Analyzing the Evolution of Testing Library Usage in Open Source Java Projects

Ahmed Zerouali
Software Engineering Lab
University of Mons, Belgium
Email: ahmed.zerouali@umons.ac.be

Tom Mens
Software Engineering Lab
University of Mons, Belgium
Email: tom.mens@umons.ac.be

***Abstract*—Software development projects frequently rely on testing-related libraries to test the functionality of the software product automatically and efficiently. Many such libraries are available for *Java*, and developers face a hard time deciding which libraries are most appropriate for their project, or when to migrate to a competing library. We empirically analysed the usage of eight testing-related libraries in 4,532 open source *Java* projects hosted on *GitHub*. We studied how frequently specific (pairs of) libraries are used over time. We also identified if and when library usages are replaced by competing ones during a project's lifetime. We found that some libraries are considerably more popular than their competitors, while some libraries become more popular over time. We observed that many projects tend to use multiple libraries together. We also observed permanent and temporary migrations between competing libraries. These findings may pave the way for recommendation tools that allow project developers to choose the most appropriate library for their needs, and to be informed of better alternatives.**

***Index Terms*—Software Evolution; Library Usage; Library Migration; Empirical Analysis; Java; Open Source; Testing**

I. Introduction

Software systems are commonly implemented on top of frameworks and rely on external libraries to reuse complex functionality and increase productivity [1]. Such software libraries typically come with a well-documented API.

In order to decide whether or not to use a particular library, software developers can rely on how the library has been used in other projects, how the library evolves over time, and when and why other projects have upgraded to a new library version or migrated to a competing library. To aid in such decisions, historical evidence of library usage in a large corpus of software projects is needed, which is the aim of the current paper.

According to a blog post [2], testing libraries (including, *JUnit*, *TestNG* and the *Spring* testing library), matching libraries (*Hamcrest* and the more recent *AssertJ*) and to a lesser extent mocking libraries (e.g., *Mockito*, *EasyMock* and their *PowerMock* extension) are among the most popular Java libraries on *GitHub*.

Because of the importance of testing in software development, and because of the availability of multiple competing libraries, it is relevant to study the evolution of how these libraries are used, as well as migrations from using one library to another one. This is the focus of our empirical study, in which we analyse the evolution of the usage of the eight aforementioned testing-related libraries in 4,532 open source *Java* projects hosted on *GitHub*.

II. Related Work

Many researchers analysed the evolution of library and API usage. Teyton et al. [3], [4] determined sets of similar libraries, including testing-related ones, in a large corpus of software projects. The results can be used for suggesting alternative libraries to developers. They analysed how and why library migrations occur, and found migrations to be relatively rare, with few projects being subject to more than one migration. Our research provides more specific details about testing-related library usage.

Lämmel et al. [5] carried out AST-based API usage analysis on a large corpus of open source *Java* projects. They studied the usage of a list of APIs extracted from built projects, reference projects and unbuilt projects. De Roover et al. [6] explored library popularity in terms of source-level usage patterns.

Suhas et al. [7] manually analysed the *Jira* issues to identify logging library migrations within Apache Software Foundation projects. They found that 33 out of 223 projects underwent at least one logging library migration and that flexibility, performance improvement, code maintenance, functionality and dependency are the main drivers for logging library migration.

Businge et al. [8] investigated the reasons behind the internal interfaces used by developers. They detected cases where developers do not read documentation or guidelines, and where they use internal interfaces to benefit from advanced functionalities.

Goeminne et al. [9], [10] empirically analysed the evolution of *Java* database access libraries. This paper follows a similar approach for testing-related *Java* libraries.

III. Methodology

We studied open source projects extracted from *GitHub*. We selected *Java* projects because *Java* is the most popular programming language according to the TIOBE index[1]. We selected *GitHub* as a data source because we need full access to the source code history in order to carry out our analysis, and because it's the largest host of *Java* source code (containing over 900K *Java* projects).

[1]http://www.tiobe.com/tiobe-index/ (November 2016)

We decided to study the most popular testing libraries based on their number of usages in the *Maven* Central Repository[2]: *JUnit*, *TestNG*, *Spring* test, *Arquillian* and Spock framework. We added the *Hamcrest* and *AssertJ* libraries that are often used as matcher frameworks to facilitate writing more complex tests. We also considered the most used mocking libraries in *Maven*[3]: *Mockito*, *EasyMock*, *PowerMock*, *JMock* and *JMockit*. Libraries for non-functional testing (such as UI testing, performance testing, acceptance testing, load testing, etc.) were excluded, but could be considered very easily in a follow-up study.

