2018 IEEE International Conference on Software Maintenance and Evolution

A Large-scale Empirical Study on Linguistic Antipatterns Affecting APIs

Emad Aghajani, Csaba Nagy, Gabriele Bavota, Michele Lanza
REVEAL @ Software Institute, Università della Svizzera italiana (USI) - Lugano, Switzerland

***Abstract*—The concept of monolithic stand-alone software systems developed completely from scratch has become obsolete, as modern systems nowadays leverage the abundant presence of Application Programming Interfaces (APIs) developed by third parties, which leads on the one hand to accelerated development, but on the other hand introduces potentially fragile dependencies on external resources. In this context, the design of any API strongly influences how developers write code utilizing it. A wrong design decision like a poorly chosen method name can lead to a steeper learning curve, due to misunderstandings, misuse and eventually bug-prone code in the client projects using the API. It is not unfrequent to find APIs with poorly expressive or misleading names, possibly lacking appropriate documentation. Such issues can manifest in what have been defined in the literature as Linguistic Antipatterns (LAs), *i.e.,* inconsistencies among the naming, documentation, and implementation of a code entity. While previous studies showed the relevance of LAs for software developers, their impact on (developers of) client projects using APIs affected by LAs has not been investigated.**

This paper fills this gap by presenting a large-scale study conducted on 1.6k releases of popular Maven libraries, 14k open-source Java projects using these libraries, and 4.4k questions related to the investigated APIs asked on Stack Overflow. In particular, we investigate whether developers of client projects have higher chances of introducing bugs when using APIs affected by LAs and if these trigger more questions on Stack Overflow as compared to non-affected APIs.

***Index Terms*—Empirical Study; Application Programming Interfaces (APIs); Linguistic Antipatterns;**

I. Introduction

The usage of Application Programming Interfaces (APIs) is an integral part of software development, and it strongly influences how developers build their applications. For instance, it has been shown that the stability of a software system highly depends on the libraries it uses [1], or that placing a method in the right API class can significantly speed up development, even up to an order of magnitude [2].

The design of an API is particularly important, and previous studies investigated what makes an API *usable* or *maintainable* [3]–[5]. Books have been written about this topic [6], [7], and developers can refer to guidelines or "best practices" prepared for these purposes [8], [9]. The design of an API directly affects its usage [10] and learning curve [11]. In such a context, factors playing a role include, but are not limited to, naming, encapsulation, object-oriented design, explicitness of pre/post-conditions, and updated documentation. On this last point, Robillard and Deline have identified API documentation as the main source of learning obstacles for developers [12]. It has also been shown that – despite many APIs being actively maintained and updated –, these documents are also prone to mistakes and inconsistencies, due to the high cost of keeping them updated and in sync with changes [13], [14].

Duala-Ekoko and Robillard [15] have shown that developers rely on the API names when the documentation is missing or is incomplete. However, assigning good names to API methods is not an easy task [16] and poorly chosen names can lead to problems later. In this context Arnaoudova *et al.* [17] formalized issues affecting the design and documentation of code components, presenting a catalogue of 17 *Linguistic Antipatterns* (LAs), representing inconsistencies among the naming, documentation, and implementation of an entity. The authors showed that LAs are perceived negatively by developers since they hinder program comprehension. A recent study [18] using Near Infrared Spectroscopy to observe the cognitive load of 70 undergraduate and graduate students working with code snippets found that the presence of LAs significantly increases the cognitive load of developers.

Given the importance of *comprehensibility* and *usability* in the context of APIs, we conjecture that APIs affected by LAs can be problematic for client projects using them. To validate our conjecture we performed a large-scale study to investigate:

- *The impact of LAs affecting APIs on the likelihood of introducing bugs in the client projects using the API.* We analyze 1.6k releases of 75 popular Maven libraries exposing a total of 1.6M unique API methods and 14k client Java projects using them. We use the LA detection tool by Arnaoudova *et al.* [17] to identify LAs affecting the 1.6M APIs. For each client project C_i using a set of APIs A_{C_i} provided by the considered libraries, we mine the commits in C_i introducing the first usage of each API in A_{C_i}. Finally, using the SZZ algorithm [19], we identify bug-inducing commits and compare the likelihood of introducing a bug in the client project when using for the first time an API affected/not-affected by LAs.
- *Whether developers tend to ask more questions on Stack Overflow about APIs affected by LAs.* This would be an indication of higher difficulties experienced by developers in comprehending and adequately using APIs affected by LAs. We analyzed 4.4k Stack Overflow questions in which one of the API methods provided by the 75 Maven libraries is explicitly mentioned. We compare the proportion of questions asked for APIs affected/not-affected by LAs.

We quantitatively and qualitatively analyze our data.

2576-3148/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSME.2018.00012

While our statistical analysis suggests that when using an API for the first time, developers of the client projects have a 29% higher chance of introducing a bug if such an API is affected by a LA, our qualitative investigation highlighting no influence of LAs on the likelihood of introducing bugs in the client project. Similarly, we found no evidence that LAs affecting APIs trigger *Stack Overflow* questions. Our findings, besides providing a different perspective on the impact of LAs on code-related activities, also emphasize the need for combining both quantitative and qualitative findings in this type of observational studies.

Structure of the paper. Section II introduces the concept of Linguistic Antipatterns and surveys the related literature. III presents the study design, while our findings are discussed in Section IV. The threats that could affect their validity are presented in Section V. Section VI concludes the paper.

II. Background

We first present the LAs introduced by Arnaoudova *et al.* [17]. Then, we survey the related literature focusing on: (i) studies investigating APIs usability, design, and documentation; and (ii) techniques and tools to help developers in using APIs.

A. Source Code Linguistic Antipatterns

Arnaoudova *et al.* [17] presented a catalogue of 17 LAs capturing inconsistencies among the naming, documentation, and implementation of attributes and methods. The authors showed that LAs are negatively perceived by developers who highlighted their negative impact on code comprehension. They also released a tool for detecting LAs in Java code[1]. We focus on the 12 LAs related to methods, since we aim at investigating their impact on (developers of) client projects using APIs affected by LAs. These 12 antipatterns are classified into three categories (A, B, and C) briefly described in the following. For each category we report one example, while we refer the interested reader to Table 1 in [17] for a complete description of the LAs accompanied by real examples found in open source projects. In the catalogue, each type of LA is identified with an id (*e.g.,* A.1 is the first LA belonging to the A category). We use the same ids to ease the mapping between Table 1 in [17] and our work.

Category A: do more than they say. This category includes four LAs (A.1 - A.4) related to methods that do more than what their signature and documentation indicate. For example, *A.1 "Get" - more than accessor* identifies getter methods which do actions other than returning the corresponding attribute without documenting it [17].

Category B: say more than they do. Includes five LAs (B.1 - B.6) related to methods doing less than what their signature/documentation says. For instance, *B.1 Not implemented condition* affects methods in which the comment suggests a conditional behavior not implemented in the body [17].

