TU Delft

Delft University of Technology

On the reaction to deprecation of clients of 4 + 1 popular Java APIs and the JDK

Sawant, Anand Ashok; Robbes, Romain; Bacchelli, Alberto

DOI
10.1007/s10664-017-9554-9

Publication date
2017

Document Version
Final published version

Published in
Empirical Software Engineering

Citation (APA)
Sawant, A. A., Robbes, R., & Bacchelli, A. (2017). On the reaction to deprecation of clients of 4 + 1 popular Java APIs and the JDK. *Empirical Software Engineering*, *23 (2018)*, 2158–2197. https://doi.org/10.1007/s10664-017-9554-9

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Empir Software Eng (2018) 23:2158–2197
https://doi.org/10.1007/s10664-017-9554-9

On the reaction to deprecation of clients of 4 + 1 popular Java APIs and the JDK

Anand Ashok Sawant[1] · **Romain Robbes**[2] · **Alberto Bacchelli**[3]

Published online: 13 October 2017
© The Author(s) 2017. This article is an open access publication

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 extend this study, by analyzing clients of both popular third-party Java APIs and the JDK API. This results in a dataset consisting of more than 25,000 clients of five popular Java APIs on GitHub, and 60 clients of the JDK API from Maven Central. 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. We make a comparison between reaction behavior for third-party APIs and JDK APIs, given that language APIs are a peculiar case in terms of wide-spread usage, documentation, and support from IDEs. Furthermore, we investigate the connection between reaction patterns of a client and the deprecation policy adopted by the API used.

Communicated by: Bram Adams and Denys Poshyvanyk

✉ Anand Ashok Sawant
A.A.Sawant@tudelft.nl

Romain Robbes
rrobbes@unibz.it

Alberto Bacchelli
bacchelli@ifi.uzh.ch

[1] Software Engineering Research Group (SERG), Delft University of Technology, Delft, The Netherlands

[2] Software and Systems Engineering Research Group (SwSE), Free University of Bozen-Bolzano, Bolzano, Italy

[3] Zurich Empirical Software Engineering Team (ZEST), University of Zurich, Zurich, Switzerland

Keywords Application programming interface · API usage · API popularity · Dataset

1 Introduction

An Application Programming Interface (API) is a definition of functionalities provided by a library or framework made available to other developer, as such. APIs promote the reuse of existing software systems (Johnson and Foote 1988). In his landmark essay "No Silver Bullet" (Brooks 1975), 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 (Fraser et al. 2007), Brooks found that indeed, reuse remains 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 (Sawant and Bacchelli 2016).

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] Deprecation is a mechanism employed by API developers to indicate that certain features are obsolete and htat they will be removed in a future release. API developers often deprecate features, and when replace them with new ones, changes can break the client's code. Studies such as Dig and Johnson's (2005) 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 are going to be replaced by a superior feature. The inherent danger of using a feature that has been marked as obsolete may be good enough motivation for developers to transition to the replacement feature.

Besides the aforementioned dangers, using deprecated features can 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.

Robbes et al. conducted the largest study of the impact of deprecation on API clients (Robbes et al. 2012), investigating deprecated methods in the Squeak (Ingalls et al. 1997) and Pharo (Black et al. 2009) software ecosystems. This study mined more than 2,600 Smalltalk projects hosted on the SqueakSource platform (Lienhard and Renggli 2005). They investigated whether the popularity of deprecated methods either increased, decreased or did not change after deprecation.

Robbes et al. found that API changes caused by deprecation can have a major impact on the studied ecosystems, and that a small percentage of the projects actually reacts to an API deprecation. Out of the projects that do react, most systematically replace the calls to deprecated features with those recommended by API developers. Surprisingly, this was done despite API developers in Smalltalk not documenting their changes as good as one would expect.

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 2600 projects in

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

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 (Juzgado and Vegas 2011) of the previous Smalltalk (Robbes et al. 2012) study, also striving to overcome its limitations. We position this study as an explorative study that has no pre-conceived notion as to what is correct behavior with respect to reaction to deprecation. To that end 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. The API clients analyzed in this study are open-source projects we collected on the GitHub social coding platform (http://www.github.com 2017).