We began with the *GitHub Java* Corpus of 14,807 open source *Java* projects extracted by Allamanis and Sutton [11] and obtained from Github Archive[4]. We removed 1,748 projects that are no longer available on *GitHub*. Based on popularity measured by number of stars, we added 7,629 popular *GitHub* projects satisfying the same filters as those applied by the *GitHub Java* Corpus. More specifically, we ignored projects that were never forked or that were forks of other projects. This filtering reduces the risk of obtaining results statistically biased by groups of strongly related individuals in the considered project population.

This led us to 20,688 projects as potential candidates for our empirical analysis. We created a local clone of the *GitHub* repository for each of them. We further restricted ourselves to those projects relying on *Maven*, in order to be able to identify project dependencies defined in Project Object Model files (pom.xml). We also restricted ourselves to projects with an active lifetime of at least two years, which seems an acceptable minimal duration for detecting potential migrations of library usage during the projects' lifetimes. This lead us to 6,424 remaining projects.

Each month, we analysed the first snapshot of each considered project, by looking at the import statements in each *Java* file of the project. We found 4,532 *Java* projects that used at least one of the considered *Java* libraries. For these projects, we analysed in total 125,580 commits, 10,033,726 *Java* source code files and 31,264,586 import statements for the considered libraries.

IV. Empirical Evaluation

The research questions in this paper are similar to the ones in [10], [12]. Our focus is on the evolution of testing-related *Java* libraries. We present each question and its results by means of tables, visualizations and statistical tests in the following subsections.

RQ_1 Which Are the Most Frequently Used Testing-Related Libraries?

Figure 1 shows the relative usage frequency of each of the 4,532 *Java* projects in our corpus that used at least one of the considered libraries at least once during its lifetime (i.e., at least one *Java* file in at least one commit imported that library). We observe a high imbalance. *JUnit* was by far the most popular: off all considered libraries, projects are very likely (97%) to use *JUnit*. In comparison, the competing *TestNG* library is used in only 11.9% of the projects. Of the considered mocking libraries, *Mockito* was by far the most used library, being used by 31.3% of all projects. *EasyMock* and *PowerMock* are considerably less popular, corresponding to 9.3% and 4.7% of all projects, respectively. The matching library *Hamcrest* (22.5%) is considerably more popular than its competitor *AssertJ*, which can be explained by the fact that *AssertJ* is much more recent. However, since its appearance in March 2013 its popularity is increasing.

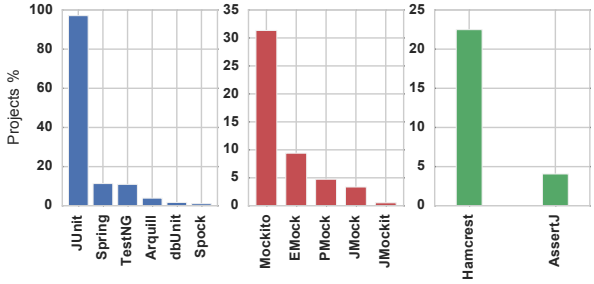

Fig. 1. Percentage of projects in which a given library is used at least once during its lifetime. Testing libraries are shown in blue, matcher libraries in green, and mocking libraries in red.

We decided to focus on those eight libraries that are used by a sufficient number of projects in our corpus. Therefore, we excluded the 5 least frequent libraries in Figure 1: *Arquillian* and *Spock*, two testing frameworks for writing integration tests; *dbUnit*, a unit testing framework for database-driven projects; and two less popular mocking libraries *JMock* and *JMockit*.

RQ_2 When Are Libraries Introduced in a Project's Lifetime?

For each project, we analysed after how long each used library got introduced (i.e., the time interval between the first project commit and the first commit where at least one *Java* project file imported the considered library). Figure 2 shows that the considered libraries are introduced early, regardless of their purpose. 56% of all projects started using at least one of these libraries as early as their first commit on *GitHub*. This could be explained by the fact that these projects were already in development before coming to *GitHub*, or because they follow a test-driven development process, implying that tests are introduced very early in the project's lifetime.