[1]http://www.veneraarnaoudova.com/linguistic-anti-pattern-detector-lapd/

Category C: do the opposite than they say. This category includes two LAs (C.1 and C.2) affecting methods implementing behavior that is the opposite as compared to the one suggested by their signature and comments. For example, *C.1 Method name and return type are opposite* identifies methods having a name that is in contradiction with their return type (*e.g.,* a method named `disable` having `ControlEnableState` as return type) [17].

Arnaoudova *et al.* [17] defined such a catalogue and provided a tool to automatically identify them. They also show that developers perceive the defined LAs as poor coding practices likely to negatively affect code comprehension.

B. On API Usability, Design, and Documentation

Several studies focused the attention on the API usability and factors promoting/hindering it.

McLellan *et al.* [4] suggest the need for usability tests for APIs in the same way in which usability tests are performed in the context of user interface design. Myers and Stylos [3] echo such a recommendation, indicating usability as one of the key factors to optimize when designing an API, no less important than its correctness.

Ko *et al.* [20] showed the difficulties experienced by developers when dealing with APIs requiring the use of multiple objects. Stylos and Myers [2], inspired by this finding, ran a user study to investigate the role played by method placement (*i.e.,* which class the method belongs to) in the usability of APIs requiring the use of multiple objects. Their findings show that method placement plays an important role, strongly impacting developers' performance when dealing with APIs.

Ellis *et al.* [21] ran a user study to assess the impact on the API usability of the factory design pattern as compared to the adoption of simple class constructors. They observed that, in many situations, adopting the factory pattern significantly lowers the API usability.

Stylos and Clarke [22] investigated whether programmers are more effective when using APIs requiring constructor parameters as compared to parameterless default constructors. Their findings highlight the strong preference (and higher effectiveness) programmers have for APIs that do not require constructor parameters.

Robillard [11] reported on the results of a survey conducted with 83 developers and investigating the APIs learning obstacles experienced by developers. Among the identified obstacles, the ones relevant for our study are the issues related with the documentation and with the API's structural design. Indeed, they are at the basis of the LAs definition.

Piccioni *et al.* [16] performed a study with 25 programmers to investigate API usability. The study takes advantage of a combination of interviews with the participants and systematic observation of their behavior during programming tasks. Among the many interesting findings they report, one is particularly important for our work which highlights the difficulty of defining proper names when designing an API.

Indeed, the naming of methods is one of the main aspects considered by Arnaudova *et al.* [17] in the definition of LAs.

Duala-Ekoko and Robillard [15] conducted a controlled experiment with 20 developers to understand the types of questions they ask when facing unfamiliar APIs. Overall, the authors collected over 20 hours of screen captured videos spanning 40 implementation tasks. As part of their findings, the authors report that developers have difficulties guessing an API semantic from its name.

Acar *et al.* [23] studied whether the usability of APIs provided by several cryptographic libraries impacts the ability of developers to create secure code. Their study has been conducted with 256 Python developers and shows that APIs designed for simplicity (*e.g.,* guiding the developers by reducing the decision space) are not always enough since poor documentation or the lack of code examples can still hinder developers' ability to cope with them. On the other side, good documentation and examples can make developers comfortable to work with complex APIs.

Robillard and Deline also emphasized the importance of reference documentation when learning how to use an API [12]. They investigated API learning obstacles faced by developers and their findings motivated several studies to understand the essential elements needed to properly document APIs.

Maalej and Robillard [24] proposed a taxonomy of knowledge types in API reference documentation by investigating the documentation of two popular frameworks. Their taxonomy overviews the types of information reported in APIs documentation and can be used by developers to evaluate the content of their documentation.

Watson *et al.* [25] reviewed the API documentation of 33 popular libraries to verify whether it includes the elements of desirable API documentation defined in previous work. They found that most of the analyzed documentations included most (or all) the aspects of desirable API documentation, with a high standard for writing quality. Uddin and Robillard [26] report on the ten most common problems with API documentation, with ambiguity, incompleteness, and incorrectness classified as the three most severe problems.

Sohan *et al.* [27] presented a controlled user study to assess the importance of usage examples in REST API documentation. They found a substantial gap in the developers' productivity when such information is missing.

Finally, it is worth mentioning the many available "catalogues of good practices" on how to design APIs. For instance, Joshua [28] provides a set of guidelines to adopt when designing APIs (*e.g.,* APIs should be self-documenting, easy to use and hard to misuse). Varga [5] defines a set of good practices to improve the APIs maintainability. Michi [29] discusses APIs common design shortcomings and suggests API design rules, *e.g.,* "*an API should be minimal, without imposing undue inconvenience on the caller*". In addition, several language-specific guidelines to help developers with API design have been defined (*e.g.,* see [8], [9]).

As compared to the discussed work, *our study is the first one investigating the impact of LAs on the (developers of) client projects using the affected APIs*. Thus, our study complements the ones performed in the literature.

C. Assisting Developers With API Usage

Several techniques and tools have been proposed in the literature to support developers in using APIs. Many of them aim at creating code examples for an API of interest. In this line of research falls MAPO, the tool proposed by Xie and Pei [30] and extended by Zhong *et al.* [31]. MAPO can mine abstract usage examples of a given API method. UP-Miner [32] is a variation of MAPO that removes the redundancy in the resulting example list.

Buse and Weimer [33] proposed to generate documented abstract API usages by extracting and synthesizing code examples of a particular API data type. Moreno *et al.* [34] presented MUSE, an approach to automatically generate concrete usage examples of a given API mined from client projects using such an API. Glassman *et al.* [35] developed a visualization tool that mines API usage code examples regarding a given API and summarizes them with the goal of assisting developers in learning API usage.

A different type of work is the approach by Robillard and Chhetri [36]. They present an automated approach developed as an IDE plugin named Krec, to identify and retrieve the relevant piece of information in API reference documentation. Krec is able to categorize the text fragments in API documentation as indispensable, valuable, or neither, based on their semantic content.

Petrosyan *et al.* [37] proposed a text classifier-based approach to automatically retrieve tutorial sections explaining how to use a given API type, while Treude and Robillard [38] present an automated approach to augment API type documentation with a complementary relevant piece of information discussed on *Stack Overflow* posts. They show their machine learning based approach surpasses existing techniques, *e.g.,* text summarization. Furthermore, Azad *et al.* [39] present a technique to predict how an API call a developer will use by identifying co-changing API elements from the change history of open source projects and *Stack Overflow* posts. Earlier, Dekel and Herbsleb [40] developed an IDE plugin, called eMoose, which augments API method invocation with usage directives extracted from Javadoc to get developers aware of them.

