On the Reaction to Deprecation of 25,357 Clients of 4+1 Popular Java APIs

Sawant, Anand; Robbes, Romain; Bacchelli, Alberto

DOI
10.1109/ICSME.2016.64

Publication date
2016

Document Version
Accepted author manuscript

Published in
Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016

2016 IEEE International Conference on Software Maintenance and Evolution

On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs

Anand Ashok Sawant
Delft University of Technology
Delft, The Netherlands
A.A.Sawant@tudelft.nl

Romain Robbes
PLEIAD @ DCC
University of Chile, Chile
rrobbes@dcc.uchile.cl

Alberto Bacchelli
Delft University of Technology
Delft, The Netherlands
A.Bacchelli@tudelft.nl

***Abstract*—Application Programming Interfaces (APIs) are a tremendous resource—that is, when they are stable. Several studies have shown that this is unfortunately not the case. Of those, a large-scale study of API changes in the Pharo Smalltalk ecosystem documented several findings about API deprecations and their impact on API clients.**

We conduct a partial replication of this study, considering more than 25,000 clients of five popular Java APIs on GitHub. This work addresses several shortcomings of the previous study, namely: a study of several distinct API clients in a popular, statically-typed language, with more accurate version information. We compare and contrast our findings with the previous study and highlight new ones, particularly on the API client update practices and the startling similarities between reaction behavior in Smalltalk and Java.

I. Introduction

An Application Programming Interface (API) is a definition of functionalities provided by a library or framework that is made available to an application developer. APIs promote the reuse of existing software systems [1]. In his landmark essay "No Silver Bullet" [2], Brooks argued that reuse of existing software was one of the most promising attacks on the essence of the complexity of programming: "*The most radical possible solution for constructing software is not to construct it at all.*"

Revisiting the essay three decades later [3], Brooks found that indeed, reuse continues to be the most promising attack on essential complexity. APIs enable this: To cite a single example, we found at least 15,000 users of the Spring API.

However, reuse comes with the cost of dependency on other components. This is not an issue when said components are stable. But evidence shows that APIs are not always stable: The Java standard API for instance has an extensive deprecated API [1]. API developers often deprecate features and replace them with new ones, and over time remove these deprecated features. These changes can break the client's code. Studies such as Dig and Johnson's [4] found that API changes breaking client code are common.

The usage of a deprecated feature can be potentially harmful. Features may be marked as deprecated because they are not thread safe, there is a security flaw, or will be replaced by a superior feature. The inherent danger of using a feature that has been marked as obsolete is good enough motivation for developers to transition to the replacement feature as suggested by the API developers themselves.

Besides the above dangers of using deprecated features, they also lead to reduced code quality, and therefore to increased maintenance costs. With deprecation being a maintenance issue, we would like to see if API clients actually react to deprecated features of an API.

To our knowledge, Robbes *et al.* conducted the largest study of the impact of deprecation on API clients [5], investigating deprecated methods in the Squeak and Pharo software ecosystems. This study mined more than 2,600 Smalltalk projects hosted on the SqueakSource platform. Based on the information gathered they looked at whether the popularity of deprecated methods either increased, decreased or remained as is after their deprecation.

The Smalltalk study found that API changes caused by deprecation can have a major impact on the ecosystem. However, a small percentage of the projects actually reacts to an API deprecation. Out of the projects that do react, most of them systematically replace the calls to deprecated features with those that are recommended by API developers. Surprisingly, this was done despite the fact that API developers in Smalltalk do not appear to be documenting their changes as well as can be expected.

The main limitation of this study is being focused on a niche programming community *i.e.*, Pharo. This resulted in a small dataset with information from only 2,600 projects in the entire ecosystem. Additionally, with Smalltalk being a dynamically typed language, the authors had to rely on heuristics to identify the reaction to deprecated API features.

We conduct a non-exact replication [6] of the previous Smalltalk [5] study, also striving to overcome its limitations. We study the reactions of more than 25,000 clients of 5 different APIs, using the statically-typed Java language; we also collect accurate API version information.

Our results confirm that only a small fraction of clients react to deprecation, also in the Java ecosystem. Out of those, systematic reactions are rare and most clients prefer to delete the call made to the deprecated entity as opposed to replacing it with the suggested alternative one. This happens despite the carefully crafted documentation accompanying most deprecated entities.

[1] see http://docs.oracle.com/javase/8/docs/api/deprecated-list.html

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.64

II. Methodology

We define the research questions and describe our research method contrasting it with the study we partially replicate [5].

A. Research Questions

In line with our partial replication target, we try to keep as much as possible the same research questions as the original work. Given our additional information, we add one novel research questions (RQ0) and alter the order and partially the methodology we use to answer the research questions; this leads to some differences in the formulation. The research questions we investigate are:

RQ0: What API versions do clients use?
RQ1: How does API method deprecation affect clients?
RQ2: What is the scale of reaction in affected clients?
RQ3: What proportion of deprecations does affect clients?
RQ4: What is the time-frame of reaction in affected clients?
RQ5: Do affected clients react similarly?

B. Research Method, Contrasted With Previous Study

Robbes *et al.* analyzed projects hosted on the SqueakSource platform, that used the Monticello versioning system. The dataset contained 7 years of evolution of more than 2,600 systems, which collectively had over 3,000 contributors. They identified 577 deprecated methods and 186 deprecated classes in this dataset. If its results were very informative, this previous study had several shortcoming that this follow-up study addresses. We describe the methodology for collecting the data for this study by describing it at increasingly finer granularity: Starting from the selection of the subject systems, to detecting the use of versions, methods, and deprecations.

Subject systems. The original study was based on a rather specific dataset, the Squeak and Pharo ecosystems found on SqueakSource. Due to this, the set of systems that were investigated in the previous study was relatively small.
To overcome this limitation, we focus on a mainstream ecosystem: Java projects hosted on the social coding platform GitHub. Java is the most popular programming language according to various rankings [7], [8] and GitHub is the most popular and largest hosting service [9]. Our criteria for selection included popularity, reliability, and variety: We measure popularity in terms of number of clients in GitHub, length of history, and overall reputation (*e.g.*, in Stack Overflow); we ensure reliability by picking APIs that are regularly developed and maintained; and we select APIs pertaining to different domains. These criteria ensure that the APIs result in a representative evolution history, do not introduce confounding factors due to poor management, and do not limit the types of clients that use them.
We limit our study to Java projects that use the Maven build system, because Maven based projects use Project Object Model (POM) files to specify and manage the API dependencies that the project refers to. We searched for POM files in the master branch of Java projects and found approximately 42,000 Maven based ones on GitHub. By parsing their POM files, we were able to obtain all the APIs that they depend on. We then created a ranking of the most popular APIs, which we used to guide our choice of APIs to investigate.
This selection step results in the choice of 5 APIs hosted on GitHub, namely: Easymock [10], Guava [11], Guice [12], Hibernate [13], and Spring [14].[2] The first 6 columns of Table I provide additional information on these APIs.
Subsequently, we select the *main subjects* of this study: The clients of APIs introducing deprecated methods. Using the aforementioned analysis of the POM files, we have the list of all possible clients. We refine it using the GHTorrent dataset [15], to select only active projects. We also remove clients that had not been actively maintained in the 6 months preceding our data collection, to eliminate 'dead' or 'stagnating' projects. We totaled 25,357 projects that refer to one or more of 5 aforementioned popular APIs. The seventh column in Table I provides an overview of the clients selected, by API.