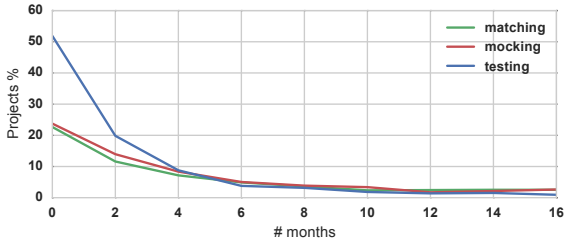

Fig. 2. Number of months after the first commit before introduction of one of the testing-related libraries.

[2]https://mvnrepository.com/open-source/testing-frameworks
[3]https://mvnrepository.com/open-source/mocking
[4]https://www.githubarchive.org/

Unsurprisingly, we observed that *JUnit* and *TestNG* are the first libraries to be introduced in respectively 88% and 57% of the projects in which they occur with other libraries. *AssertJ* was never found to be introduced first, probably because it is a much more recent library.

Table I compares the relative order of introduction of the different library types. As expected, testing libraries tend to be introduced before matching (71.6%) and before mocking (57.9%) libraries. In 41.4% cases, mocking libraries are introduced together with testing libraries, and in 50.3% cases they are introduced before matching libraries.

TABLE I
INTRODUCTION ORDER OF TESTING-RELATED LIBRARY TYPES

A→B	#projects	A before B	A after B	A = B
testing→matching	983	**71.6%**	1.0%	27.4%
testing→mocking	1443	**57.9%**	0.7%	41.4%
mocking→matching	523	**50.3%**	21.8%	27.9%

RQ_3 *Which Libraries Are Used Over a Project's Lifetime?*

We analysed if projects use different libraries over their lifetime. The results are shown using Venn diagrams in Figure 3. *JUnit* occurs as the *only* testing library in 61.3% of all projects (2,777 in total) and 97% of all considered projects have used *JUnit* at least once. *TestNG* is used as the *only* testing library much less frequently, in 2.3% (106) of all projects using at least one of the considered libraries, while 11% of all projects have used *TestNG* at least once in their lifetime. All projects that used either *Hamcrest*, *Spring* or *AssertJ* also used at least one other library during their lifetime. In the overwhelming majority of the cases, *Hamcrest* and *AssertJ* are used in projects that have used *JUnit* in their lifetime.

RQ_4 *Which Libraries Are Used Simultaneously in Projects?*

Of all projects that used at least two of the considered libraries during their lifetime, we computed which pairs of libraries were actually being used *simultaneously* in a specific project snapshot (not necessarily within the same *Java* files). Table II shows the percentage of projects using a library A that also used library B simultaneously at least once during their lifetime.

Nearly all projects that use *Hamcrest*, *AssertJ*, *Spring*, *Mockito* or *PowerMock* also use *JUnit* (values >94%). Those that don't, tend to use *TestNG* instead. Unsurprisingly, *JUnit* is used much less frequently with its competitor *TestNG*: projects that use *JUnit* rarely use *TestNG* (7%). However, more than 3 out of 5 projects that use *TestNG* also used *JUnit* simultaneously at some point (64%). This may indicate that projects using *TestNG* actually migrated away from *JUnit* somewhere during their lifetime. RQ_6 studies this library migration phenomenon in more detail.

Projects using *Hamcrest* rarely use *AssertJ* (7.36%). The other way round, more than 2 out of 5 projects that use *AssertJ* also use *Hamcrest* simultaneously (40.8%). This may be a sign that many projects that use *Hamcrest* are in the process of migrating to *AssertJ*.

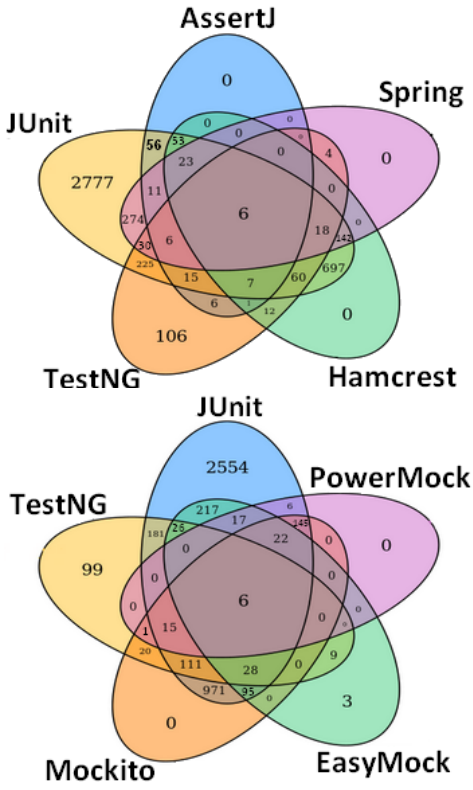

Fig. 3. Number of projects using different testing-related libraries at least once during their lifetime (not necessarily simultaneously).

