concerns the age of the dataset of projects used to obtain our data, as it dates to September 2015. While this dataset may not reflect current library usage trends, and would be older than the data used for the other metrics, it still shows the relative popularity since the same data set is used for all libraries and this is sufficient for the purposes of our survey. For our website implementation, we do not depend on Boa for this task in order to provide updated data.

Release Frequency When calculating this metric, we did not discard releases that may have been release candidates or beta versions of a particular library. This may have impacted the release frequency numbers for some libraries. However, we believe that this threat does not affect the findings of this work, as the metric data shown to participants had the objective of making them reason about the metrics that are important when choosing libraries.

Issue Response Time and Closing Time Past work has shown that bug reports are used as operational data by organizations and development teams, and not meant for research purposes [2]. This means that details of bug reports may not accurately reflect the true story behind each issue. However, our objective with these two metrics was to convey to developers an estimation of time for response and closing times for bug reports which we believe is not affected by this threat.

Performance and Security As we mentioned in the previous paragraph, bug reports are not intended to be used for research. Therefore, using issues to detect performance or security problems of libraries may inaccurately reflect the information related to these non-functional requirements. Additionally, we do not run performance or security tests on these libraries, nor do we examine other sources of information which may have information about these

problems, which means that we do not provide an exhaustive list of issues or vulnerabilities. Nonetheless, we believe that bug reports related to these problems may still have valuable information in their title descriptions that may indicate the presence of performance or security problems. Finding objective measurements for performance and security was one of the most difficult parts of this research, future work should investigate alternative measurements that may further mitigate this threat.

Backwards Compatibility For this metric, we did not consider changes that do not generate compiler errors, but modify the functionality of existing methods, resulting in undesired behavior. Future work could analyze regression tests in continuous integration builds to include this type of changes.

Last Modification Date When obtaining the last modification date of repositories, we do not analyze the types of changes made in the latest commit. Therefore, minor changes made within the repository such as correcting typographical errors, may inaccurately reflect that functionality changes have been made to a library. However, our intention with this metric was to inform about any recent activity from the development team of each library, which we believe is still reflected by any kind of change made within the repository.

Last Discussed on Stack Overflow Similar to our previous threat, we do not analyze the contents of the latest questions asked in Stack Overflow containing a library tag, which may result in questions whose information is not relevant to the library. A possible future improvement to this metric could be to analyze whether tagged libraries are one of the topics of discussion in the questions.

7.2 Internal Validity

Metrics The list of metrics used in the survey is not comprehensive for the purposes of library comparisons, as Section 5.7 suggests, and there is room for improvement in terms of our extraction and presentation methods. However,

the goal of our survey was to assess the general usefulness of metric-based comparisons and to gather information and feedback about the metrics that are most important to developers, with the objective of creating a publicly available implementation in the future. Therefore, our survey findings are not affected by potential small inaccuracies of the metrics we employed, especially in a comparison context.

Implementation of scripts To mitigate any potential bugs in the scripts we use to extract data for our metrics, we manually verified various samples of the results. Additionally, we make our scripts publicly available for others to verify or replicate our work.

Training Dataset for Classification To create a dataset of bug reports related to performance and security problems, we manually classify bug reports based on their title. This could have an impact on the predictions of the classifier. To mitigate this threat, we agreed on the manual classifications of bug reports. Additionally, we use the titles of bug reports provided by other researchers [33]. Performance and Security are difficult metrics to accurately quantify and we plan to investigate alternative methods.

Security and Performance Keywords We rely on a set of manually-obtained keywords to filter bug reports related to performance or security problems. However, since we manually obtained our set of keywords, it is by no means complete, which implies that our methodology might miss bug reports related to these non-functional requirements. To mitigate this issue, we used the same set of keywords for all libraries, which at least guarantees an equal treatment to all subject systems.

Participants and Library Domains As part of our survey, we presented participants with a list of library domains to evaluate not necessarily related to their domains of interest. Similarly, participants who evaluated multiple domains, may have provided higher ratings to metrics due to the familiarity

with metrics obtained in previous evaluations. This could have impacted the findings in the survey responses. To mitigate this threat and generate interest among participants, we only included popular domains in our subject systems, and we made the evaluation of multiple domains optional.