III. Study Design

The *goal* of the study is to investigate (i) the impact of LAs affecting APIs on the likelihood of introducing bugs in the client projects using the API, and (ii) whether developers are more prone to ask questions on Stack Overflow when the APIs are affected by LAs. The *context* is represented by 1.6k releases of 75 Maven libraries, 14k client projects using those libraries, and 4.4k Stack Overflow questions. The *quality focus* is on APIs source code quality and comprehensibility that might be negatively affected by the presence of LAs.

A. Research Questions

Our study addresses the following two research questions:

RQ$_1$. *What is the impact of the LAs affecting APIs on the likelihood of introducing bugs in the client project?* Here with "client project" we refer to the project using the API.

We conjecture that the presence of LAs in the APIs can create issues to the developers of the client projects that might misinterpret the API and introduce bugs when using it. Indeed, previous studies showed the negative impact of LAs on the comprehensibility of the affected code components [17]. Note that we do not limit our analysis to the comparison of APIs affected and not-affected by LAs, but we also investigate how each of the 12 different method-related LAs defined by Arnaoudova *et al.* [17] increases the likelihood of introducing bugs when using APIs affected by it.

RQ_2. *Are APIs affected by LAs more likely to trigger discussion on Stack Overflow?* This research question aims at verifying whether LAs trigger more questions from developers using the affected API methods. As for RQ_1, we also report the types of LAs triggering more questions.

B. Context Selection

To answer our research questions the first step is the selection of the Java libraries to analyze, and their client projects. We limit our study to Java since, as we stated in Section II, the tool we use to detect LAs only supports Java code.

Due to the need of automatically identifying the client projects of a given library, we decided to focus our study on Maven libraries. Indeed, client projects interested in using a Maven library simply define a `pom.xml` file to specify the libraries they want to use. We selected all libraries belonging to the four most popular Maven categories[2]: *Testing Frameworks* (45 libraries and 1,103 releases), *Logging Frameworks* (38 and 997), *Core Utilities* (5 and 248), and *JSON Libraries* (67 and 1,369). The number of mined releases excludes non-final-release versions such as *beta*, *release candidate*, *etc.* since APIs might be not yet finalized in those versions.

Overall, we collected 3,708 release versions of selected libraries and we mined GitHub to identify their client projects. Using the GitHub search API, we first identified in GitHub all Java projects having at least one `pom.xml` file, needed to declare dependencies toward Maven libraries. This resulted in the identification of 17,659 client projects, using 118,626 `pom.xml` files and declaring ∼1.1M dependencies in total. We downloaded all the identified pom files and converted them into a standard format. This is needed since it is possible to use variables in pom files, or to declare dependencies using version intervals or relative version schemas (*e.g.,* declaring a dependency towards the `latest` version of a library). Since we need to know the exact version from which the client project depends on, we used the `mvn help:effective-pom` command to preprocess the pom files and obtain dependencies with their absolute version numbers.

Finally, we excluded all `pom.xml` files not reporting any dependency towards one of the 3,708 library releases subject of our study. This left us with 14,743 client projects.

Once collected the ∼14k client projects, we excluded from our study all library releases for which we did not identify any client project.

[2]https://mvnrepository.com/open-source

TABLE I
MAVEN LIBRARIES AND CLIENT PROJECTS CONSIDERED

Category	#Libraries	#Releases	#Client Projects
Testing Frameworks	25	268	13,169
Logging Frameworks	19	304	8,732
Core Utilities	5	175	7,343
JSON Libraries	26	545	6,703
Total	**75**	**1,642**	**14,743**

Indeed, client projects are needed to answer RQ_1, and we preferred to have a consistent dataset for both research questions. This decreased the number of libraries considered in our study to 75 for a total of 1,642 releases. Table I shows the number of libraries and releases we consider for each of the four popular Maven categories as well as the number of client projects identified for them.

C. Data Extraction

This section describes the data extraction process we followed to answer our research questions.

1) Parsing the libraries and the client projects, and identifying LAs: We downloaded the source code of the 1,642 library releases by using the `mvn dependency:sources` command. Then, we used the *Eclipse JDT Parser* to parse the code of each library to extract all the method declarations creating a database of 1.6M public (*i.e.,* API) and 800k private methods. When only considering the latest release of each of the considered libraries (used for RQ_2 as well), these numbers drop to 57k and 29k for public and private methods, respectively. Besides that, during the parsing process we also extracted precise type information related to the fully qualified class name of the method parameters' type, the return type, and the class defining each method. This information is needed to accurately (i) identify API invocations in the client projects (needed for RQ_1) and (ii) link Stack Overflow questions to library APIs (needed for RQ_2).

To identify the LAs affecting the APIs, we exploited the tool by Arnaoudova *et al.* [17] and able to detect the linguistic antipatterns described in Section II.

2) RQ_1-specific data extraction: Similarly to what was done for libraries, we used the *Eclipse JDT Parser* to parse the 14k client projects and extract from them a total number of 96M invocations together with their precise type information. In particular, given the `jar` files of the libraries the client project depends on, the parser is able to bind the invoked methods to their original declaration and extract type information regarding the parameters, return type and the class whose the method belongs to. Having available the fully qualified class name of the class defining the invoked method, it is possible to differentiate between local and non-local invocations. We mark as local invocations (and exclude them since irrelevant for our study) all those related to methods declared in classes having one of the client project's packages in their fully qualified name. From the remaining non-local invocations, we exclude the ones related to methods belonging to classes from the `java.*` packages.

Finally, we compare all the remaining non-local invocations with the APIs declared in the library versions the client project depends on (excluding libraries not considered in our study). Such a matching is precise thanks to the fact that we consider the complete method signature, the fully qualified names of the types of its parameters, its return type, and of the class declaring it.

The collected method calls from the client projects to the libraries are necessary but not sufficient for answering RQ_1. Indeed, our goal is to compare the likelihood of introducing a bug in the client project when using for the first time an API affected/not-affected by LAs. To this aim, we also need to identify (i) the exact commit in which each API used by each client project has been introduced for the first time in its code, and (ii) the bug-introducing commits, meaning commits that likely induced a bug-fixing activity. This way we can count when the use of an API for the first time (affected/not-affected by LAs) resulted in the introduction of bugs.

We used the `git log -L` $\mathtt{l}_n$`,`$\mathtt{l}_n$`:`$\mathtt{F}_{path}$ command to identify for each API invocation in the client projects the commit in their change history in which they have been introduced for the first time. In the command, $\mathtt{l}_n$ indicates the line number in which the method invocation is present in the client's code, and $\mathtt{F}_{path}$ is the path of the client's file containing the invocation. The command traces back the commit history of the source code at the given line, and we took the commit where the API invocation was first added.