Furthermore, we also investigate the special case of the Java Development Kit API (JDK), which may present peculiarities, because of its role popularity and tailored integration with most IDEs. For example, due to these features developers might be more likely to react to deprecation in the API of the language, as opposed to deprecations in an API that they use. To perform this analysis, we collect data from Maven Central (which allows for a more reliable way to collect JDK version data). We collected data from 56,410 projects and their histories, out of which we analyze 60 projects (selected to reduce the size of data to be processed) to see how they dealt with deprecated API elements in Java's standard APIs.

Our results confirm that only a small fraction of clients react to deprecation. In fact, in the case of the JDK clients, only 4 are affected and all 4 of these introduce calls to deprecated entities at the time of usage. 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.

One of the more interesting phenomena that the we observed was that out of the 5 APIs for which we observed the reaction pattern, we see that each of the APIs has its own way in which it deprecated features, which then has an impact on the client. APIs such as Spring appear to deprecate their features in a more conservative manner and thus impact very few clients. On the other hand, Guava appears to constantly making changes to their API, thus forcing their clients to deal with deprecation in the API, at the risk of having clients not upgrading to the latest version of the API. Given these patterns that we observed, we investigate whether we can categorize APIs based on the strategy they use when deprecating features. To this end we look at 50 popular Java APIs, and develop heuristics characterizing how these APIs deprecate features.

2 Methodology

We define the research questions and describe our research method contrasting it with the study we expand upon (Robbes et al. 2012).

2.1 Research Questions

To better contrast our results with the previous study on Smalltalk, we try to maintain the same research questions as the original work whenever possible.

The aim of these research questions is similar to the original paper and they aim to determine (1) whether deprecation of an API artifact affects API clients, (2) whether API clients do react to deprecation and (3), and to understand if immediately actionable information can be derived to alleviate the problem. To do this, we find out how often a deprecated entity impacts API clients and how these clients deal with it.

Given the additional information at our disposal in this paper, we add two novel research questions (RQ0 and RQ6). RQ0 aims to understand the API version upgrade behavior of API clients and RQ6 looks at the impact of various deprecation policies on the reaction of API clients. Furthermore, we alter the original order and partially change 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?
RQ6: How are clients impacted by API deprecation policies?

2.2 Research Method, Contrasted with the Previous Study

Robbes et al. analyzed projects hosted on the SqueakSource platform, which used the Monticello versioning system. The dataset contained 7 years of evolution of more than 2600 systems, which collectively had over 3000 contributors. They identified 577 deprecated methods and 186 deprecated classes. The results were informative, but this previous study had several shortcomings that we address. We describe the methodology to collect 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. In this work, the methodologies for the collection of Third-party API usage and JDK API usage is different, and these differences are reflected in Figs. 1 and 2.

For Third-Party APIs, we select candidate APIs based on their popularity (Fig. 1, top left); we then build the list of their clients (Fig. 1, bottom left), keeping only active projects; finally, for each project, we locate the usages of individual API elements in successive version of these projects (Fig. 1, right).