7.3 External Validity

Survey Results The findings presented in this work are based only on our sample set of participants and may not generalize beyond this context. However, to reduce possible opinion biases, we recruited participants with varying backgrounds and through different sources.

Sample of Survey Participants While our objective was to obtain a balanced set of survey participants from each of the different recruitment strategies, not all participants whom we invited took part in our survey, which means that the results are biased toward the largest group of developers, which were recruited from the snowball sampling strategy. To mitigate this issue, we present part of our results by also specifying the background of participants.

List of Libraries Our survey website did not provide an exhaustive list of libraries per domain as not all libraries had publicly available repositories and issue tracking systems. This may affect our findings and conclusions, but we believe this effect would be confined to the library choice in Q_{D1}, which was not explicitly used in our data analysis.

7.4 Limitations

The presented work focuses only on Java libraries and, therefore, some of our code to extract metrics is language-specific. However, the proposed methodologies could be applied to libraries in any programming language. Similarly, our target systems were only open source libraries as their repositories and issue tracking systems are freely available.

For our implementation of our extraction methodologies, we query the Github API to gather metric data. Nonetheless, the Github API is susceptible to query limits which can hinder the time that it takes to collect this information. To avoid query limits, offline Github dataset mirrors such as GHTorrent [11] could be used as a replacement for directly querying the Github API.

Chapter 8

Conclusions and Future Work

This thesis introduced metric-based comparisons of libraries, a technique that aims to assist software developers in making informed decisions when selecting appropriate libraries for their projects. Motivated by examples of developers who need to choose a library from a specific domain and who also care about certain aspects of libraries, we presented our approach, whose purpose is to work similarly to how products can be compared when shopping online. In our proposed methodology, metric data is extracted from a variety of sources, namely software repositories, issue tracking systems, and Q&A websites, with the intention of being used as comparison points for users needing libraries from a given domain.

By implementing an initial version of a metric-based comparison of Java libraries and conducting a survey to software developers from distinct backgrounds, we evaluated the usefulness of this technique for selecting library purposes. Additionally, we gained insight from participants on the metrics that matter the most when selecting libraries, and the desired metrics that they would like to see in future implementations. With this information, we created a website whose objective is to serve as a continuous comparison and surveying tool which will allow to further know the information that influence developers the most when choosing software libraries.

8.1 Future Work

There are several enhancements that could be made to the present work. As mentioned in the discussion of Chapter 5, additions could be made to the current set of metrics. Automatable metrics related to the quality and availability of documentation of libraries need to be investigated. Similarly, usability metrics which can be automatically extracted represents another challenge that needs to be researched. Another highly requested addition was the functionality information of libraries. This information could be added in the form of tasks described in natural language extracted from Q&A websites such as Stack Overflow, similar to what has been done in past research [40]. While other work has measured documentation and usability with the combination of machine learning classifiers and sentiment analysis [49], the application of current sentiment analysis tools on Software Engineering artifacts is yet to produce reliable results even with customized datasets [22].

Additionally, there is room to improve the current set of employed metrics. The classifiers used to detect performance and security bug reports could be further improved with customized datasets based on library domains. Bug report information such as their severity could be added to obtain a closer look at how severe issues are treated by the respective developer communities.

Furthermore, long-term studies could be conducted to observe the usefulness of metric-based comparisons in the selection of libraries and the successful integration of chosen libraries in software projects.

References

[1] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 207–216, ISBN: 978-1-4673-2936-1. 45

[2] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 298–308, ISBN: 978-1-4244-3453-4. 54

[3] H. Borges, A. C. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of Github repositories," *CoRR*, vol. abs/1606.04984, 2016. arXiv: 1606.04984. 7

[4] J. Corbin and A. Strauss, *Basics of qualitative research. techniques and procedures for developing grounded theory*, 3rd. 2008. 36

[5] D. A. da Costa, S. McIntosh, U. Kulesza, and A. E. Hassan, "The impact of switching to a rapid release cycle on integration delay of addressed issues: an empirical study of the Mozilla Firefox project," in *Proc. of the International Conference on Mining Software Repositories (MSR)*, 2016, pp. 374–385. 11