API version usage. Explicit library dependencies are rarely mentioned in Smalltalk and there are several ways to specify them, often programmatically and not declaratively; also, Smalltalk does not use import statements as Java does. Thus, it is hard to detect dependencies between projects (heuristics are needed [16]) and to analyze the impact of deprecated methods on client. In contrast, Maven projects specify their dependencies explicitly and declaratively: We can determine the API version a project depends on, hence answer more questions, such as if projects freeze or upgrade their dependencies.
In particular, we only consider projects that encode specific versions of APIs, or unspecified versions (which are resolved to the latest API version at that date). We do not consider ranges of versions, however very few projects use those (84 for all 5 APIs, while we include 25,357 API dependencies to these 5 APIs). In addition, few projects use unspecified API versions (269 of the 25,357, which we do include).

Fine-grained method/annotation usage. Due to the lack of explicit type information in Smalltalk, there is no way of actually knowing if a specific class is referenced and whether the method invocation found is actually from that referenced class. This does not present an issue when it comes to method invocations on methods that have unique names in the ecosystem. However, in the case of methods that have common names such as `toString` or `name` or `item`, this can lead to some imprecise results. In the previous study, Robbes *et al.* resorted to manual analysis of the reactions to an API change, but had to discard cases which were too noisy. In this study, Java's static type system addresses this issue without the need for a tedious, and conservative manual analysis. On the other hand, Java APIs can be used in various manners. In Guava, actual method invocations are made on object instances of the Guava API classes, as one would expect. However in Guice, clients use annotations to invoke API functionality, resulting in a radically different interaction model. These API usage variabilities must be considered.

[2]We do not mine the JDK itself, because to identify the JDK version required by a client one needs to rely on the client using the Maven compiler plugin. Yet, this plugin is rarely used, since it is mainly used to specify a JDK version other than the default one used by the client.

Table I
SUMMARY INFORMATION ON SELECTED CLIENTS AND APIS

API (GitHub repo)	Description	Inception	Releases	Unique entities		Number of clients	Usage across history	
				Classes	Methods		Invocations	Annotations
EasyMock (easymock/ easymock)	A testing framework that allows for the mocking of Java objects during testing.	Feb 06	14	102	623	649	38,523	-
Guava (google/guava)	A collections API that provides data structures that are an extension to the datastructures already present in the Java SDK. Examples of these new datastructures includes: multimaps, multisets and bitmaps.	Apr 10	18	2,310	14,828	3,013	1,148,412	-
Guice (google/guice)	A dependency injection library created by Google.	Jun 07	8	319	1,999	654	59,097	48,945
Hibernate (hibernate/ hibernate-orm)	A framework for mapping an object oriented domain to a relational database domain. We focus on the core and entitymanager projects under the hibernate banner.	Nov 08	77	2,037	11,625	6,038	196,169	16,259
Spring (spring-projects/ spring-framework)	A framework that provides an Inversion of Control(IoC) container, which allows developers to access Java objects with the help of reflection. We choose to focus on just the spring-core, spring-context and spring-test modules due to their popularity.	Feb 07	40	5,376	41,948	15,003	19,894	40,525

While mining for API usage we have to ensure that we connect a method invocation or annotation usage to the parent class to which it belongs. There are multiple approaches that can be taken to mining the usage data from source code. The first uses pattern matching to match a method name and the import in a Java file to find what API a certain method invocation belongs to. The second uses the tool PPA [17] which can work on partial programs and find the usage of a certain method of an API. The third builds the code of a client project and then parse the bytecode to find type-resolved invocations. Finally, the fourth uses the Eclipse JDT AST parser to mine type-resolved invocations from a source code file. We created a method, fine-GRAPE, based on the last approach [18], [19] that meets the following requirements:[3] (1) fine-GRAPE handles the large-scale data in GitHub, (2) it does not depend on building the client code, (3) it results in a type-checked API usage dataset, (4) it collects explicit version usage information, and (5) it processes the whole history of each client.

Detect deprecation. In Smalltalk, users insert a call to a deprecation method in the body of the deprecated method. This call often indicates which feature replaces the deprecated call. However, there is no IDE support. The IDE does not indicate to developers that the feature being used is deprecated. Instead, calls to deprecated methods output runtime warnings.

In contrast Java provides two different mechanisms to mark a feature as deprecated. The first is the `@deprecated` annotation provided in the Javadoc specification. This annotation is generally used to mark an artifact as deprecated in the documentation of the code. This feature is present in Java since JDK version 1.1. Since this annotation is purely for documentation purposes, there is no provision for it to be used in compiler level warnings. This is reflected in the Java Language Specification(JLS). However, the standard Sun JDK compiler does issue a warning to a developer when it encounters the usage of an artifact that has been marked as deprecated using this mechanism. More recently, JDK 1.5 introduced a second mechanism to mark an artifact as deprecated with a source code annotation called `@Deprecated` (The same JDK introduced the use of source code annotations). This annotation is a compiler directive to define that an artifact is deprecated. This feature is part of the Java Language Specification; as such any Java compiler supports it. It is now common practice to use both annotations when marking a certain feature as deprecated. The first is used so that developers can indicate in the Javadoc the reasons behind the deprecation of the artifact and the suggested replacement. The other is now the standard way in which Java marks features as deprecated.

To identify the deprecated features we first download the different versions of the APIs used by the clients from the Maven central server. These APIs are in the form of Java Archive (JAR) files, containing the compiled classes of the API source. We use the ASM [20] class file parsing library to parse all the classes and their respective methods. Whenever an instance of the `@Deprecated` annotation is found we mark the entity it refers to as deprecated and store this in our database. Since our approach only detects compiler annotations, we do not handle the Javadoc tag. See the threats to validity section for a discussion of this. We also do not handle methods that were removed from the API without warning, as these are out of scope of this study.