For the mocking libraries, *PowerMock* is mostly used as extension of *Mockito* (86.5%), and much less as an extension of *EasyMock* (19.1%). As expected, projects that use *Mockito* rarely use *EasyMock* (8.23%). However, more than one out of four projects that use *EasyMock* also use *Mockito* (27.5%). This is relatively a high percentage if we consider that *EasyMock* and *Mockito* are competitors.

TABLE II
HEATMAP SHOWING THE PERCENTAGE OF PROJECTS USING LIBRARY A (LINES) THAT ALSO USE LIBRARY B (COLUMNS) SIMULTANEOUSLY AT LEAST ONCE DURING THEIR LIFETIME. (LIGHTER BACKGROUNDS CORRESPOND TO HIGHER PROPORTIONS.)

	JUnit	TestNG	Spring	Hamcrest	AssertJ	Mockito	EasyMock	PowerMock
JUnit	*100%*	7%	12%	23%	4%	32%	9%	5%
TestNG	64%	*100%*	13%	18%	6%	33%	12%	4%
Spring	99%	12%	100%	37%	8%	53%	18%	10%
Hamcrest	99%	9%	18%	100%	7%	59%	10%	10%
AssertJ	95%	16%	23%	41%	100%	61%	11%	15%
Mockito	98%	11%	19%	42%	8%	100%	8%	13%
EasyMock	77%	14%	22%	24%	5%	28%	100%	10%
PowerMock	99%	9%	23%	47%	13%	87%	19%	100%

RQ_5: *How Frequently Are Libraries Used Over Time?*

Figure 4 (top) shows the monthly evolution over time of the usage of testing and matching libraries. The proportion

of projects using *JUnit* is decreasing (but remains very high), while *TestNG* and *Spring* have a stable (but low) proportion of projects using them. The proportional usage of *Hamcrest* and *AssertJ* is increasing over time.

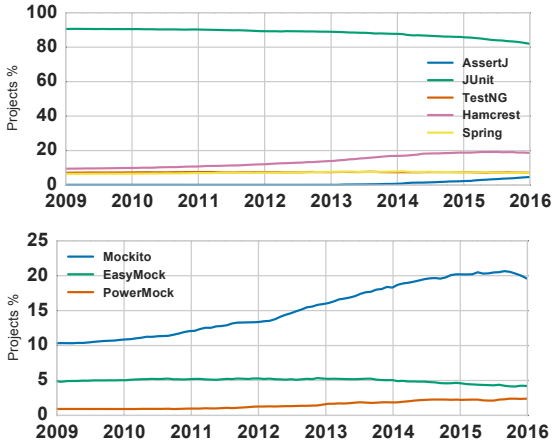

Fig. 4. Monthly evolution of the (proportion of) *Java* projects using testing-related libraries.

For mocking libraries, Figure 4 (bottom) shows that the proportional usage of *Mockito* and *PowerMock* has remarkably increased, whereas the usage of *EasyMock* is slightly declining over time.

RQ_6 *How Frequently do Libraries Co-occur at File Level?*

For projects using certain pairs of libraries simultaneously, we explored if these libraries are also used together within the same *Java* files belonging to the project. For the mocking libraries we would expect *Mockito* and *EasyMock* to be rarely used together, since they are competing libraries. For *PowerMock* on the other hand, we would expect it to be used frequently in the same files where *Mockito* and *EasyMock* are used, given that *PowerMock* is an extension for these two libraries.[5]

This intuition is confirmed by Figure 5 (top) that, for each pair of libraries, shows a violin plot with the distribution across projects of the ratio between the number of files that relate to each or both libraries, and the total number of files that relate to any of them. Because our data set is not normally distributed, we used the nonparametric Kolmogorov-Smirnov test to verify, for each library pair, if the distributions are similar (null hypothesis H_0) or different (alternative hypothesis H_1). H_0 was rejected only for the pairs (*Mockito*,*PowerMock*) and (*EasyMock*,*PowerMock*) with statistical significance ($p < 0.001$ and $p < 0.002$, respectively).

Since *JUnit* and *TestNG* are two competing testing libraries, we do not expect *Java* project files to use both libraries simultaneously. The same holds for *Hamcrest* and *AssertJ* for the same reasons. This intuition is again confirmed by Figure 5 (bottom). The Kolmogorov-Smirnov test for these pairs of libraries did not allow us to reject H_0 with statistical significance ($p > 0.01$).