[6] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *35th International Conference on Software Engineering*, ser. ICSE'13, San Francisco, CA, May 2013, pp. 422–431. 13

[7] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10, Cape Town, South Africa: ACM, 2010, pp. 52–56, ISBN: 978-1-60558-974-9. 11

[8] (2018). Git tags, [Online]. Available: https://git-scm.com/book/en/v2/Git-Basics-Tagging. 17

[9] R. L. Glass, "Frequently forgotten fundamental facts about software engineering," *IEEE Software*, vol. 18, no. 3, pp. 112–111, May 2001, ISSN: 0740-7459. DOI: 10.1109/MS.2001.922739. 1

[10] L. A. Goodman, "Snowball sampling," *Ann. Math. Statist.*, vol. 32, no. 1, pp. 148–170, Mar. 1961. DOI: 10.1214/aoms/1177705148. 29

[11] G. Gousios, "The ghtorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 233–236, ISBN: 978-1-4673-2936-1. 13, 58

[12] R. J. Grissom and J. J. Kim, *Effect sizes for research : A broad practical approach*, English. Mahwah, N.J. ; London : Lawrence Erlbaum Associates, 2005, Formerly CIP, ISBN: 0805850147 (alk. paper). 30

[13] E. Guzman, D. Azócar, and Y. Li, "Sentiment analysis of commit comments in Github: An empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, Hyderabad, India: ACM, 2014, pp. 352–355, ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597118. 10

[14] A. Hora and M. T. Valente, "Apiwave: Keeping track of API popularity and migration," in *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution*, ser. ICSME '15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 321–323, ISBN: 978-1-4673-7532-0. 6, 13

[15] http://boa.cs.iastate.edu/, Accessed on year 2017. 16

[16] https://mvnrepository.com/, Accessed on year 2017. 25

[17] N. Juristo and A. M. Moreno, *Basics of software engineering experimentation*, 1st. Springer Publishing Company, Incorporated, 2010, ISBN: 1441950117, 9781441950116. 53

[18] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: A case study for the Apache Software Foundation projects," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, Austin, Texas: ACM, 2016, pp. 154–164, ISBN: 978-1-4503-4186-8. 1, 8

[19] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "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*, ser. MSR '12, Zurich, Switzerland: IEEE Press, 2012, pp. 179–188, ISBN: 978-1-4673-1761-0. 11

[20] A. Lamkanfi and S. Demeyer, "Filtering bug reports for fix-time analysis," in *2012 16th European Conference on Software Maintenance and Reengineering*, Mar. 2012, pp. 379–384. 11

[21] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 1–10. 9, 20

[22] B. Lin, F. Zampetti, M. Di Penta, G. Bavota, M. Lanza, and R. Oliveto, "Sentiment analysis for software engineering: How far can we go?" In *Proceedings of the 40th International Conference on Software Engineering*, ACM, 2018, pp. 94–104. 60

[23] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "API change and fault proneness: A threat to the success of Android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia: ACM, 2013, pp. 477–487, ISBN: 978-1-4503-2237-9. 2, 8, 20, 30

[24] W. Maalej and M. P. Robillard, "Patterns of knowledge in API reference documentation," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013. 41

[25] A. Mauczka, F. Brosch, C. Schanes, and T. Grechenig, "Dataset of developer-labeled commit messages," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15, Florence, Italy: IEEE Press, 2015, pp. 490–493, ISBN: 978-0-7695-5594-2. 12

[26] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "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*, ser. IWPSE-Evol '09, Amsterdam, The Netherlands: ACM, 2009, pp. 57–62, ISBN: 978-1-60558-678-6. 6, 7, 9

[27] Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining API popularity," in *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques*, ser. TAIC PART'10, Windsor, UK: Springer-Verlag, 2010, pp. 173–180, ISBN: 3-642-15584-7, 978-3-642-15584-0. 6, 7

[28] *Mining API aspects in API reviews*, `http://swat.polymtl.ca/data/opinionvalue-technical-report.pdf`, Accessed: 2017-08-01. 9