III. RESULTS

In this section we answer the research questions we detailed in Section II-A. Figure 1 exemplifies the behavior of an API and its clients, when possible we refer to it to explain the methodology behind the answer to each research question.

[3]More details on fine-GRAPE can be found in our prior work [19].

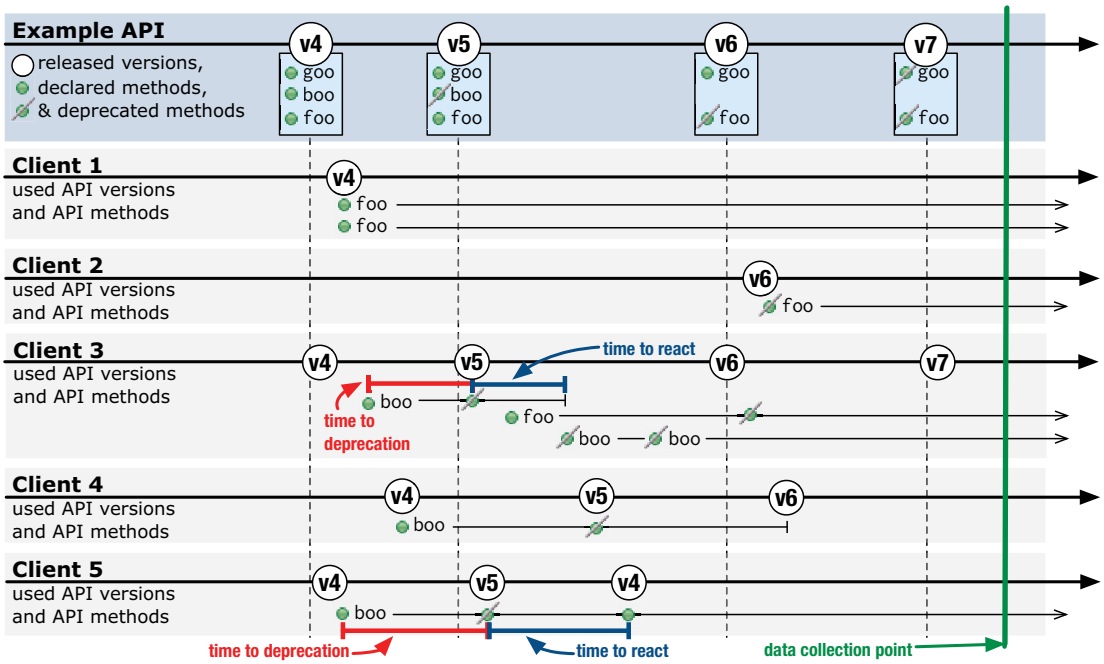

Figure 1. Exemplification of the behavior of an API and its clients

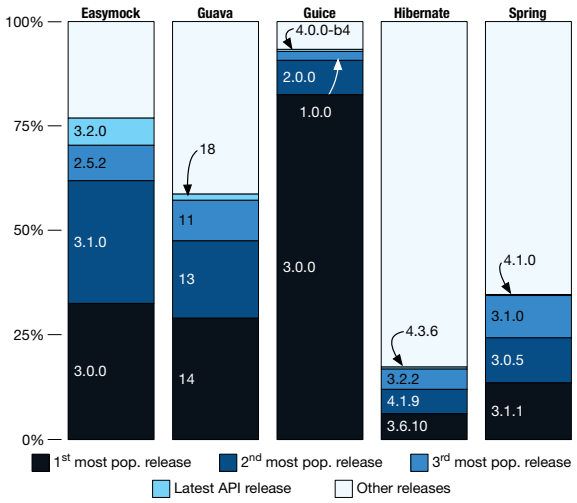

Figure 2. Popularity breakdown of versions, by API

RQ0: What API versions do clients use?

Our first research question seeks to investigate popularity of API versions and to understand the version change behavior of the clients. This sets the ground for the following answers.

We start considering all the available versions of each API and measure the popularity in terms of how many clients were actually using it at the time of our data collection. In the example in Figure 1, we would count popularity as 1 for v7, 2 for v6, and 1 for v4. The column 'number of clients' in Table I specifies the absolute number of clients per each API and Figure 2 reports the version popularity results, by API.

The general trend shows that a large number of different versions of the APIs are used and the existence of a significant fragmentation between the versions (especially in the case of Hibernate, where the top three versions are used by less than 25% of the clients). Further, the general trend is that older versions of the APIs are more popular.

This initial results hint at the fact that clients have, to say the least, a delayed upgrading behavior, which could be related with how they deal with maintenance and deprecated methods. For this reason, we analyze whether the clients ever updated their dependencies or if they "froze" their dependencies—that is, if they never updated their API version. In the example in Figure 1, we count three clients who upgraded version in their history. If projects update we measure how long they took to do so (time between the release of the new version of the API in Maven central, and when the project's POM file is updated).

Table II
UPDATE BEHAVIOR OF CLIENTS, BY API

	updated clients		update time (in days)			
			mean	median	Q_1	Q_3
Easymock	63	10%	404	272	103	592
Guava	610	20%	140	72	32	139
Guice	49	8%	783	909	251	1,150
Hibernate	2,454	41%	245	63	33	368
Spring	11,112	74%	195	69	37	186

Table II summarizes the results. The vast majority of the clients we consider freeze to one single version of the API they use. Further, we see that this holds for all the APIs, except for Spring, whose clients have at least one update in 74% of the cases. In terms of time to update, interestingly, the median is lower for clients of APIs that have more clients that update, such as Hibernate and Spring. In general, update time varies considerably—we will come back to this in RQ3.

RQ1: How does API method deprecation affect clients?

In RQ0 we showed that most clients do not adopt new API versions. We now focus on the clients that use deprecated methods and on whether and how they react to deprecation.

Affected by deprecation. From the data, we classify clients into 4 categories, which we describe referring to Figure 1:

- *Unaffected:* These clients never use a deprecated method. None of the clients in Figure 1 belong to this category.
- *Potentially affected:* These clients do not use any deprecated method, but should they upgrade their version, they would be affected. Client 1 in Figure 1 belongs to this category.
- *Affected:* These clients use a method when it was declared as deprecated, but do not change the API version throughout their history, as it happens in the case of Client 2.
- *Affected and changing version:* These clients use at least one method declared as deprecated and also update their API version. Clients 3, 4, and 5 belong to this category.