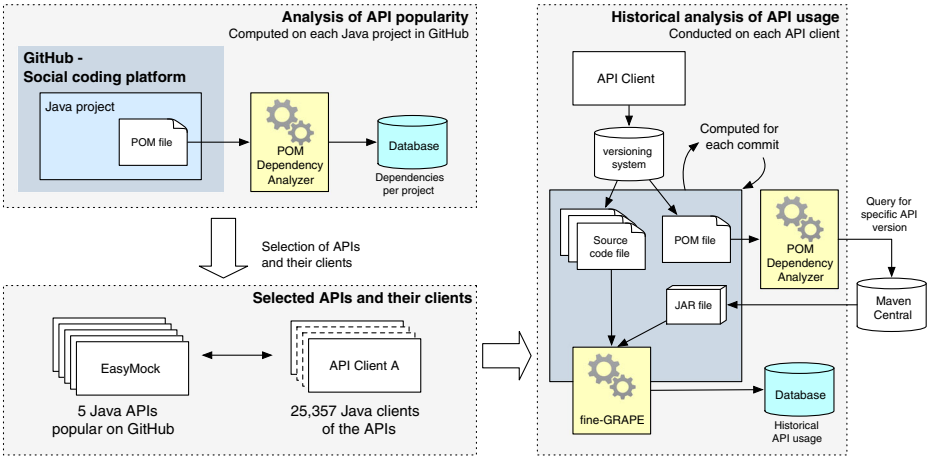


Fig. 1 Methodology used to mine data from Third-party API clients

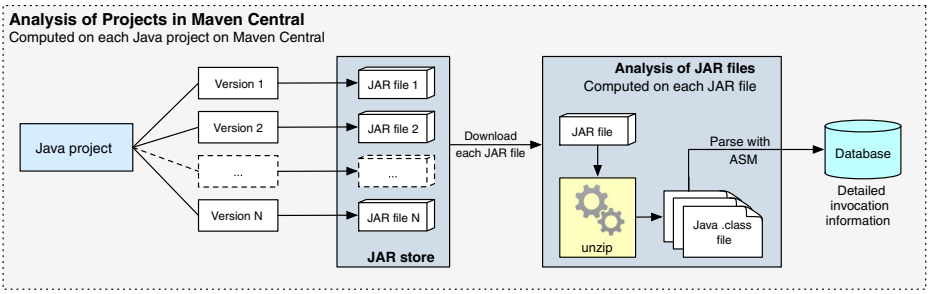


Fig. 2 Methodology used to mine data from JDK API clients

For the JDK, every Java project is essentially a client. We first select a diverse sample of Java projects from Maven Central to study. We then download successive compiled version of these systems from Maven Central (Fig. 2, left), before analyzing the bytecode to infer both the JDK version used to compile it, and the use of JDK features (Fig. 2, right).

2.2.1 System Source

The original study was conducted on the Squeak and Pharo ecosystems found on SqueakSource, thus the set of systems that were investigated was relatively small. To overcome this limitation, we focus on a mainstream ecosystem: Java projects hosted on the social coding platform GitHub and in Maven central. Java is the most popular programming language according to various rankings (http://www.tiobe.com/tiobe_index 2017; http://pypl.github.io 2017), GitHub is the most popular and largest hosting service (Gousios et al. 2014) and Maven Central is the largest store of JAR files (Raemaekers et al. 2014).

Third-party APIs (Fig. 1, top left). Our criteria for selection of APIs includes popularity, reliability, and variety: We measure popularity in terms of number of clients each API has on GitHub and select from the top 20 as identified by the fine-GRAPE dataset (Sawant and Bacchelli 2015). We ensure reliability by picking APIs that are regularly developed and maintained i.e., those that have at least 10 commits in a 6 week period before the data has been collected. 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.

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 projects on GitHub. By parsing their POM files, we obtained all the APIs 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, namely: Easymock (2016), Guava (2016), Guice (2016), Hibernate (2016), and Spring (2016). The first 6 columns of Table 1 provide additional information on these APIs.

JDK APIs. Clients of the JDK are not necessarily hard to find, GitHub alone contains 879,265 Java based projects. However, we are interested in accurately inferring the version of the JDK being used and whether these clients react to the deprecation of features in various versions of the JDK. This is not a trivial endeavor, given that Java source

Table 1 Summary information on selected clients and APIs

API				Unique entities		Number of	Usage across histories	
(GitHub repo)	Description	Inception	Releases	Classes	Methods	clients	Invocations	Annotations
EasyMock (easymock/easymock)	A testing framework that allows for the mocking of Java objects during testing	Feb 06	14	102	623	649	38,253	–
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	2310	14,828	3013	1,148,412	–
Guice (google/guice)	A dependency injection library created by Google	Jun 07	8	319	1999	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	2037	11,625	6038	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	5376	41,948	15,003	19,894	40,525

code files do not specify the version of Java that they are meant for. To overcome this challenge, we use the Java Archive (JAR) files of projects that were released to Maven Central.

Maven Central is the central repository for all libraries that can be used by Maven based projects. It is one of the largest stores of JAR files, where most small and large organizations release their source code in built form. JAR files consist of class files, which are a result of compiling Java source code. These class files contain metadata on the version of the JDK being used. Thus, making JAR files an appropriate source of data for JDK clients, given that version resolution can be done in it.

2.2.2 *Selection of Main Subjects*

We select the *main subjects* of this study:

Third-party APIs (Fig. 1, bottom left) To select the clients of APIs introducing deprecated methods, we use the aforementioned analysis of the POM files. We refine our process using the GHTorrent dataset (Gousios 2013), 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 1 provides an overview of the clients selected, by API.

JDK APIs (Fig. 2, left) At the time of data collection, Maven Central included 1,297,604 JAR files pertaining to 150,326 projects that build and release their code on the central repository. Analyzing all these projects would pose serious technical challenges, thus, we produced a selection criterium to reduce the number of projects, while preserving representativeness.