[29] F. L. de la Mora, *Library Comparison Website*, `http://smr.cs.ualberta.ca/comparelibraries/`, 2018. 44

[30] F. L. de la Mora, *Code Repository*, `https://github.com/ualberta-smr/LibraryMetricScripts`, 2018. 44

[31] F. L. de la Mora and S. Nadi, "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 Track (ICSE NIER '18)*, 2018. iii

[32] S. Mostafa, R. Rodriguez, and X. Wang, "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*, ser. ISSTA 2017, Santa Barbara, CA, USA: ACM, 2017, pp. 215–225, ISBN: 978-1-4503-5076-1. 2, 12, 20

[33] M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto, "A dataset of high impact bugs: Manually-classified issue reports," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15, Florence, Italy: IEEE Press, 2015, pp. 518–521, ISBN: 978-0-7695-5594-2. 8, 22, 56

[34] M. Ortu, B. Adams, G. Destefanis, P. Tourani, M. Marchesi, and R. Tonelli, "Are bullies more productive?: Empirical study of affectiveness vs. issue fixing time," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15, Florence, Italy: IEEE Press, 2015, pp. 303–313, ISBN: 978-0-7695-5594-2. 2, 11, 20

[35] M. Ortu, A. Murgia, G. Destefanis, P. Tourani, R. Tonelli, M. Marchesi, and B. Adams, "The emotional side of software developers in JIRA," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, Austin, Texas: ACM, 2016, pp. 480–483, ISBN: 978-1-4503-4186-8. 10

[36] N. Pandey, A. Hudait, D. K. Sanyal, and A. Sen, "Automated classification of issue reports from a software issue tracker," in *Progress in Intelligent Computing Techniques: Theory, Practice, and Applications: Proceedings of ICACNI 2016, Volume 1*, P. K. Sa, M. N. Sahoo, M. Murugappan, Y. Wu, and B. Majhi, Eds. Singapore: Springer Singapore, 2018, pp. 423–430, ISBN: 978-981-10-3373-5. 20

[37] L. D. Panjer, "Predicting Eclipse bug lifetimes," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, May 2007, pp. 29–29. 11

[38] C. Parnin and C. Treude, "Measuring API documentation on the web," in *Proceedings of the 2Nd International Workshop on Web 2.0 for Software Engineering*, ser. Web2SE '11, Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 25–30, ISBN: 978-1-4503-0595-2. 41

[39] D. Pletea, B. Vasilescu, and A. Serebrenik, "Security and emotion: Sentiment analysis of security discussions on Github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, Hyderabad, India: ACM, 2014, pp. 348–351, ISBN: 978-1-4503-2863-0. 10

[40] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic API recommendation using crowdsourced knowledge," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, Mar. 2016, pp. 349–359. DOI: 10.1109/SANER.2016.80. 5, 60

[41] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, May 2013, ISSN: 0098-5589. DOI: 10.1109/TSE.2012.63. 42

[42] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, *et al.*, "On-demand developer documentation," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2017, pp. 479–483. 40

[43] A. A. Sawant and A. Bacchelli, "A dataset for API usage," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15, Florence, Italy: IEEE Press, 2015, pp. 506–509, ISBN: 978-0-7695-5594-2. 7

[44] V. Sinha, A. Lazar, and B. Sharif, "Analyzing developer sentiment in commit logs," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16, Austin, Texas: ACM, 2016, pp. 520–523, ISBN: 978-1-4503-4186-8. 10

[45] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proceedings of the 36th International Conference on Software Engineering*, ACM, 2014, pp. 643–652. 41

[46] C. Teyton, J. R. Falleri, and X. Blanc, "Mining library migration graphs," in *2012 19th Working Conference on Reverse Engineering*, Oct. 2012, pp. 289–298. 6

[47] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A study of library migrations in Java," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 1030–1052, Nov. 2014, ISSN: 2047-7473. 2, 8

[48] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, Oct. 2013, pp. 182–191. DOI: 10.1109/WCRE.2013.6671293. 6

[49] G. Uddin and F. Khomh, "Automatic summarization of API reviews," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '17, 2017. 2, 5, 20, 42, 60