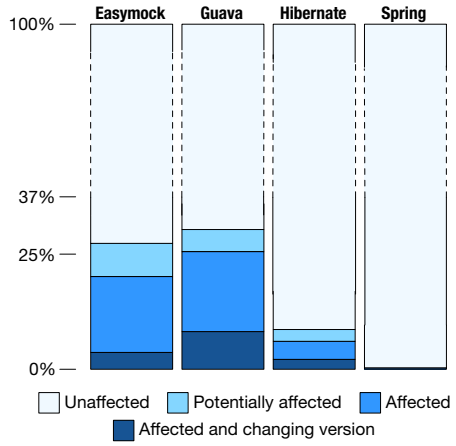

Figure 3. Deprecation status of clients of each API

Figure 3 reports the breakdown of the clients in the four categories. The clearest pattern is that the vast majority of clients, across all APIs, *never use any deprecated method* throughout their entire history. This is particularly surprising in the case of Hibernate, as it deprecated most of its methods (we will discuss this in RQ3). Clients affected by deprecation vary from more than 20% for Easymock and Guava, to less than 10% for Hibernate, and barely any for Spring. Of these, less than one third also change their API version, thus highlighting a very static behavior of clients with respect to API usage, despite our selection of active projects.

Common reactions to deprecation. We investigate how 'Affected and changing version' clients deal with deprecation. We exclude 'Affected' clients, since they do not have strong incentives to fix a deprecation warning if they do not update their API, as the method is still functional in their version.
The 'Affected and changing version' clients of Easymock and Guava largely react to deprecated entities (71% and 65%). For Hibernate and Spring we see a similar minority of clients that react (31% and 32%). For all the APIs the relative number of clients that fix all calls made to a deprecated entity is between 16% and 22%. Out of the clients that react, we find that at the method level, the most popular reaction is to *delete the reference* to the deprecated method (median of 50% to 67% for Easymock, Guava and Hibernate and 100% for Spring). We define as *deletion* a reaction in which the deprecated entity is removed and no new invocation to the same API is added. Some Hibernate and Guava clients roll back to a previous version where the entity is not yet deprecated. Easymock, Guava and Hibernate clients tend to replace deprecated calls with other calls to the same API, however this number is small. Surprisingly, a vast majority of projects (95 to 100%) add calls to deprecated API elements, despite the deprecation being already in place. This concerns even the ones that end up migrating all their deprecated API elements later on.

The strange case of Guice. We analyzed all the Guice projects and looked for usage of a deprecated annotation or method, however we find that none of the projects have used either. The reason is that Guice does not have many methods or annotations that have been deprecated. In fact, Guice follows a very aggressive deprecation policy: methods are removed from the API without being deprecated previously. We observed this behavior in the Pharo ecosystem as well, and studied it separately [21]. In our next research questions, we thus do not analyze Guice, as the deprecations are not explicitly marked.

RQ2: What is the scale of reaction in affected clients?

The work we partially replicate [5] measures the reactions of individual API changes in terms of commits and developers affected. Having exact API dependency information, we can measure API evolution on a per-API basis, rather than per-API element. It is hence more interesting to measure the magnitude of the changes necessary between two API versions in terms of the number of methods calls that need to be updated between two versions. Another measure of the difficulty of the task is the number of *different* deprecated methods one has to react to: It is easier to adapt to 10 usages of the same deprecation than it is to react to 10 usages of 10 different deprecated methods.

Actual reactions. We measure the scale of the actual reactions of clients that do react to API changes. We count separately reactions to the same deprecated method and the number of single reactions. In Figure 1, client 3, after upgrading to v5 and before upgrading to v6, makes two modifications to statements including the deprecated method 'boo'. We count these as two reactions to deprecation but count one unique deprecated method. We consider that client 5 reacts to deprecation, when rolling back from v5 to v4: we count one reaction and one unique deprecated method.
We focus on the upper half of the distribution (median, upper quartile, 95th percentile, and maximum), to assess the critical cases; we expect the effort needed in the bottom half to be low. Table III reports the results. The first column reports the absolute number of non-frozen affected clients that reacted. The scale of reaction varies: the majority of clients react on less than a dozen of statements with a single unique deprecated method involved. Springs stands out with a median number of

statements with reactions of 31 and the median number of *unique* deprecated methods involved of 17. Outliers, invest more heavily in reacting to deprecated methods. As seen next, this may explain the reluctance of some projects to update.

Table III
SCALE OF ACTUAL CLIENTS' REACTION TO METHOD DEPRECATION

	non-frozen affected clients that reacted	statements with reaction (unique deprecated methods involved)			
		median	Q_3	95th perc.	max
Easymock	17	11 (1)	21 (2)	109 (3)	109 (3)
Guava	161	3 (1)	8 (2)	127 (5)	283 (10)
Hibernate	40	5 (1)	20 (16)	41 (27)	59 (40)
Spring	10	31 (17)	54 (21)	104 (27)	131 (27)

Potential reactions. Since a large portion of project do not react, we wondered how much work was accumulating should they wish to update their dependencies. We thus counted the number of updates that a project would need to perform in order to make their code base compliant with the latest version of the API (*i.e.*, removing all deprecation warnings). In Figure 1, the only client that is potentially affected by deprecation is client 1, which would would have two statements needing reaction (*i.e.*, those involving the method 'foo') in which only one unique deprecated method is involved.

As before, we focus on the upper half of the distribution. Table IV reports the results. In this case the first column reports the absolute number of clients that would need reaction. We notice that the vast majority of clients use two or less unique deprecated methods. However, they would generally need to react to a higher number of statements, compared to the clients that reacted reported in Table III, except for those using Spring.

Overall, if the majority of projects would not need to invest a large effort to upgrade to the latest version, a significant minority of projects, would need to update a large quantities of methods. This can explain their reluctance to do so. However, this situation, if left unchecked—as is the case now—can and does grow out of control. If there is a silver lining, it is that the number of unique methods to update is generally low, hence the adaptations can be systematic. Outliers would run in troubles, with several unique methods to adapt to.

Table IV
SCALE OF POTENTIAL CLIENTS' REACTION TO METHOD DEPRECATION

	clients potentially needing reaction	statements potentially needing reaction (unique deprecated methods involved)			
		median	Q_3	95th perc.	max
Easymock	178	55 (1)	254 (1)	1,120 (5)	4,464 (7)
Guava	917	12 (1)	42 (2)	319 (7)	8,568 (44)
Hibernate	521	15 (1)	35 (1)	216 (2)	17,471 (140)
Spring	41	3 (1)	4 (1)	51 (2)	205 (55)

RQ3: What proportion of deprecations does affect clients?

The previous research question shows that most of the actual and potential reactions of clients to method deprecations involves a few unique methods. This does not tell us how these methods are distributed across all the deprecated API methods. We compute the proportion of deprecated methods clients use.

In Figure 1, there is at least one usage of deprecated methods 'boo' and 'foo', while there is no usage of 'goo'. In this case, we would count 3 unique deprecated methods, of which one is never used by clients.