As a first step to filter our project list, we decide to eliminate those projects that have 4 or fewer releases on Maven Central. Projects that have had so few releases are not likely to have changed the version of the JDK that they use to compile their codebase, thus rendering them uninteresting to our current purpose. Performing this elimination step leaves us with 56,410 projects and 1,144,134 JARs that we can analyze. All these JAR files for each of the projects is downloaded, unzippped and all method invocations is stored in a database.

Despite having this amount of data, we cannot process all projects for the purpose of this paper, thus, as a further step of filtering we use the technique outlined by Nagappan et al. (2013) to sample our set of projects, creating a small and representative dataset of *diverse* projects to analyze. The technique allows for the creation of a diverse and representative sample set of projects that likely cover the entire spectrum of projects. They define coverage as:

$$coverage = \frac{|U_{p \epsilon P}\{q | similar(p, q)\}|}{|U|}$$

Here U is the universe of projects from which a selection is to be made. P is the set of dimensions along which these projects can be classified. A similarity score is computed for all projects based on the defined dimensions of the entire universe. The algorithm keeps adding projects to the subset that increases the similarity score until a coverage of 100% is achieved.

For our analysis, we define the following dimensions to model the projects:

- **Number of versions released:** The higher the number of versions released by a project, the more likely the project has made a change in the version of the JDK (or any other API) that it uses.
- **Median version release time:** Projects with a lower median release time, release new versions more often than those with a high inter-release period. This distinction is important because projects that release often might not actually react to deprecation due to the cost involved, on the other hand, projects with long inter-release time spans might have a lot of time at their disposal to fix their code.
- **Lifespan of the project:** Projects that have lasted a long time (e.g., Spring which has been around for 13 years) have more of a chance of having used multiple versions of Java and being affected by deprecation due to upgrading the version of the JDK being used, as opposed to those projects that are young.
- **Number of classes:** Projects that are larger (those that have more classes) might have a different reaction pattern to those that are smaller, measuring the number of classes allows us to make this distinction.
- **Starting version:** Projects that start with a more recent version of the JDK might not be affected by deprecation as opposed to those that use an old version of Java.

Nagappan et al. provide R scripts implementing their project selection technique. We ran these scripts with our dimensions of interest as input and collected a list of 60 diverse projects that cover the entire space of projects (100% coverage). We thus use these 60 projects for our analysis of client's reactions to deprecation in entities of the JDK API.

2.2.3 API Version Usage

Explicit library dependencies are rarely mentioned in Smalltalk. There are several ways to specify these dependencies, often programmatically and not declaratively (for instance, Smalltalk does not use import statements as Java does). Thus, detecting and analyzing dependencies between projects requires heuristics (Lungu et al. 2010). In contrast, Maven projects specify their dependencies explicitly and declaratively: We can thus determine the API version a project depends on. Hence, we can answer more questions, such as if projects freeze or upgrade their dependencies. This is more complicated in the case of JDK clients, as the version definition is not explicit. To overcome this flaw, we use an alternative data source (Maven Central) such that all information at our disposal can be considered accurate.

Third-party APIs. 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 because 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).

JDK APIs. (Fig. 2, right) Accurate resolution of the version of the Java API is a challenge on its own, since there is no easy way to find out the version of the JDK being used by a Java project by only inspecting the source code. However, there are three techniques that can be used and these are outlined below:

1. Projects that use maven sometimes use the maven compiler plugin in the POM file. This plugin allows the specification of the version of Java that is being used in the

source code and the bytecode version to which this source code is to be compiled. Often these versions can be the same, there is no hard requirement for them to be different. One can download all the POM files and parse them to find the usage of this plugin and see what version of Java is being used.

2. The Java JDK has evolved over time. With every new version, new features have been introduced. These features are incompatible with older versions of the JDK. However, these features are forward compatible, thus ensuring that anyone using that feature must either use the version of the JDK in which the feature was introduced or a later version. Thus, based on the language features being used one can ascertain a range of JDKs that may have been used to compile a certain file. Also, cross-referencing the date of usage of the feature with the date of JDK releases could allow us to narrow the range. Overall, this would give us a small range of versions that might have been used.
3. The previous two approaches both rely on parsing the source code to extract the usage of the Java features and to estimate the version of the JDK being used. However, the most reliable way to infer the version of Java being used is to look at the compiled class files of the source code. These class files contain a two-digit integer header that specifies the version of the JDK that was used to compile it, e.g., a class header could be 50, which implies that the class was compiled using JDK 1.6.