[50] C. Vendome, M. Linares-Vásquez, G. Bavota, M. D. Penta, D. German, and D. Poshyvanyk, "Machine learning-based detection of open source license exceptions," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 118–129. DOI: 10.1109/ICSE.2017.19. 42

[51] J. Visser, A. van Deursen, and S. Raemaekers, "Measuring software library stability through historical version analysis," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ser. ICSM '12, Washington, DC, USA: IEEE Computer Society, 2012, pp. 378–387, ISBN: 978-1-4673-2313-0. 9

[52] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of API breaking changes: A large scale study," in *24th International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER '17, 2017, pp. 138–147. 2, 12, 19

Appendix A

Plots and Survey Recruitment Materials

This Appendix contains the recruitment materials that were sent to potential participants of the developer survey conducted in this work, as explained in Chapter 4, and the plots for metrics as specified in Chapter 5 of this thesis.

Appendix A.1 has the e-mail template used to contact Github users, Appendix A.2 contains the template for Stack Overflow users for the same purpose, and the general e-mail sent to university students can be found in Appendix A.3. Appendix A.4 contains plots of the usefulness ratings for the Backwards Compatibility, Issue Closing Time, and Issue Response Time, Popularity, Last Discussed on Stack Overflow, and Last Modification Date metrics by domain.

A.1 Recruitment Template E-mail for Github Users

Dear [Github username],

We are a group of researchers from the Department of Computing Science at the University of Alberta, Canada, who work on developing tools and methodologies to help software developers use Java libraries more easily and correctly. We have developed a new technique for comparing Java libraries from the same domain. We noticed that you have committed to a source file that uses the [library name] library from the [domain name] domain in the Github project [repository], which is one of the libraries included in our work. Accordingly, we would like to invite you to participate in a short survey about our comparison technique. The survey can be found in this [URL]. We would appreciate it if you can fill out the survey as soon as possible, but note that it will remain open until January 26. The survey should take no more than 5-7 minutes of your time. Note that the survey is completely anonymous. We only record the fact that you have used one of the libraries in our data, but do not record anything about your identity or activities. More information about how the data we collect is used can be found on the information page of the survey.

Thank you for your time.

Fernando Lopez de La Mora (email: lopezdel@ualberta.ca)
Sarah Nadi (email: nadi@ualberta.ca, website: http://www.sarahnadi.org)
The plan for this study has been reviewed for its adherence to ethical guidelines by a Research Ethics Board at the University of Alberta. For questions regarding participant rights and ethical conduct of research, contact the Research Ethics Office at (+1)-(780)-492-2615.

A.2 Recruitment Template E-mail for Stack Overflow Users

Dear [Stack Overflow username],

We are a group of researchers from the Department of Computing Science at the University of Alberta, Canada, who work on developing tools and methodologies to help software developers use Java libraries more easily and correctly. We have developed a new technique for comparing Java libraries from the same domain. We noticed that you have been involved in a Stack Overflow post regarding Java testing libraries (question #[number of question]), which is one of the domains we consider in our work. Accordingly, we would like to invite you to participate in a short survey about our comparison technique. The survey can be found in this link: [URL] and will be open until January 26th. The survey should not take more than 5-7 minutes of your time. Note that the survey is completely anonymous. We only record the fact that you have used been involved in a StackOverflow discussion of the libraries in our data, but do not record anything about your identity or activities. More information about how the data we collect is used can be found on the information page of the survey.

Thank you for your time.

Fernando Lopez de La Mora (email: lopezdel@ualberta.ca)
Sarah Nadi (email: nadi@ualberta.ca, website: http://www.sarahnadi.org)
The plan for this study has been reviewed for its adherence to ethical guidelines by a Research Ethics Board at the University of Alberta. For questions regarding participant rights and ethical conduct of research, contact the Research Ethics Office at (+1)-(780)-492-2615.