To identify bug-fixing activities performed during the change history of the client projects, we used an approach proposed by Fischer *et al.* [41], *i.e.,* by mining regular expressions containing issue IDs and the keyword "fix" in the commit notes, *e.g.,"fixed issue #ID"* or *"issue ID"*. Then, we identify commits that introduced bugs[3] by using the SZZ algorithm [19], which is based on the annotation/blame feature of versioning systems. In summary, given a bug-fix commit, k, the approach works as follows:

1) For each file f_i, $i = 1 \dots m_k$ involved in the bug-fix k (m_k is the number of files changed in the bug-fix k), and fixed in its revision *rel-fix*$_{i,k}$, we extract the file revision just *before* the bug fixing (*rel-fix*$_{i,k} - 1$).
2) Starting from the revision *rel-fix*$_{i,k} - 1$, for each source line in f_i changed to fix the bug k the *blame* feature of git is used to identify the file revision where the last change to that line occurred. This produces, for each file f_i, a set of $n_{i,k}$ fix-inducing revisions *rel-bug*$_{i,j,k}$, $j = 1 \dots n_{i,k}$. Thus, more than one commit can be indicated by the SZZ algorithm as responsible for inducing a bug.

Matching the commits in which APIs have been introduced for the first time and those that introduced bugs will allow us to answer RQ_1 through the data analysis described later.

[3]The right terminology is "when the bug induced the fix" because of the intrinsic limitations of the SZZ algorithm, which cannot precisely identify whether a change actually introduced the bug.

3) RQ_2-specific data extraction: We mine the official Stack Overflow dump released in June 2017 to identify all questions explicitly mentioning an API method from the latest release of one of the 75 considered libraries. Such an analysis is limited to the latest library releases since API versions are rarely explicitly mentioned in the posts.

For this analysis, we implemented an approach to extract the qualified names of methods referenced in the code blocks of *Stack Overflow* questions. Existing approaches typically look for class names of APIs mentioned in the text, code block or `href` markup links of Stack Overflow [38], [42], [43]. However, we need an approach which can link *Stack Overflow* questions to exact API methods, also considering their parameters.

First, we extract code blocks from Stack Overflow posts tagged with the Java tag, we parse these code blocks with the srcML infrastructure [44], and then we collect the method signatures for method invocations with an algorithm which runs on the AST of a code block provided by srcML. For parsing, we have chosen srcML as a lightweight and robust parser, which can tolerate the usually incomplete source fragments on *Stack Overflow* but provides the necessary information for our analysis. Here, we have to be prepared for sample code snippets with often missing `import` statements or even class or method declarations. In addition, developers tend to use code blocks sometimes only for formatting purposes, *e.g.,* to emphasize numbers or sample commands sometimes written in other languages. To avoid these, we filtered code blocks shorter than 20 characters. After the extraction of code blocks, our algorithm collects the type information (*i.e.,* class name) of declaration nodes in the AST and for method invocations on local/instance variables, it pairs the method name and number of arguments with the type information of the related variable. If it cannot find the referenced variable, handles the reference as a static reference. As a result, for each code block we have a set of `className,methodName,numberOfArguments` tuples describing all method invocations in the code block. The extracted method references are stored in a database along with the API method declarations and they are linked to each other.

We analyzed 1,269,994 questions in Stack Overflow having a total number of 2,071,992 code blocks. After the parsing step, the collection of class and method name pairs provided us 804,104 unique tuples and a total number of 3,308,072 method references.

Knowing the Stack Overflow questions referencing each specific API will allow us to answer RQ_2 by verifying whether APIs affected by LAs trigger more questions from developers.

D. Data Analysis

To answer RQ_1, we compare the likelihood of introducing a bug in the first commit introducing in the client projects APIs affected and not-affected by LAs. In particular, we compute the following four groups:

- **ANB**$_{Clean}$, the number of commits introducing for the first time an API not affected by any LA that do not induce a bug;

TABLE II
NUMBER OF LIBRARIES/RELEASES/METHODS AFFECTED BY LAS

LA id	LA Name	Overall			public (APIs)		
		#Libraries	#Releases	#Methods	#Libraries	#Releases	#Methods
A.1	"Get" - more than accessor	34	669	4,160	29	564	2,680
A.2	"Is" returns more than boolean	16	348	6,337	15	330	5,977
A.3	"Set" method returns	32	697	5,680	23	593	4,029
A.4	Expecting but not getting single instance	37	853	5,846	33	822	4,459
B.1	Not implemented condition	41	831	10,418	38	803	9,539
B.2	Validation method does not confirm	30	621	2,170	20	371	561
B.3	"Get" method does not return	17	424	1,147	13	293	781
B.4	Not answered question	10	282	1,463	6	118	1,148
B.5	Transform method does not return	12	136	624	9	112	276
B.6	Expecting but not getting a collection	27	634	2,491	25	586	2,080
C.1	Method name and return type are opposite	6	44	63	4	30	49
C.2	Method signature and comment are opposite	40	909	5,549	35	771	3,690
		59	**1,078**	**43,778**	**56**	**1,047**	**33,633**

- $\mathbf{AB}_{Clean}$, the number of commits introducing for the first time an API not affected by any LA that induce a bug;
- $\mathbf{ANB}_{LA}$, the number of commits introducing for the first time an API affected by a LA that do not induce a bug;
- $\mathbf{AB}_{LA}$, the number of commits introducing for the first time an API affected by a LA that induce a bug;

Then, we use Fisher's exact test [45] to test whether the proportions of $\text{AB}_{Clean}/\text{ANB}_{Clean}$ and $\text{AB}_{LA}/\text{ANB}_{LA}$ significantly differ. In addition, we use the Odds Ratio (OR) [45] of the two proportions as effect size measure. An OR of 1 indicates that the condition or event under study (*i.e.,* the chances of inducing a bug) is equally likely in two compared groups (*e.g.,* clean *vs* LA). An OR greater than 1 indicates that the condition or event is more likely in the first group (that, in our analysis, will be LA). On the other hand, an OR lower than 1 indicates that the condition or event is more likely in the second group (Clean).

We also perform the same analysis when considering specific types of LAs. Meaning that, for each of the LA_i types we detected in our dataset, we compute the groups $\mathbf{ANB}_{LA_i}$ and $\mathbf{AB}_{LA_i}$ and again compare the proportion $\text{AB}_{LA_i}/\text{ANB}_{LA_i}$ with that of the $Clean$ group, with the goal of identifying what the most "dangerous" LAs are (if any).

To answer RQ_2, again we rely on the Fisher's exact test and on the odds ratio to verify whether developers tend to ask more questions about APIs affected by LAs as compared to clean APIs. We also compare the distributions representing the number of Stack Overflow questions triggered by APIs affected and not-affected by LAs. We use the Mann-Whitney test to compare the two distributions [46] with results intended as statistically significant at $\alpha = 0.05$. We also estimate the magnitude of the differences by using the Cliff's Delta (d), a non-parametric effect size measure [47]. We follow well-established guidelines to interpret it: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [47].

Finally, we qualitatively analyze our findings in both research questions.