We tried the first method outlined above to resolve the version of the JDK being used. However, when we looked at 135,739 POM files that we managed to download from GitHub, we found that only 7722 files used the maven compiler plugin. In the larger context, this was a very small number of files that specified the version of the JDK. Thus, we found this option nonviable.

The second method can at times result in an inaccurate resolution of the version being used. This would make it hard to answer the research questions we have with the same accuracy as for the clients of the 5 Java APIs, hence we decided against using it.

Reading the compiled version of a class is the most accurate way to infer the version of the JDK being used by a Java project. However, this would imply that we would have to compile all the Java based project at our disposal. Even with these projects using the Maven build tool, this task is problematic. First, there is a chance that the dependencies specified in the POM file might relate to certain internal dependencies that are hosted by the project's developers and are not publicly available. Second, some of these POM files might use a PGP key to verify the authenticity of the dependencies being used, and this key is not available to us. Third, often the projects on GitHub do not ship with tests that work in all environments and they might need a certain testing environment to work properly, without which the Maven build fails. Fourth, it is very time-consuming to compile every version of every file to get its class file.

With the limitations of compiling GitHub based Java projects, we found an alternative source of Java based data; Maven central. Maven Central is the standard repository host for Java projects. Here developers release their projects as libraries such that others can use them. These projects are in the form of Java Archive (JAR) files that contain class files. We find this to be an appropriate source of data, given that there are 150,326 distinct projects released on Maven Central with 1,297,604 JAR files associated with them. These JAR files can be unpacked to parse the usage of Java features and at the same time ascertain the version of the JDK that was used to compile the source code.

2.2.4 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 a manual analysis of the reactions to an API change but had to discard cases which were too noisy.

Third-party APIs. (Fig. 1, right) 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.

While mining for API usage we must 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 (Dagenais and Hendren 2008) 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 (Sawant and Bacchelli 2015, 2016) that meets the following requirements:[2] (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.

JDK API. (Fig. 2, right) In the case of JDK clients, we look at class files that are retrieved from JAR files. These class files contain accurate API usage information. This information can be parsed using the ASM (2016) library, which uses the visitor pattern to visit a class file. For each class, we extract information on the version of the JDK being used, the annotations used in the class, and the method invocations made. For each of the method invocations, we have accurate information on the class that the method belongs to, the parameters and type of parameters being passed to the method, and what the expected return value is. Overall, we ensure that while parsing the JDK clients we obtain an accurate representation of usage of the JDK.

2.3 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.

[2]More details on fine-GRAPE can be found in our prior work (Sawant and Bacchelli 2016).

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 (2016) 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 stores 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 the scope of this study.

3 RQ0: What API Versions do Clients Use?

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

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 Fig. 3, we would count popularity as 1 for v7, 2 for v6, and 2 for v4. The column 'number of clients' in Table 1 specifies the absolute number of clients per each API and Fig. 4 reports the version popularity results, by API.

Third-party APIs. The general trend shows that a large number of clients use different versions of the APIs and that there is 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 result may indicate that clients have 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 updated or never updated their dependencies. In the example in Fig. 3, 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).