A.3 Recruitment Template E-mail for University Students

Hello,

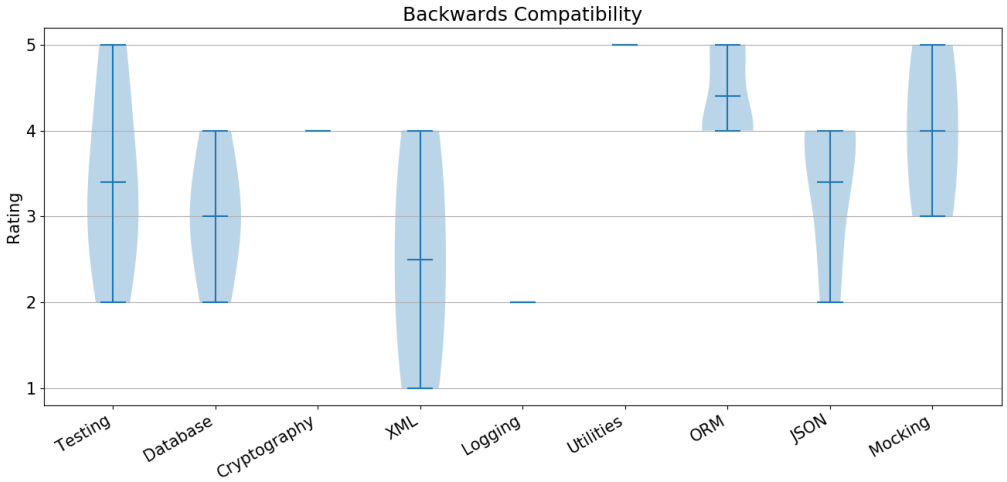
Do you have experience using Java? If so, we would like to invite you to support our research by participating in one short survey involving techniques to help developers use Java libraries more easily.

The survey can be found at [URL] and it is open until January 26th and should not take more than 5-7 minutes of your time.

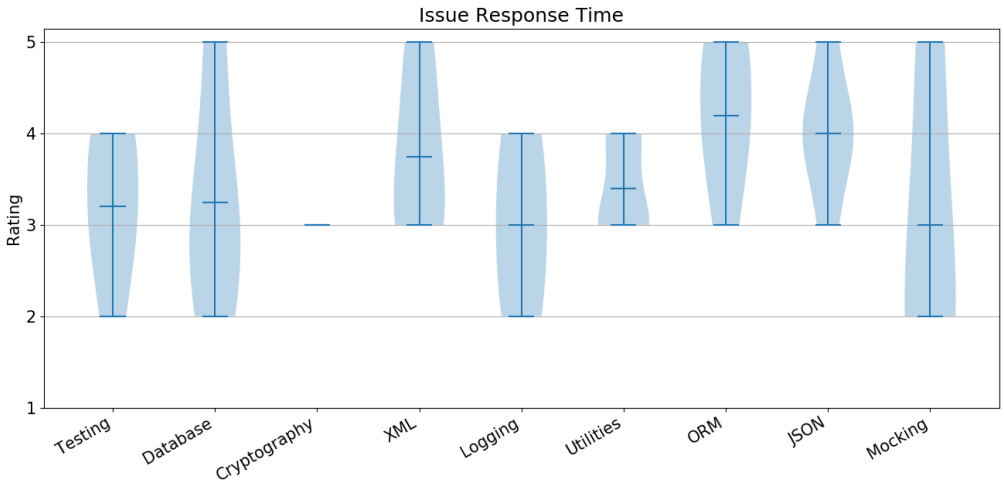
Thank you for your time.

Fernando Lopez de La Mora (e-mail: lopezdel@ualberta.ca)
Sarah Nadi (email: nadi@ualberta.ca, website: http://www.sarahnadi.org)
The plan for this study has been reviewed for its adherence to ethical guidelines by a Research Ethics Board at the University of Alberta. For questions regarding participant rights and ethical conduct of research, contact the Research Ethics Office at (+1)-(780)-492-2615.