IV. RESULTS

Before answering our research questions, we start by describing our dataset from different perspectives in order to give the reader a complete view of the subject APIs, client projects, and LAs.

Out of 2.4M methods defined in the 1,642 Maven releases we analyzed, 1.6M methods (66.0%) are public (*i.e.,* API methods). When only considering the latest release of the 75 subject libraries, the percentage of public methods is stable at 66.3%. Thus, although an API should be as minimal as possible to avoid revealing unnecessary details [29], the studied Maven libraries expose a high number of public methods.

We also inspected the presence of Javadoc documentation in the 2.4M methods. We used the *Eclipse JDT Parser* to detect comments using the Javadoc syntax (*i.e.,* `/** ... */`) right before a method declaration. We found that 22.5% of methods have a Javadoc comment, with such a percentage increasing to 46.2% when only focusing on public methods. These percentages are quite stable when only considering the latest release of each library (48.6% for public methods and 23.3% for all methods). While the higher Javadoc coverage for public methods as compared to private methods is a quite expected results (since these are the methods client projects are supposed to use), we still found that more than half of public methods are not documented through Javadoc. A possible explanation for this finding could be that many public getter and setter methods present in the studied libraries are not documented since, in many cases, their code is self-explanatory. We verified such an explanation by computing the number of public getter and setter methods in the set of 2.4M methods and by verifying how many of them are not documented. Overall, we found 518k getters and setters (406k getters and 111k setters), 189k of which (148k getters and 41k setters) documented (36%). Thus, excluding getters and setters from the counting, we still have 49% of uncommented public methods, do not substantially changing our finding.

Table II reports the LAs we found in our dataset. For each of the twelve LAs we considered, we report: (i) its id (column "LA id") allowing its mapping to Table 1 in the work by Arnaoudova *et al.* [17]; (ii) its name, providing a short description of the type

of issue it captures; (iii) the number of libraries and releases, among the ones we studied (*i.e.,* 75 libraries and 1,642 releases), in which we found at least one method affected by it; and (iv) the total number of methods affected by it. We report this information both when considering all methods ("Overall" in Table II) as well as when only focusing on public methods.

We found 43,778 methods out of 2.4M (1.8%) affected by LAs, with such a percentage growing to 2.1% when only focusing on public methods (33,633 out of 1.6M). What is more interesting is that 64% of the studied releases have at least one public method affected by a LA. Thus, knowing whether the LAs increase the likelihood of introducing bugs in the client projects and of triggering questions on *Stack Overflow* is worth investigating.

Moving to the client projects and their relationship with the libraries we found 96.8M method invocations in the 14,635 client projects: 53.9M (55.7%) are local invocations[4], 28.1M (29.1%) are related to Java APIs, 2.2M (2.2%) concern invocations to APIs belonging to the studied libraries, and the remaining 12.6M (13.0%) target APIs from other libraries.

TABLE III
TOP-TEN RELEASES IN TERMS OF PERCENTAGE OF PUBLIC API METHODS USED BY THEIR CLIENTS

Library Release (groupId:artifactId:version)	#used	#API	%
net.minidev:json-smart:2.3	184	453	41%
org.hamcrest:hamcrest-core:1.3	89	252	35%
org.hamcrest:hamcrest-all:1.3	185	655	28%
com.googlecode.json-simple:json-simple:1.1.1	26	105	25%
junit:junit:4.12	344	1,369	25%
commons-lang:commons-lang:2.6	476	2,317	21%
org.slf4j:slf4j-api:1.7.25	87	439	20%
com.google.code.gson:gson:2.8.2	146	763	19%
com.unboundid.components:json:1.0.0	34	194	18%
com.esotericsoftware.minlog:minlog:1.2	4	31	13%

Interestingly, we found that client projects only use a very limited subset of the public methods exposed by libraries. Considering all releases, we found that only 2.4% of public methods (38,246 out of 1,613,176) are used by at least one client project. Such a percentage grows to 7.0% (3,986 out of 57,369) when only focusing on public methods belonging to the latest releases of the analyzed libraries. This finding confirms what has been observed by Sawant and Bacchelli [48], who reported that a considerably small portion of an API is actually used by developers. Table III reports the top 10 library releases in terms of the percentage of their APIs used by at least one client project.

Another interesting observation derived from our dataset is that 83.9% of API methods used in client projects (the same percentage holds when considering all releases as well as when only focusing on the latest release) is accompanied by a Javadoc documentation. Such a percentage is much higher as compared to the percentage of all public methods having a Javadoc comment (*i.e.,* 51.0% in the best case scenario, when not considering getters and setters).

[4]We discriminate between local and non-local invocations as described in Section III-C2.

Although we did not dig further into this finding via qualitative analysis, these numbers clearly show a correlation between the presence of Javadoc comment in public methods and their usage in client projects. The problem here is the impossibility to define the direction of the causation. Indeed, we do not know whether the client projects actually tend to use documented APIs or, instead, are the developers of the APIs that tend to only document APIs they expect to be used by client projects. Such an investigation is part of our future research agenda.

A. RQ_1: What is the impact of the LAs affecting APIs on the likelihood of introducing bugs in the client project?

As explained in Section III-D, we extracted for each client project: (i) the commit in which each API it uses has been added for the first time in its code, and (ii) its bug-introducing commits, meaning the commits identified by the SZZ algorithm as likely to have triggered a bug-fixing activity in the future.

Having this data, we computed the cardinality of the four sets AB_{Clean}, ANB_{Clean}, AB_{LA}, and ANB_{LA} (see Section III-D for their definition). When considering all the twelve types of LAs, we obtained the following cardinalities: AB_{Clean}=1980, ANB_{Clean}=54918, AB_{LA}= 122, and ANB_{LA}=2612, leading to a statistically significant (p-value = 0.007) odds ratio of 1.29. This means that *when an API call is introduced in a client project for the first time, the likelihood of introducing a bug is 29% higher if the API is affected by a linguistic antipattern.*

TABLE IV
ODDS RATIO BY TYPE OF LA (SIGNIFICANT RESULTS ONLY)

id	LA Name	AB_{LA}	ANB_{LA}	Ratio	p-value
B.1	Not implemented condition	36	521	1.92	0.000
B.4	Not answered question	13	187	1.93	0.031
B.5	Transform method does not return	7	50	3.88	0.003

We also performed the same analysis for the 12 types of LAs we considered, and report the results in Table IV for the LAs for which we obtained a statistically significant odds ratio. The first thing that leaps to the eyes is that all the LAs substantially increasing the chance of introducing bugs belong to the "B category" of LAs. These LAs are related to methods that do less than what their signature/documentation says.