Table V
DEPRECATED METHODS AFFECTING CLIENTS, BY API

	unique deprecated methods			
	defined by API		used by clients	
	count	% over total	count	% over all deprecated
Easymock	124	20%	16	13%
Guava	1,479	10%	104	7%
Hibernate	7,591	65%	487	6%
Spring	1,320	3%	149	11%

Table V summarizes the results, including the total count of deprecated methods per API with proportion over the total count of methods and the count of how many of these deprecated methods are used by clients. APIs are not shy in deprecating methods, with more than 1,000 deprecations for Guava, Spring, or Hibernate. The case of Hibernate is particularly striking with 65% of unique methods being eventually deprecated, indicating that this API makes a heavy usage of this feature of Java. The proportion of deprecated methods that affect clients is rather low, around 10% in all 4 of the APIs.

RQ4: What is the time-frame of reaction in affected clients?

We investigate the amount of time it takes for a method to become deprecated ('time to deprecation') and the amount of time developers take to react to a it ('time to react'). The former is defined as the interval between the introduction of the call and when it was deprecated, as seen in client 3 (Figure 1); the latter is the amount of time between the reaction to a deprecation and when it was deprecated (clients 3 and 5).

Time to deprecation. We analyzed the time to deprecation for each of the instances where we found a deprecated entity. The median time for all API clients is 0 days. This highlights a startling fact: Most of the introductions of deprecated method calls happen when clients already know they are deprecated. In other words, when clients introduce a call to a deprecated method, it is usually done despite the fact that they know *a priori* that the call is already deprecated. This indicates that clients do not appear to mind using deprecated features.

Time to react. Figure 4 reports the time it takes clients to react to a method deprecation, once it is visible. We see that, for most clients across all APIs, the median reaction time is low: It is 0 days for Guava, Hibernate, and Spring, while for Easymock it is 25 days. A reaction time of 0 days indicates that

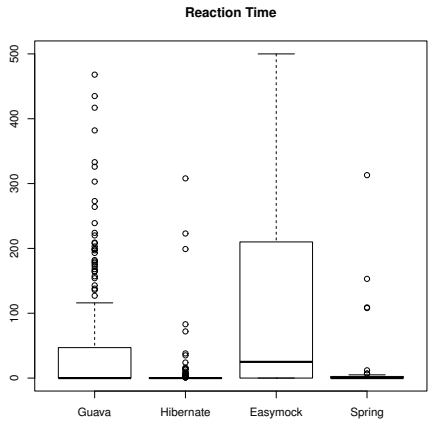

Figure 4. Days taken by clients to react to a method deprecation once visible.

closeQuietly

```
@Deprecated
public static void closeQuietly(@Nullable
                                Closeable closeable)
```

Deprecated. *Where possible, use the try-with-resources statement if using JDK7 or `Closer` on JDK6 to close one or more `Closeable` objects. This method is deprecated because it is easy to misuse and may swallow IO exceptions that really should be thrown and handled. See Guava issue 1118 for a more detailed explanation of the reasons for deprecation and see Closing Resources for more information on the problems with closing `Closeable` objects and some of the preferred solutions for handling it correctly. This method is scheduled to be removed in Guava 16.0.*

Equivalent to calling `close(closeable, true)`, but with no IOException in the signature.

Parameters:

`closeable` - the `Closeable` object to be closed, or null, in which case this method does nothing

Figure 5. Example of Javadoc associated with deprecated API artifact

most deprecated method call are reacted upon on the same day the call was either introduced or marked as deprecated. Barring outliers, reaction times in Hibernate and Spring are uniformly fast (the third quartiles being at 0 and 2.5 days). Reaction times are however longer for clients of Guava and Easymock, with an upper quartile of 47 and 200 days respectively. Outliers have a long reaction time, in the order of hundreds of days.

RQ5: Do affected clients react similarly?

Replacing a deprecated entity with an invocation to a non-deprecated one is a desirable reaction as the client of an API continues using it. This research question seeks to investigate the clients' behavior when it comes to replacement reactions.

Such an analysis allows us to ascertain whether an approach inspired by Schäfer *et al.*'s [22] would work on the clients in our dataset. Their approach recommends API changes to a client based on common, or systematic patterns in the evolution of other clients of the same API.

Consistency of replacements. There is no definite way to identify if a new call made to the API is a replacement for the original deprecated call, so we rely on a heuristic: We analyze the co-change relationships in each class file across all the projects; if we find a commit where a client removes a usage of a deprecated method (*e.g.*, `add(String)`) and adds a reference to another method in the same API (*e.g.*, `add(String, Integer)`), this new method invocation is a possible replacement for the original deprecated entity. A drawback is that in-house replacements or replacements from other competing APIs cannot be identified. Nonetheless, we compute the frequencies of these co-change relationships to find whether clients react uniformly to a deprecation.

We found that Easymock has no systematic transitions: there are only 3 distinct methods for which there are replacements and the highest frequency of the co-change relationships is 34%. For Guava we find 23 API replacements; in 17% of the cases there is a *systematic* transition *i.e.*, there is only one way in which a deprecated method is replaced by clients. Spring clients only react by deleting deprecated entities instead of replacing them, resulting in no information on replacements of features. In Hibernate, we find only 4 distinct methods where replacements were made. There were no systematic replacements and the maximum frequency is 75%.

Since API replacements are rather uncommon in our dataset, with the exception of Guava, we find that while an approach such as the one of Schäfer *et al.* could conceptually be quite useful, we would not be able to implement it in our case due to the small amount of replacement data.

Quality of documentation. There are very few clients reacting to deprecation by actually replacing the deprecated call with one that is not deprecated. This led us to question the quality of the documentation of these APIs. Ideally one would like to have a clear explanation of the correct replacement for a deprecated method, as in the Javadoc reported in Figure 5. However, given the results we obtained, we thought this could be not the case. We systematically inspected the Javadoc to see if deprecated features had documentation on why the feature was deprecated, and if there was an indication of appropriate replacement or whether a replacement is needed.

We perform a manual analysis to analyze the quality of the API documentations. For Guava, we investigate all 104 deprecated methods that had an impact on clients; for Easymock, we look at all 16 deprecated methods that had impact on clients; for Spring and Hibernate, we inspected sample of methods (100 each) that have an impact on the clients.

In Easymock, 15 of the 16 deprecated methods are instance creation methods, whose deprecation message directs the reader to using a Builder pattern instead of these methods. The last deprecation message is the only one with a rationale and is also the most problematic: the method is incompatible with Java version 7 since its more conservative compiler does not accept it; no replacement is given.