A.4 Plots


Backwards Compatibility
Rating
5
4
3
2
1
Testing
Database
Cryptography
XML
Logging
Utilities
ORM
JSON
Mocking


Issue Closing Time
Rating
5
4
3
2
1
Testing
Database
Cryptography
XML
Logging
Utilities
ORM
JSON
Mocking


Issue Response Time
Rating
5
4
3
2
1
Testing
Database
Cryptography
XML
Logging
Utilities
ORM
JSON
Mocking

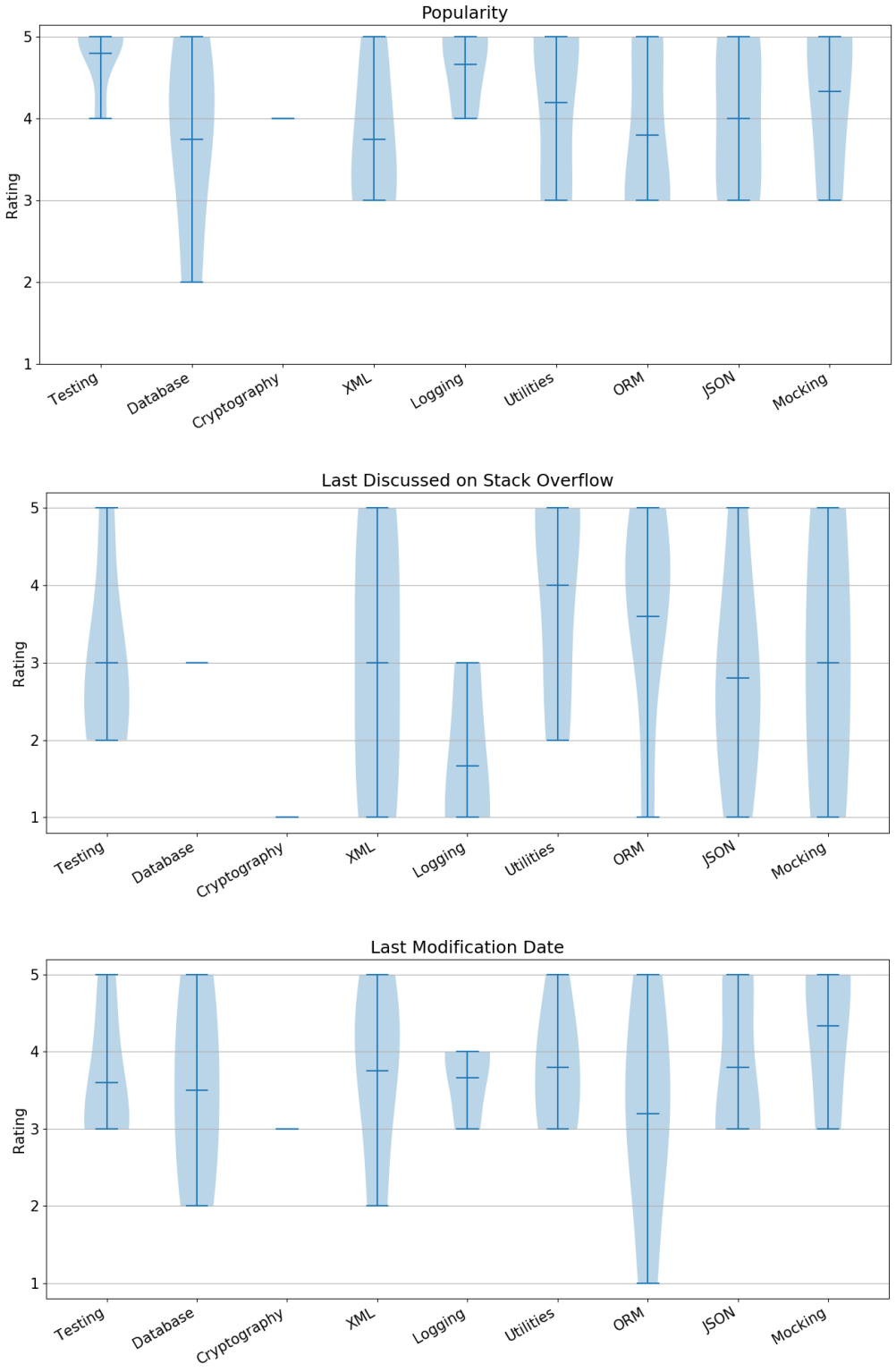


Figure A.1: Usefulness ratings for the Backwards Compatibility, Issue Closing Time, and Issue Response Time, Popularity, Last Discussed on Stack Overflow, and Last Modification Date metrics by domain