The *B.1 - Not implemented condition* LA affects methods in which the comment suggests a conditional behavior that is not implemented in the method's body [17]. For example, if the comment states "*Returns true if the balance is higher than 0, false otherwise*" but that does not implement any if statement to check the balance is affected by this LA. For B.1 the odds ratio is 1.92, indicating that developers have 92% higher chance of introducing a bug when committing for the first time usages of APIs affected by this LA as compared to clean APIs. While the statistical analysis provides a quite bold message, we manually analyzed all 36 commits in which, according to our data, the B.1 LA induced a bug-fixing activity, to verify what the role played by the LA actually was.

We found that in none of the 36 analyzed commits the usage of the API affected by the LA was actually the trigger for the future bug-fixing activity. This is due to the fact that the LAs of type B.1 involved in the 36 commits, while not false positives according to the B.1 definition, are not harmful. Let us explain why with one representative example, the case of the `concat` method implemented in *com.google.guava* library. In the Javadoc comment of the `concat` method it is documented a conditional behavior: "`@Throws NullPointerException` if any of the provided iterators is null", and such a behavior is not implemented in the method body through a conditional statement verifying whether the iterators provided as parameters are null. This makes `concat` affected by the *B.1 - Not implemented condition* LA. However, the `concat` method invokes the `checkNotNull` method by passing to it the iterators. The latter method is the one implementing the conditional statement throwing a `NullPointerException` when needed. Clearly, detecting these cases is far from trivial, since it requires interprocedural code analysis, currently not supported by the LA detection tool we used. In this specific case the bug was introduced in the same commit in which an invocation to this API was added in the client project, but the bug was not due to a misuse of such API. Similar observations hold for the other 35 commits.

The *B.4 - Not answered question* LA affects methods having their name in the form of predicate (*e.g.,* `isValidURL`) but do not returning a `boolean` [17]. For B.4 the odds ratio is 1.93, indicating that developers have 93% higher chance of introducing a bug when committing for the first time usages of APIs affected by this LA as compared to clean APIs. Also in this case our manual analysis did not highlight a direct effect of the LA on the bug introduction. The detection tool perfectly worked, and did not detect any false positive. The problem was in the specific context in which the LAs were detected. Indeed, all the B.4 instances involved in the bug-inducing commits were detected in methods from classes assisting in the validation of arguments. For example, the `isTrue` method from the `Assert` class of the *org.springframework* library has its name in the form of predicate but returns `void`. The reason is that, as documented in the Javadoc, this method "asserts a `boolean` expression, throwing an `IllegalArgumentException` if the expression evaluates to `false`". In such a context, while the B.4 LA clearly affects the method, it is unlikely to be harmful. All the bug-inducing commits we analyzed follows such a pattern, and did not play a direct role in the introduction of the bugs we analyzed.

Finally, the *B.5 - Transform method does not return* LA is the one exhibiting the highest odds ratio (3.88), indicating that developers have $\sim$4 times the chance of introducing bugs when working with APIs affected by B.5 as compared to clean APIs. This LA affects methods having a name suggesting the transformation of an object but not returning anything (as opposed to the expected transformed object) [17]. In this case, our qualitative analysis showed that all the bug-introducing commits were related to the usage, from different client projects, of the `toJson` method from the *com.google.code.gson*. This method actually returns `void` in the library releases involved in the bug-inducing commits, thus being classified as affected by the B.5 LA. However, `toJson` takes as one of its parameters a `writer` that, as documented in the Javadoc, represents the "*Writer to which the Json representation needs to be written*". In other words, while the transformation of the object to JSON does not result in a new object to be returned, the output of this transformation is written somewhere and well documented in the method. Also in this case, the LA did not look responsible for the bug introduction in the analyzed cases.

Summary for RQ$_1$: Our statistical analysis indicated that when introducing for the first time APIs affected by LAs in the code base, developers have 29% higher chance of introducing bugs as compared to when using clean APIs. Such an effect is mostly due to three types of LAs, namely *B.1 - Not implemented condition*, *B.4 - Not answered question*, and *B.5 - Transform method does not return*. However, in our qualitative analysis we did not find any strong evidence of their negative impact on the likelihood of introducing bugs.

B. RQ$_2$: Are APIs affected by LAs more likely to trigger discussion on Stack Overflow?

We had to face a number of challenges when linking APIs to *Stack Overflow* questions. We found classes with the same names in multiple libraries and/or in the Java API. When looking for *Stack Overflow* questions mentioning these classes but do not reporting a package import in the code block (as it is very frequent in *Stack Overflow* posts), it is not possible to identify precisely which class of which library is referenced at that location. For example, the `dbunit` library has an `InputStream` class in the `org.dbunit.util.Base64` package. Moreover, this class has a `read` method without parameters just like the `read` method of `java.io.InputStream`. We found 616 questions with a code block using the `InputStream.read()` method but without a reference to the library or the package name the method belongs to. Thus, it is impossible to determine whether the `dbunit` library or the Java API was referenced in these questions. For this reason, we filter out from the set of APIs to link to the *Stack Overflow* questions (i) all classes appearing with the same name as another class in the Java API and/or in another library; and (ii) all methods which appear with the same name and arguments in another class of another library. We found that 200 classes of the libraries appear with the same name in the Java 9 API (Java Platform, SE, and JDK) and 136 classes in the Java EE 7 API. We also found 7,291 methods appearing with the same name, declaring class and number of parameters in multiple libraries. In the end, we have 34,260 public methods of 5,261 classes in our dataset to investigate how LAs trigger discussions on Stack Overflow. Remember that in this investigation we only focus on the APIs present in the last release of the 75 subject libraries.

These API methods were referenced in 4,464 questions on *Stack Overflow* including 135 questions related to LAs.

TABLE V
ODDS RATIO OF METHODS DISCUSSED IN STACK OVERFLOW QUESTIONS WITH/WITHOUT LINGUISTIC ANTIPATTERNS

SO_{LA}/$NoSO_{LA}$	=	39/716	=	0.0544	(a)
SO_{Clean}/$NoSO_{Clean}$	=	891/33,406	=	0.0266	(b)
OddsRatio	=	**(a)/(b)**	=	**2.05**	

To address RQ_2, we calculate the *odds ratio* of methods (not)mentioned in Stack Overflow questions and methods (not)affected by LAs. Table V shows the different method sets needed to calculate the odds ratio. As done in RQ_1, LA is the set of methods affected by Linguistic Antipatterns, while $Clean$ are methods not affected. SO are methods mentioned at least in one *Stack Overflow* question, and $NoSO$ are methods not mentioned at all. As a result, the *odds ratio* is 2.05 indicating that methods affected by LAs are twice more likely to trigger questions on *Stack Overflow* than clean methods.

When comparing the distribution of the number of *Stack Overflow* questions related to methods affected and not affected by LAs with the Mann-Whitney, the p-value turns out to be 0.249 which, at a $\alpha = 0.05$, indicates no significant difference (and a negligible effect size of -0.06).