[5]https://github.com/jayway/powermock/wiki/GettingStarted

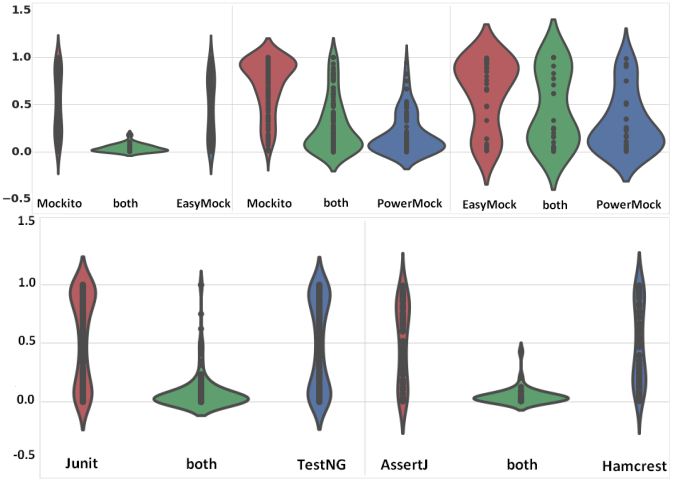

Fig. 5. Proportional distribution of *Java* files (in all projects) relating to pairs of testing-related libraries.

RQ_7 *Do Projects Migrate to Competing Libraries?*

We analysed all considered libraries used by each project's *Java* files once every month, to determine if a project p permanently switches from library l_1 to library l_2 during its observed lifetime. We define a ***permanent*** *library migration* from l_1 to l_2 in p, if $\exists$ time t_1 where project p uses library l_1 and does not use l_2, while $\exists\ t_2 > t_1$ such that $\forall\ t \geq t_2$ project p uses l_2 but does not use l_1.

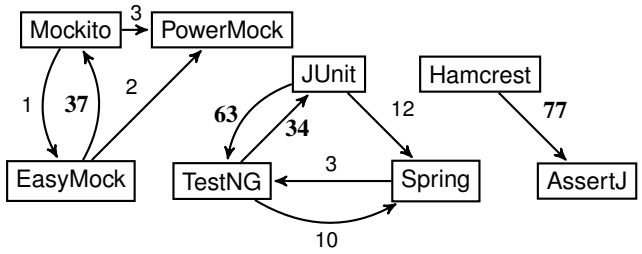

Fig. 6. Number of migrations observed between testing libraries.

The migration graph of Figure 6 visualises all permanent migrations for the considered libraries. We observed a high number of permanent migrations (63) from *JUnit* to *TestNG*, while only 34 projects permanently migrated from *TestNG* to *JUnit*. 50 out of these 97 projects didn't involve a transition phase (during which both libraries are used simultaneously) for the migration.

We observed the highest number of permanent migrations from *Hamcrest* to *AssertJ* (77) even if these two libraries were used together in only 90 (i.e., 1.98%) of all considered projects. No permanent migrations were observed from *AssertJ* to *Hamcrest*, indicating the increasing use of *AssertJ* as a competing library. For the mocking libraries, we observe most migrations (37) from *EasyMock* to *Mockito*, most likely because it offers more functionality.

We also found cases (not shown in the graph) of ***temporary*** *library migration*. Nine projects temporarily migrated from

JUnit to *TestNG* and returned to *JUnit* after some time. Four other projects performed the opposite temporary migration. Four projects migrated from *Hamcrest* to *AssertJ* and returned to using *Hamcrest*.

V. Threats to Validity

Our research suffers from the same threats as other research relying on *GitHub* [13]. Our results may not be generalisable to non-*Java* projects or to closed-source industrial projects that are typically subject to more restricted development rules. While we studied the usage of eight testing-related *Java* libraries, the proposed methodology is applicable to other types of libraries as well. Our results may, however, be biased by the fact that we have excluded projects with a lifetime of less than two years, as well as projects that are no longer available in *GitHub*. In our approach we assume that a library is being used by a *Java* project if one of the project files contains specific import statements pertaining to that library. This approach may lead to false positives, since imported classes and interfaces are not necessarily used in the source code.