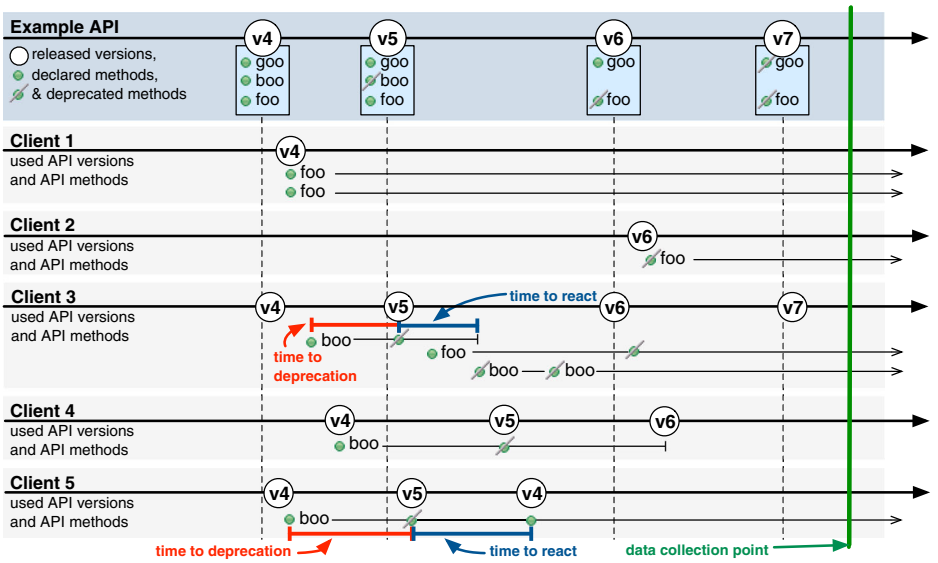


Fig. 3 Exemplification of the behavior of an API and its clients

Table 2 summarizes the results. Most clients freeze to one version of the API they use. 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, the median is lower for clients of APIs that

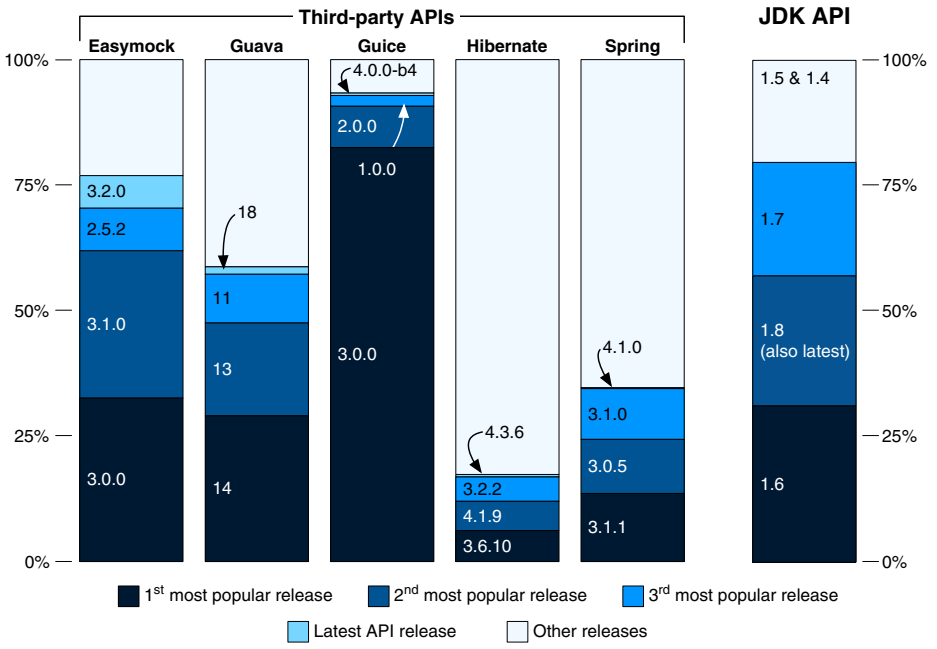


Fig. 4 Popularity breakdown of versions, by API

Table 2 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	1150
Hibernate	2454	41%	245	63	33	368
Spring	11,112	74%	195	69	37	186
JDK	19	33%	745	1507	996	1738

have more clients updating, such as Hibernate and Spring. In general, update time varies considerably—we will come back to this in RQ3.

JDK API. When doing this analysis, we notice one anomaly: two of the projects we analyzed have no bytecode associated with it. This is because the developers who released these projects to Maven central, released them only as Javadoc JAR files, without source code files. During our data collection process, we ensured that anything marked as Javadoc was not downloaded, however, in the case of these projects they were not marked as Javadoc but as source files. This seems to indicate that a non-negligible amount of Maven Central JAR files are not particularly useful or do not provide source code in some way. From this point forward, we discard the two projects made only of Javadoc and continue focusing on the 58 projects for which we do have the source code.

The clients are evenly distributed among versions 1.6, 1.7, and 1.8. Java 1.5 lags behind these three other versions, but despite being more than 13 years old, 11 clients adopt it. The number of clients using Java 1.6 is 18; despite that, there have been several years for Java clients to update to Java 1.7 (released in 2011) or Java 1.8 (2014).

Only 19 out of the 58 clients ever change the version of Java that is being used. The median time to update is almost 5 years; this is to be expected since Java has few major releases in any given timespan.

4 RQ1: How does API Method Deprecation Affect Clients?

Answering RQ0, we found 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 Fig. 3:

- *Unaffected:* These clients never use a deprecated method. None of the clients in Fig. 3 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 Fig. 3 belongs to this category.
- *Affected:* These clients use a method which is already in a deprecated state, but do not change the API version throughout their history. It happens in the case of Client 2.