TABLE VI
LINGUISTIC ANTIPATTERNS FOUND IN METHODS WITH RELATED QUESTIONS ON SO

LA id	LA name	#Quest.
A.1	"Get" - more than accessor	21
A.2	"Is" returns more than boolean	4
A.3	"Set" method returns	3
A.4	Expecting but not getting single instance	10
B.1	Not implemented condition	20
B.4	Not answered question	10
B.7	Method does not return the corresponding attribute	32
C.1	Method name and return type are opposite	1
C.2	Method signature and comment are opposite	34

We also investigated which LAs affect the APIs discussed in *Stack Overflow* questions. Table VI shows the number of different LAs found in methods which were also mentioned in *Stack Overflow* questions. The major part of the warnings is related to the A1, B7, C2 and B1 categories. Note that in Table VI we report the total number of questions asked for the methods affected by LAs (*i.e.,* 135), while in the analysis with odds ratio we considered the number of methods affected by LAs and linked to at least one Stack Overflow question (*i.e.,* 39). Lastly, the detailed list of libraries having methods affected by LAs and discussed on SO can be seen in Table VII.

We manually investigated all these 135 questions. Our approach to identifying methods in code blocks of *Stack Overflow* questions spotted the library and the correct method of the API for 106 questions. For the rest, five questions are not available online anymore on *Stack Overflow* (we relied on the last release of the *Stack Overflow* database dump from June 2017), and in the remaining 24 cases it found a method with the same name, parameters and a declaring class of another library. This means that the approach was successful in 82% of the methods manually inspected.

TABLE VII
LIBRARIES HAVING METHODS AFFECTED BY LAS AND DISCUSSED ON SO

GroupId	ArtifactId	Methods	Questions
com.google.guava	guava	5	27
org.codehaus.plexus	plexus-utils	3	19
org.springframework	spring-core	6	19
log4j	log4j	4	17
xmlunit	xmlunit	1	15
junit	junit	3	13
commons-lang	commons-lang	1	5
org.apache.logging.log4j	log4-core	3	5
com.fasterxml.jackson.core	jackson-databind	2	3
org.apache.logging.log4j	log4j-api	2	3
com.pivotallabs	robolectric	2	2
org.springframework	spring-test	2	2
org.testng	testng	1	2
com.fasterxml.jackson.core	jackson-core	1	1
com.google.code.gson	gson	1	1
com.jayway.restassured	rest-assured	1	1
org.httpunit	httpunit	1	1

We also checked whether the questions were indeed closely related to the API methods. We found 50 cases closely related to the usage of the method in the API, while in other cases the method was part of the sample code, but the questions were about some other code components or were not related to how the API should be used. For instance, the `Stopwatch.stop()` method of `com.google.guava` appeared in 14 questions discussing some performance issues. The code samples in these questions measured time with `Stopwatch`, but they were not related to the usage of `Stopwatch`.

Regarding the discussion of a problem related to the LAs, we found only three cases where the problem mentioned in the question could indeed originate from a problem related to the LA affecting the method. Even for these cases, this was not explicitly mentioned. An example can be seen in Figure 1. The `Assume.assumeTrue` method of *junit* has the following LA: "*C.2: Method comments and signature use antonyms: false versus true. Signature: Assume.assumeTrue(boolean b): void*". The reason for the LA is the documentation of the method, which says the following: "*If called with an expression evaluating to false, the test will halt and be ignored*". More interestingly, the `assumeFalse` has the following comment: *The inverse of assumeTrue(boolean).*" Although, the problem in the question is not explicitly related to the antonym in the documentation. Indeed, the documentation misses the information.

Summary for RQ_2: We did not find clear evidence that the existence of LAs admittedly triggers questions on *Stack Overflow*. We notice, however, that just like the example in Figure 1, some LAs are probably more prone to misunderstandings.

V. THREATS TO VALIDITY

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. This is the most important kind of threat for our study, and is related to:

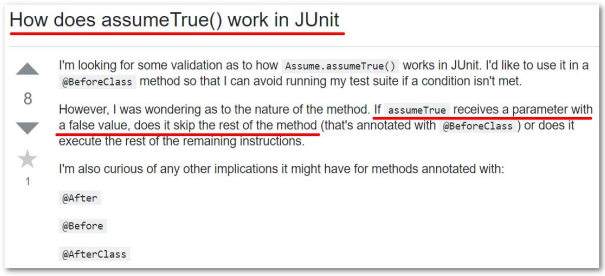

Fig. 1. An example SO question related to the usage of `Assume.assumeTrue(boolean)`, a *junit* method with a LA because of an antonym in the documentation.

- *RQ_1: Approximations due to identifying bug-fixing commits using regular expressions [41].* We used the approach proposed by Fischer *et al.* [41] mining regular expressions in commit notes to identify bug-fixing commits, thus possibly identifying false positive and missing false negative commits.
- *RQ_1: Approximations due to identifying bug-inducing commits using the SZZ algorithm [19].* We used heuristics to limit the number of false positives, for example excluding blank lines from the set of bug-inducing changes. However, we are aware of possible imprecisions introduced by the SZZ algorithm especially due to tangled commits [49] comprising a bug-fixing activity as well as other changes (*e.g.,* some refactoring operations).
- *RQ_1 and RQ_2: Accuracy of the LA detection tool.* To detect LAs we used the tools developed by Arnaoudova *et al.* [17]. Given the magnitude of our study, manually validating the output of the tool was clearly not an option. However, from the study reported in the original original paper introducing the tool we used [50], we know that the tool's precision for the twelve considered LAs is $\sim$77%. Moreover, our qualitative analysis helped in identifying some borderline LA instances impacting our findings.
- *RQ_2: Imprecisions in identifying Stack Overflow questions related to the investigated APIs.* Our approach to link *Stack Overflow* questions to methods of APIs relied on the extraction of method signatures from code blocks. This approach can miss cases when a method signature cannot be extracted from the code block or when it can be extracted, but the same signature appears in multiple APIs. This introduces imprecision in identifying questions. To estimate this imprecision, we manually investigated a sample set of 135 *Stack Overflow* questions which were related to methods with LAs. We observed a precision of 82%. However, due to a large number of questions tagged with Java, we could not estimate the recall of our approach, and we may miss questions related to APIs.

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. This type of threats strongly affect the findings of both our research questions. For what concerns RQ_1, the bug introductions for commits related to APIs affected by LAs might be due to several factors totally unrelated to the presence of LAs, and similar observations hold for RQ_2. For this reason, we addressed internal validity by qualitatively analyzing our results.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. Although this is mainly an observational study, wherever possible we used an appropriate support of statistical procedures, integrated with effect size measures, and qualitative analysis.

Threats to *external validity* concern the generalization of results. In RQ_1 we studied a total of 1,642 releases from 75 popular libraries and their 14,743 client projects, thus ensuring a good generalizability of our results for what concerns Java libraries and client projects. In RQ_2 we limited our study to the latest release of each of the 75 considered libraries due to the need of linking *Stack Overflow* questions to API methods. For both research questions, larger replications of our study possibly performed by also including languages different than Java can help to confirm or contradict our findings.