In Guava, 61 messages recommend a replacement, 39 state the method is no longer needed and hence can be safely deleted, and only 5 deprecated methods do not have a message. It is also the API with the most diverse deprecation

messages. Most messages that state a method is no longer needed are rather cryptic ("no need to use this"). On the other hand, several messages have more precise rationales, such as stating that functionality is being redistributed to other classes. Others provides several alternative recommendations and detailed instructions and one method provides as many as four alternatives, although this is because the deprecated method does not have exact equivalents. Guava also specifies in the deprecation message when entities will be removed (*e.g.*, "This method is scheduled for removal in Guava 16.0", or even "This method is scheduled for deletion in June 2013.").

For Hibernate, all the messages provide a replacement, but most provide no rationale for the deprecation. The only exceptions are messages stating the advantages of a recommended database connection compared to the deprecated one.

For Spring, the messages provide a replacement (88) or state that the method is no longer needed (12). Spring is the only API that is consistent in specifying in which version of the API the methods were deprecated. On the other hand, most of the messages do not specify any rationale for the decision, except JDK version testing methods that are no longer needed since Spring does not run in early JDK versions anymore.

Overall, maintainers of popular APIs make an effort to provide their clients with high-quality documentation. We classify this as high quality documentation as there is sufficient support provided to clients. If we found rationales as to why a method was deprecated, this was far from systematic. Despite replacement being not the only suggested solution, it is the most common; this is in contrast to the actual behavior of clients. In spite of the good quality of the documentation, clients are far from likely to follow it.

Summary of findings

We first investigated how many API clients actively maintain their projects by updating their dependencies. We found that, for all the APIs, only a minority of clients upgrade/change the version of the API they use. As a direct consequence of this, older versions of APIs are more popular than newer ones.

We then looked at the number of projects that are affected by deprecation. We focused on projects that change version and are affected by deprecation as they are the ones that show a full range of reactions. Clients of Guava, Easymock and Hibernate (to a lesser degree) were the ones that were most affected, whereas clients of Spring were virtually unaffected by deprecation and for Guice we could find no data due to Guice's aggressive deprecation policy. We also found that for most of the clients that were affected, they introduced a call to a deprecated entity, despite knowing that it was deprecated.

Looking at the reaction behavior of these clients, we saw that 'deletion' was the most popular way to react to a deprecated entity. Replacements were seldom performed, and finding systematic replacements was rarer. This is despite the fact that APIs provide excellent documentation that should aid in the replacement of a deprecated feature. When a reaction did take place, it was usually almost right after it was first marked as deprecated.

IV. Discussion

We now discuss our main findings and contrast them with the findings of the Smalltalk study we partially replicate. Based on this, we give recommendations on future research directions. We also present threats to validity.

A. Comparison with the deprecation study on Smalltalk

Contrasting our results with those of the study we partially replicate, several interesting findings emerge:

Proportion of deprecated methods affecting clients. Both studies found that only a small proportion of deprecated methods does affect clients. In the case of Smalltalk, this proportion is below 15%, in our results we found it to be around 10%. Considering that the two studies investigate two largely different ecosystems, languages, and communities, this similarity is noteworthy. Even though API developers do not know exactly how their clients use the methods they write and would be interested in this information [23], the functionalities they deprecate are mostly unused by the clients, thus deprecation causes few problems. Nevertheless, this also suggests that the majority of effort that API developers make in properly deprecating some methods and documenting alternatives is not actually necessary: API developers, in most of the cases, could directly remove the methods they instead diligently deprecate.

Not reacting to deprecation. Despite the differences in the deprecation mechanisms and warnings, the vast majority of the clients in both studies do *not* react to deprecation. In this study, we could also quantify the impact of deprecation should clients decide to upgrade their API versions and find that, in some cases, the impact would be very high. By not reacting to deprecated calls, we see that the technical debt accrued can grow to large and unmanageable proportions (*e.g.*, some Hibernate client would have to change 17,471 API invocations). We also found more counter-reactions (*i.e.*, adding more calls to methods that are known to be deprecated) than for Smalltalk clients. This may be related to the way in which the two platforms raise deprecation warnings: In Java, a deprecation gives a compile-time warning that can be easily ignored, while in Smalltalk, some deprecations lead to run-time errors, which require intervention.

Systematic changes and deprecation messages. The Smalltalk study found that in a large number of cases, most clients conduct systematic replacements to deprecated API elements. In our study, we find that, instead, replacements are not that common. We deem this difference to be extremely surprising. In fact, the clients we consider have access to very precise documentation that should act as an aid in the transition from a deprecated API artifact to one that is not deprecated; while this is not the case for Smalltalk, where only half of the deprecation messages were deemed as useful. This seems to indicate that proper documentation is not a good enough incentive for API clients to adopt a correct behavior, also from a maintenance perspective, when facing deprecated methods. As an indication to developers of language platforms, we have some evidence to suggest more stringent policies on how deprecation impacts clients' run-time behavior.

Clients of deprecated methods. Overall, we see in the behavior of API clients that deprecation mechanisms are not ideal. We thought of two reasons for this: (1) developers of clients do not see the importance of removing references to deprecated artifacts, and (2) current incentives are not working to overcome this situation. Incentives could be both in the behavior of the API introducing deprecated calls and in the restriction posed by the engineers of the language. This situation highlights the need for further research on this topic to understand whether and how deprecation could be revisited to have a more positive impact on keeping low technical debt and improve maintainability of software systems. In the following we detail some of the first steps in this direction, clearly emerging from the findings in our study.

B. Future research directions

If it ain't broke, don't fix it. We were surprised that so many projects did not update their API versions. Those that do are often not in a hurry, as we see for Easymock or Guice. Developers also routinely leave deprecated method calls in their code base despite the warnings, and even often add new calls. This is in spite of all the APIs providing precise instructions on which replacements to use. As such the effort to upgrade to a new version piles up. Studies can be designed and carried out to determine the reasons of these choices, thus indicating how future implementations of deprecation can give better incentives to clients of deprecated methods.

Difference in deprecation strategies. In the clients that do upgrade, we can see differences between the APIs. We were particularly impressed by Spring, which has by far the most clients and also the least clients using deprecations. It appears that their deprecation strategy is very conservative, even if they deprecated a lot of methods. This may explain why much more Spring clients do upgrade their API version. Likewise, perhaps the very aggressive deprecation policy of Guice, which removes methods without warnings, has an impact on the vast majority of the clients that decide to stick with their version of Guice. We note that the APIs with the highest proportion of projects with deprecated calls are also the ones where the projects are least likely to upgrade. We did not investigate this further, as our focus was mostly on the behavior of clients, but studies suggesting API developers the best strategies for persuading clients to follow deprecation messages would be very informative for the actual practice of software evolution.