- *Affected and changing version:* These clients use at least one method which gets deprecated *after* updating the API version being used. Clients 3, 4, and 5 belong to this category.

Figure 5 reports the breakdown of the clients in the four categories.

Third-party APIs. Across all third-party APIs, *most clients 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 discuss this in RQ3). Clients affected by deprecation vary from more than 20% for Easymock and Guava to less than 10% for Hibernate and almost 0% for Spring. Of these clients, less than one third also change their API version, thus highlighting a stationary 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, which 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.

71% and 65% of 'Affected and changing version' clients of Easymock and Guava react to deprecated entities. For Hibernate and Spring, we see 31% and 32% of clients that react. For all the APIs, the 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

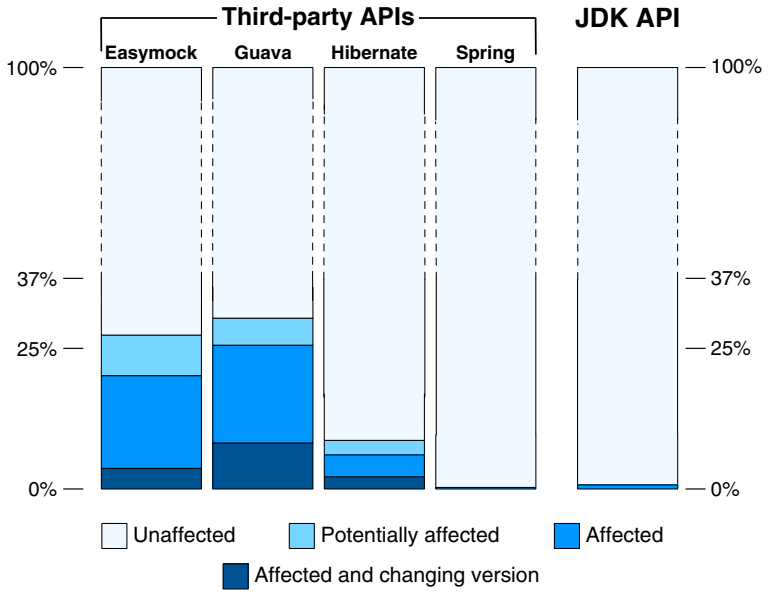


Fig. 5 Deprecation status of clients of each API

calls with other calls to the same API, however, this number is small. In contrast to what one would expect as a reaction to deprecation (due to the semantics of the deprecation warning), a vast majority of projects (95 to 100%) add calls to deprecated API elements, despite the deprecation being already in place. This concerns even clients that migrate all their deprecated API elements later on.

> **The strange case of Guice.** We analyzed all the Guice clients and looked for usage of a deprecated annotation or method, however, we find that *none of the projects* have ever used deprecated entities. The reason is that Guice does not have many methods or annotations that have been deprecated since it follows a very aggressive deprecation policy. In Guice, methods are removed from the API without being deprecated previously. We observed this behavior in the Pharo ecosystem as well and studied it separately (Hora et al. 2015). In our next research questions (RQ2–RQ5), thus, *we do not analyze Guice*, as the deprecations are not explicitly marked. However, we do not remove Guice from our study to keep it organic and contrast the deprecation policy it uses with that of other APIs in RQ6.

JDK API. We see a surprising trend in the case of the Java clients, as also reported on the rightmost bar in Fig. 5. Only 4 out of 58 projects are affected by deprecation. Also, none of these projects are among those that change the version of the API that they use; this implies that at the time of usage of the deprecated feature, the client already knew that it was deprecated.

5 RQ2: What is the Scale of Reaction in Affected Clients?

The work of Robbes et al. (2012) 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. Hence, we can 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 must react to: It seems reasonable to think that adapting to 10 usages of the same deprecation is easier than reacting to 10 usages of 10 different deprecated methods.