VI. Conclusion and Future Work

We investigated the impact of Linguistic Antipatterns (LAs) affecting APIs on the developers of client projects using such APIs. We studied whether (i) developers are more likely to introduce bugs when using for the first time APIs affected by LAs as compared to clean APIs, and (ii) developers tend to ask more questions when working with APIs affected by LAs as compared to clean APIs.

While our statistical analysis indicated some effect of LAs on the likelihood of introducing bugs and of triggering Stack Overflow questions, our qualitative analysis did not allow us to explain such a phenomenon. Clearly, this does not contradict the strong empirical evidence showing the negative impact of LAs on code comprehensibility [17], [18], nor the fact that LAs are considered as bad programming practices by software developers [17]. However, our findings call for additional investigation about the impact on LAs on code-related activities, maybe conducted through controlled experiments better allowing to isolate the effect of the studied variable.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects PROBE (SNF Project No. 172799) and JITRA (SNF Project No. 172479), and CHOOSE for sponsoring our trip to the conference.

References

[1] M. Kechagia, D. Mitropoulos, and D. Spinellis, "Charting the API minefield using software telemetry data," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1785–1830, Dec 2015.

[2] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 105–112.

[3] B. A. Myers and J. Stylos, "Improving API usability," *Communications of the ACM*, vol. 59, no. 6, pp. 62–69, 2016.

[4] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi, "Building more usable APIs," *IEEE software*, vol. 15, no. 3, pp. 78–86, 1998.

[5] E. Varga, *Creating Maintainable APIs: A Practical, Case-Study Approach*. Apress, 2016.

[6] J. Tulach, *Practical API Design: Confessions of a Java Framework Architect*, 1st ed. Berkely, CA, USA: Apress, 2008.

[7] J. Bloch, *Effective Java (3rd Edition) (The Java Series)*, 3rd ed. Pearson Education Inc., 2018.

[8] M. Reddy, *API Design for C++*. Elsevier, 2011.

[9] "https://swift.org/documentation/api-design-guidelines/," 2018.

[10] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of API-misuse detectors," *arXiv preprint arXiv:1712.00242*, 2017.

[11] M. P. Robillard, "What makes APIs hard to learn? Answers from developers," *IEEE software*, vol. 26, no. 6, 2009.

[12] M. P. Robillard and R. Deline, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[13] H. Zhong and Z. Su, "Detecting API documentation errors," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. ACM, 2013, pp. 803–816.

[14] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing APIs documentation and code to detect directive defects," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 27–37.

[15] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 266–276.

[16] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of API usability," in *Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 5–14.

[17] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, 2016.

[18] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope, "The effect of poor source code lexicon and readability on developers' cognitive load," in *Proceedings of the 26th IEEE/ACM International Conference on Program Comprehension, ICPC 2018*, Gothenburg, Sweden, 2018.

[19] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*. ACM, 2005.

[20] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, 2004, pp. 199–206.

[21] B. Ellis, J. Stylos, and B. Myers, "The factory pattern in API design: A usability evaluation," in *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 302–312.

[22] J. Stylos and S. Clarke, "Usability implications of requiring parameters in objects' constructors," in *Proceedings of the 29th International conference on Software Engineering, ICSE 2007*. IEEE Computer Society, 2007, pp. 529–539.

[23] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic APIs," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 154–171.

[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.

[25] R. Watson, M. Stamnes, J. Jeannot-Schroeder, and J. H. Spyridakis, "API documentation and software community values: a survey of open-source API documentation," in *Proceedings of the 31st ACM International Conference on Design of Communication*. ACM, 2013, pp. 165–174.

[26] G. Uddin and M. P. Robillard, "How API documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[27] S. Sohan, F. Maurer, C. Anslow, and M. P. Robillard, "A study of the effectiveness of usage examples in rest API documentation," in *Proceedings of the 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2017, pp. 53–61.

[28] J. Bloch, "How to design a good API and why it matters," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006, pp. 506–507.

[29] M. Henning, "API design matters," *Queue*, vol. 5, no. 4, pp. 24–36, 2007.

[30] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. ACM, 2006, pp. 54–57.

[31] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *Proceedings of the 23rd European Conference on Object-Oriented Programming, ECOOP 2009*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 318–343.

[32] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, 2013, pp. 319–328.

[33] R. P. L. Buse and W. Weimer, "Synthesizing API usage examples," in *Proceedings of the 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE, 2012, pp. 782–792.

[34] L. Moreno, G. Bavota, M. D. Penta, R. Oliveto, and A. Marcus, "How can I use this method?" in *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, 2015, pp. 880–890.

[35] E. Glassman, T. Zhang, B. Hartmann, and M. Kim, "Visualizing API usage examples at scale," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2018.

[36] M. P. Robillard and Y. B. Chhetri, "Recommending reference API documentation," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1558–1586, 2015.

[37] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining API types using text classification," in *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering, ICSE 2015*. IEEE, 2015, pp. 869–879.

[38] C. Treude and M. P. Robillard, "Augmenting API documentation with insights from Stack Overflow," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 392–403.

[39] S. Azad, P. C. Rigby, and L. Guerrouj, "Generating API call rules from version history and stack overflow posts," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 4, p. 29, 2017.

[40] U. Dekel and J. D. Herbsleb, "Improving API documentation usability with knowledge pushing," in *Proceedings of the 31st International Conference on Software Engineering, ICSE 2009*. IEEE, 2009, pp. 320–330.

[41] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *19th International Conference on Software Maintenance (ICSM 2003), 22-26 September 2003, Amsterdam, The Netherlands*, 2003, pp. 23–.

[42] D. Kavaler, D. Posnett, C. Gibler, H. Chen, P. T. Devanbu, and V. Filkov, "Using and asking: APIs used in the android market and asked about in StackOverflow," in *Social Informatics - 5th International Conference, SocInfo 2013, Kyoto, Japan, November 25-27*, 2013, pp. 405–418.

[43] C. Parnin, C. Treude, and L. Grammel, "Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow," Georgia Institute of Technology, Tech. Rep., 2012.

[44] J. I. Maletic and M. L. Collard, "Exploration, analysis, and manipulation of source code using srcML," in *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering, ICSE 2015*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 951–952.

[45] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. CRC Press, 2003.

[46] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[47] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[48] A. A. Sawant and A. Bacchelli, "fine-GRAPE: fine-grained APi usage extractor–an approach and dataset to investigate API usage," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1348–1371, 2017.

[49] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, May 18-19, 2013*, 2013, pp. 121–130.

[50] V. Arnaoudova, M. D. Penta, G. Antoniol, and Y. Guéhéneuc, "A new family of software anti-patterns: Linguistic anti-patterns," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, 2013, pp. 187–196.