Impact of deprecation messages. We also wonder if the deprecation messages that Guava has, which explicitly state when the method will be removed, could act as a double-edged sword: Part of the clients could be motivated to upgrade quickly, while others may be discouraged and not update the API or roll back. In the case of Easymock, the particular deprecated method that has no documented alternative may be a roadblock to upgrade. Studies can be devised to better understand the role of deprecation messages and their real effectiveness.

C. Threats to validity

Since we do not detect deprecation that is only specified by Javadoc tags, we may underestimate the impact of API deprecation in some cases. To quantify the size of this threat, we manually checked each API and found that this is an issue only for Hibernate before version 4, while the other APIs are unaffected. For this reason, a fraction of Hibernate clients could show not completely correct behavior. We considered crawling the online Javadoc of Hibernate to recover these tags, but we found that the Javadoc of some versions of the API were missing (e.g. version 3.1.9).

Even though our findings are focused on the clients, for which we have a statistically significant sample, some of the results depend on the analyzed APIs (such as the impact of the API deprecation strategies on the clients). As we suggested earlier in this section, further studies could be conducted to investigate these aspects.

The use of projects from GitHub leads to a number of threats, as documented by Kalliamvakou *et al.* [24]. In our data collection, we tried to mitigate these biases (*e.g.*, we only selected active projects), but some limitations are still present. The projects are all open-source and some may be personal projects where maintenance may not be a priority. GitHub projects may be toy projects or not projects at all (still from [24]); we think this is unlikely, as we only select projects that use Maven: these are by definition Java projects, and, by using Maven, show that they adhere to a minimum of software engineering practices.

Finally, we only look at the master branch of the projects. We assume that projects follow the git convention that the master branch is the latest working copy of the code [25]. However, we may be missing reactions to API deprecations that have not yet been merged in the main branch.

V. Related Work

Studies of API Evolution. Several studies of API evolution have been performed, at smaller or larger scales. Most of these studies focused on the API side, rather than the client one as the one we conducted.

For example, Dig and Johnson studied and classified the API breaking changes in 4 APIs [26]; they did not investigate their impact on clients. They found that 80% of the changes were due to refactorings. Cossette and Walker [27] studied five Java APIs in order to evaluate how API evolution recommenders would perform on these cases. They found that all recommenders handle a subset of the cases, but that none of them could handle all the cases they referenced.

The Android APIs have been extensively studied. McDonnell *et al.* [28] investigate stability and adoption of the Android API on 10 systems; the API changes are derived from Android documentation. They found that the API is evolving quickly, and that clients have troubles catching up with the evolution. Linares-Vásquez *et al.* also study the changes in Android, but from the perspective of questions and answers on Stack Overflow [29], not API clients directly. Bavota *et al.* [30] study how changes in the APIs of mobile apps (responsible

for defects if not reacted upon) correlate with user ratings: successful applications depended on less change-prone APIs. This is one of the few large-scale studies, with more than 5,000 API applications. Wang *et al.* [31] study the specific case of the evolution of 11 REST APIs. Instead of analyzing API clients, they also collect questions and answers from Stack Overflow that concern the changing API elements.

Among the studies considering clients of API, we find for example the one by Espinha *et al.* [32], who study 43 mobile client applications depending on web APIs and how they respond to web API evolution. Also, Raemaekers *et al.* investigated the relation among breaking changes, deprecation, and semantic versioning [33]. They found that API developers introduce deprecated artifacts and breaking changes in equal measure across both minor and major API versions, thus not allowing clients to predict API stability from semantic versioning. Finally, previous work including one of the authors of this paper (*i.e.*, [5] and [21]) are large-scale studies of API clients in the Pharo ecosystem. The first study focused on API deprecations, while the second one focused on API changes that were not marked as deprecations beforehand. Another work [34] analyze deprecation messages in more than 600 Java systems, finding that 64% of deprecated methods have replacement messages.

Mining of API Usage. Studies that present approaches to mining API usage from client code are related to our work, especially with respect to the data collection methodology.

One of the earliest works done in this field is the work of Xie and Pei [35] where they developed a tool called MAPO (Mining API usage Pattern from Open source repositories). MAPO mines code search engines for API usage samples and presents the results to the developer for inspection. Mileva *et al.* [36] worked in the field of API popularity; they looked at the dependencies of projects hosted on Apache and Sourceforge. Based on this information they ranked the usage of API elements such as methods and classes. This allowed them to predict the popularity trend of APIs and their elements. Hou *et al.* [37] used a popularity based approach to improve code completion. They developed a tool that gave code completion suggestions based on the frequency with which a certain class or method of an API was used in the APIs ecosystem. Lämmel *et al.* [38] mine usages of popular Java APIs by crawling SourceForge to create a corpus of usage examples that forms a basis for a study on API evolution. The API usages are mined using type resolved Java ASTs, and these usages are stored in a database.

Supporting API evolution. Beyond empirical studies on APIs evolution, researchers have proposed several approaches to support API evolution and reduce the efforts of client developers. Chow and Notkin [39] present an approach where the API developers annotate changed methods with replacement rules that will be used to update client systems. Henkel and Diwan [40] propose CatchUp!, a tool using an IDE to capture and replay refactorings related to the API evolution. Dig *et al.* [41] propose a refactoring-aware version control system for the same purposes. Dagenais and Robillard observe the framework's evolution to make API change recommendations [42], while Schäfer *et al.* observe the client's evolution [22]. Wu *et al.* present a hybrid approach [43] that includes textual similarity. Nguyen *et al.* [44] propose a tool (LibSync) that uses graph-based techniques to help developers migrate from one framework version to another. Finally, Holmes and Walker notify developer of external changes to focus their attention on these events [45].

VI. Conclusion

We have presented an empirical study on the effect of deprecation of Java API artifacts on their clients. This is a non-exact replication of a similar study done on the Smalltalk ecosystem. The main differences between the two studies is in the type systems of the language targeted (static type vs dynamic type) and the scale of the dataset (25,357 vs 2,600 clients).

We found that few API clients update the API version that they use. In addition, the percentage of clients that are affected by deprecated entities is less than 20% for most APIs—except for Spring where the percentage was unusually low. Most clients that are affected do not typically react to the deprecated entity, but when a reaction does take place it is—surprisingly—preferred to react by deletion of the offending invocation as opposed to replacing it with recommended functionality. When clients do not upgrade their API versions, they silently accumulate a potentially large amount of technical debt in the form of future API changes when they do finally upgrade; we suspect this can serve as an incentive not to upgrade at all.