VI. Conclusion and Future Work

We studied the usage of eight popular testing, matching and mocking libraries in a large corpus of *GitHub*-hosted *Java* projects. We observed that many of these libraries are being used simultaneously, with *JUnit* being the most prominent testing library. Some libraries were found to complement or reinforce one another (e.g., *PowerMock* which extends either *Mockito* or *EasyMock*) while others are in competition (e.g., *JUnit* versus *TestNG*, *Mockito* versus *EasyMock*, *Hamcrest* versus *AssertJ*).

We found that 5% of the considered projects to be subject to library migrations, in which a project replaces one of its used libraries by another. This migration was mainly permanent. For example, projects tend to migrate from *EasyMock* to *Mockito* and from *Hamcrest* to *AssertJ*, but not the other way round. In a limited number of cases, the library migrations were temporary, with the opposite migration being observed later on in the project's lifetime. We aim to study *why* such "inverse" migrations happen.

Our findings about when, why and how projects perform library migrations is promising, but requires a more in-depth analysis. In future work we plan consider the effect of the specific library *version* on the migration phenomenon. Upgrading to a new major library release may imply significant functional changes, potentially leading to an increased migration towards (or away from) this particular version. We observed cases like this, e.g., the project livetribe-slp first used *JUnit* 3, then migrated to *TestNG*, and then returned to using *JUnit* 4.

We aim to use and extend our preliminary results to provide recommendation tools or dashboards that make it easier for project developers to choose the most appropriate library (version) for their needs, and to be informed of competing (versions of) libraries to migrate to, accompanied by a justification of why and when to perform this migration.

Building upon the work of De Roover et al. [6], we aim to conduct fine-grained analyses of how frequently each of a library's functions (as provided by its public API) are used, how this evolves over time, and whether certain combinations of functionalities of different libraries are frequently used together. Library developers can benefit from this analysis to better understand how their major library versions are being used in practice, in order to provide incentives to increase their library's adoption rate and to avoid its users to migrate to competing libraries.

We also plan to analyse the effort of migrating between different libraries, as well as the effort of upgrading to a new major version of a library. Based on this effort analysis, we aim to provide tools to reduce this effort and hence facilitate support for upgrading to a different library version or migrating to a different library.

Acknowledgment

This research is part of FRFC research projects T.0022.13 and J.0023.16 financed by F.R.S.-FNRS, Belgium. We thank Alexandre Decan, Eleni Constantinou and Maelick Claes for their useful feedback.

References

[1] S. Moser and O. Nierstrasz, "The effect of object-oriented frameworks on developer productivity," *Computer*, vol. 29, no. 9, 1996.

[2] A. Zhitnitsky, "We analyzed 60,678 libraries on github - here are the top 100," Apr. 2015. [Online]. Available: http://blog.takipi.com/we-analyzed-60678-libraries-on-github-here-are-the-top-100/

[3] C. Teyton, J. Falleri, and X. Blanc, "Mining library migration graphs," in *Working Conf. Reverse Engineering (WCRE)*, 2012, pp. 289–298.

[4] 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, 2014.

[5] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *ACM Symp. Applied Computing*. ACM , 2011, pp. 1317–1324.

[6] C. De Roover, R. Lämmel, and E. Pek, "Multi-dimensional exploration of API usage," in *Int'l Conf. Program Comprehension*, May 2013, pp. 152–161.

[7] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: A case study for the Apache software foundation projects," in *Int'l Conf. Mining Software Repositories (MSR)*. ACM, 2016, pp. 154–164.

[8] J. Businge, A. Serebrenik, and M. van den Brand, "Analyzing the eclipse API usage: Putting the developer in the loop," in *European Conf. Software Maintenance and Reengineering*, 2013, pp. 37–46.

[9] M. Goeminne and T. Mens, "Towards a survival analysis of database framework usage in Java projects," in *Int'l Conf. Software Maintenance and Evolution*, Sep. 2015, pp. 551–555.

[10] A. Decan, M. Goeminne, and T. Mens, "On the interaction of relational database access technologies in open source java projects," in *Postproceedings of SATToSE 2015 Seminar on Advanced Techniques and Tools for Software Evolution*, 2017. [Online]. Available: https://arxiv.org/abs/1701.00416

[11] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Working Conf. Mining Software Repositories*. IEEE, 2013, pp. 207–216.

[12] M. Goeminne, A. Decan, and T. Mens, "Co-evolving code-related and database-related changes in a data-intensive software system," in *CSMR-WCRE*, 2014, pp. 353–357.

[13] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, and D. Damian, "The promises and perils of mining GitHub," in *Working Conf. Mining Software Repositories*, 2014, pp. 92–101.