The results of this study are in some aspects similar to that of the Smalltalk study. This comes as a surprise to us as we expected that the reactions to deprecations by clients would be more prevalent, owing to the fact that Java is a statically typed language. On the other hand, we found that the number of replacements in Smalltalk was higher than in Java, despite Java APIs being better documented. This leads us to question as future work what the reasons behind this are and what can be improved in Java to change this.

This study is the first to analyze the client reaction behavior to deprecated entities in a statically-typed and mainstream language like Java. The conclusions drawn in this study are based on a dataset derived from mining type-checked API usages from a large set of clients. From the data we gathered, we conclude that deprecation mechanisms as implemented in Java do not provide the right incentives for most developers to migrate away from the deprecated API elements, even with the downsides that using deprecated entities entail.

Given that there is currently a proposal to revamp Java's deprecation system,[4] studies such as this one and its potential follow-ups are especially timely.

[4]https://bugs.openjdk.java.net/browse/JDK-8065614

REFERENCES

[1] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.

[2] F. P. Brooks, "No silver bullet," *Software state-of-the-art*, pp. 14–29, 1975.

[3] S. D. Fraser, F. P. Brooks Jr, M. Fowler, R. Lopez, A. Namioka, L. Northrop, D. L. Parnas, and D. Thomas, "No silver bullet reloaded: retrospective on essence and accidents of software engineering," in *Proceedings of 22nd ACM SIGPLAN Conference on Object-oriented programming systems and applications (OOPSLA)*. ACM, 2007, pp. 1026–1030.

[4] D. Dig and R. E. Johnson, "The role of refactorings in api evolution," in *ICSM 2005: Proceedings of the 21st International Conference on Software Maintenance*, 2005, pp. 389–398.

[5] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation?: the case of a smalltalk ecosystem," in *Proceedings of 20th International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012, p. 56.

[6] N. J. Juzgado and S. Vegas, "The role of non-exact replications in software engineering experiments," *Empirical Software Engineering*, vol. 16, no. 3, pp. 295–324, 2011.

[7] "Tiobe index," http://www.tiobe.com/tiobe_index, accessed on 10 Apr 2016.

[8] "PYPL popularity of programming language," http://pypl.github.io, accessed on 10 Apr 2016.

[9] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean GHTorrent: Github data on demand," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 384–387.

[10] "Easymock api repository," https://github.com/easymock/easymock, accessed on 7 April 2016.

[11] "Guava api repository," https://github.com/google/guava, accessed on 7 April 2016.

[12] "Guice api repository," https://github.com/google/guice, accessed on 7 April 2016.

[13] "Hibernate api repository," https://github.com/hibernate/hibernate-orm, accessed on 7 April 2016.

[14] "Spring api repository," https://github.com/spring-projects/spring-framework, accessed on 7 April 2016.

[15] G. Gousios, "The ghtorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR 2013, 2013, pp. 233–236.

[16] M. Lungu, R. Robbes, and M. Lanza, "Recovering inter-project dependencies in software ecosystems," in *ASE'10: Proceedings of the 25th IEEE/ACM international conference on Automated Software Engineering*, ser. ASE '10, 2010, pp. 309–312.

[17] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," *ACM Sigplan Notices*, vol. 43, no. 10, pp. 313–328, 2008.

[18] A. A. Sawant and A. Bacchelli, "A dataset for api usage," in *Proceedings of 12th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2015, pp. 506–509.

[19] ——, "fine-grape: fine-grained api usage extractor – an approach and dataset to investigate api usage," *Empirical Software Engineering*, pp. 1–24, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10664-016-9444-6

[20] "Asm bytecode manipulator," http://asm.ow2.org/, accessed on 7 April 2016.

[21] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to api evolution? the pharo ecosystem case," in *Proceedings of 31st International Conference on Software Maintenance and Evolution (ICSME)*, 2015, p. in press.

[22] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *Proceedings of 30th International Conference on Software Engineering (ICSE)*, 2008, pp. 471–480.

[23] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering*, ser. ICSE '14. ACM, 2014, pp. 12–23.

[24] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 92–101.

[25] S. Chacon, *Pro git*. Apress, 2009.

[26] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring," *Journal of software maintenance and evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006.

[27] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *Proceedings of 20th International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012, p. 55.

[28] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *Proceedings of 29th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2013, pp. 70–79.

[29] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," in *Proceedings of 22nd International Conference on Program Comprehension (ICPC)*. ACM, 2014, pp. 83–94.

[30] G. Bavota, M. Linares-Vasquez, C. E. Bernal-Cardenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change-and fault-proneness on the user ratings of android apps," *IEEE Transactions on Software Engineering (TSE)*, vol. 41, no. 4, pp. 384–407, 2015.

[31] S. Wang, I. Keivanloo, and Y. Zou, "How do developers react to restful api evolution?" *Service-Oriented Computing*, pp. 245–259, 2014.

[32] T. Espinha, A. Zaidman, and H.-G. Gross, "Web api fragility: How robust is your mobile application?" in *Proceedings of the 2nd International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2015, pp. 12–21.

[33] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *28th IEEE International Conference onSoftware Maintenance (ICSM)*. IEEE, 2012, pp. 378–387.

[34] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "Do developers deprecate apis with replacement messages? a large-scale analysis on java systems," in *23nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Osaka, Japan, March 14-18, 2016*, 2016, p. to appear.

[35] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," in *Proceedings of the 2006 International workshop on Mining Software Repositories (MSR)*. ACM, 2006, pp. 54–57.

[36] Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining api popularity," in *Testing–Practice and Research Techniques*. Springer, 2010, pp. 173–180.

[37] D. Hou and D. M. Pletcher, "Towards a better code completion system by api grouping, filtering, and popularity-based ranking," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, ser. RSSE '10. New York, NY, USA: ACM, 2010, pp. 26–30. [Online]. Available: http://doi.acm.org/10.1145/1808920.1808926

[38] R. Lämmel, E. Pek, and J. Starek, "Large-scale, ast-based api-usage analysis of open-source java projects," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, 2011, p. 1317. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1982185.1982471

[39] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *Proceedings of International Conference on Software Maintenance (ICSM)*, 1996, pp. 359–368.

[40] J. Henkel and A. Diwan, "Catchup!: Capturing and replaying refactorings to support API evolution," in *Proceedings of 27th International Conference on Software Engineering (ICSE)*, 2005, pp. 274–283.

[41] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen, "Refactoring-aware configuration management for object-oriented programs," in *29th International Conference on Software Engineering*, 2007, pp. 427–436.

[42] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *Proceedings of 30th International Conference on Software engineering (ICSE)*, 2008, pp. 481–490.

[43] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 325–334.

[44] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010, pp. 302–321.

[45] R. Holmes and R. J. Walker, "Customized awareness: recommending relevant external change events," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 465–474.