Third-party APIs. We consider both 'actual reactions' and 'potential ones'. **Actual reactions.** We measure the scale of the actual reactions to API changes. We count separately reactions to the same deprecated method and the number of single reactions. In Fig. 3, 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 3 reports the results. The first column reports the absolute number of non-frozen affected clients that reacted. The scale of reaction varies: Most of the clients react to less than a dozen of statements with a single unique deprecated method involved. Spring stands out with 31 median number of statements with reactions and 17 median

Table 3 Scale of actual clients' reaction to method deprecation

	Non-frozen affected clients that reacted	Statements with reaction (unique deprecated methods involved)			
		Median	Q3	95th perc.	Max
Easymock	17	11 (1)	21 (2)	109 (3)	(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)

number of *unique* deprecated methods involved. Outliers invest more heavily in reacting to deprecated methods. As seen next, this may explain the reluctance of some projects to update.

Potential reactions. Since a large portion of projects do not react, we investigated 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 to render their code base compliant with the latest version of the API (i.e., removing all deprecation warnings). In Fig. 3, the only client that is potentially affected by deprecation is client 1, which would have two statements needing reaction (i.e., those involving the method 'foo') and one unique deprecated method is involved.

As before, we focus on the upper half of the distribution. Table 4 reports the results. In this case, the first column reports the absolute number of clients that would need a reaction. We notice that most of the 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 3, 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 lot 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, especially if these APIs start removing the deprecated features. 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 have several unique methods to adapt to.

JDK API. We consider only 'potential reactions' in the case of the JDK API, as there can be no 'actual reactions'. We analyzed all the 58 clients, fast-forwarding them through all the JDK APIs version to see whether they would be affected by deprecation, should they

Table 4 Scale of potential clients' reaction to method deprecation

	Clients potentially needing reaction	Statements potentially needing reaction (unique deprecated methods involved)			
		Median	Q3	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)

upgrade. We found that *none* of these clients would be affected in the event that they would all upgrade to the latest version of the JDK. In other words, none of the analyzed clients use features that have been deprecated in newer versions of Java.

Overall, reflecting on the reasons why deprecation has almost zero impact on these clients (answers to RQ0, RQ1, and RQ2), we can hypothesize that the tight integration of the JDK API in the most popular IDEs and the amount of documentation available to aid a developer in selecting the most appropriate API feature play a role on the facts we encountered. For the purpose of this study, we cannot continue with the other research questions (except for RQ6, in which we cluster the API deprecation behavior of JDK API) given that there is no reaction data to be analyzed; however, we discuss (Section 11.2) on why we found such a low number of clients affected by deprecation in the case of the JDK API.

6 RQ3: What Proportion of Deprecations does Affect Clients?

The previous research question shows that most of the actual and potential reactions of third-party API clients to method deprecations involve 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 Fig. 3, 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 5 summarizes the results, including the proportion of deprecated methods per API over the total count of methods and the count of how many of these deprecated methods are used by clients. APIs such as for Guava, Spring, or Hibernate have more than 1,000 deprecations. For Hibernate, 65% of unique methods get eventually deprecated, indicating that this API makes a heavy usage of this Java feature. The proportion of deprecated methods that affect clients is, around 10% in all 4 of the APIs.

7 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 period of time developers take to react to it ('time to react') to see if developers react as soon as they notice that a feature they are using is deprecated. The former is defined as the interval between the introduction of the call and when it was deprecated, as

Table 5 Deprecated methods affecting clients, by API

	Unique deprecated methods	
	Defined by API Count (% over total)	Used by clients Count (% over all deprecated)
Easymock	124 (20%)	16 (13%)
Guava	1479 (10%)	104 (7%)
Hibernate	7591 (65%)	487 (6%)
Spring	1320 (3%)	149 (11%)

seen in client 3 (Fig. 3); 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: 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 entity, that they know a priori that the entity is already deprecated. This seems to indicate that clients do not mind using deprecated features.

Time to React Figure 6 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 0 days for Guava, Hibernate, and Spring, while for Easymock it is 25 days. A reaction time of 0 days can indicate that most deprecated method invocations are reacted upon on the same day the call was either introduced or marked as deprecated. To confirm this, we looked a little deeper at 20 of these cases to see why the reaction time is 0 days. We see that in all of the cases the developers have upgraded a version of the API they use, this leads to them noticing that a feature they use is now deprecated. They react to this deprecation immediately after the upgrade in version, thus resulting in a reaction time of 0 days.

Barring outliers, reaction times for Hibernate and Spring are in the third quartiles, being at 0 and 2.5 days. Reaction times are 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. We looked individually at the 9 outliers that have a reaction time in excess of 300 days. Distilling the actual rationale behind a change is non-trivial. We look at the commit messages and any kind of code comments that might exist. Only one commit

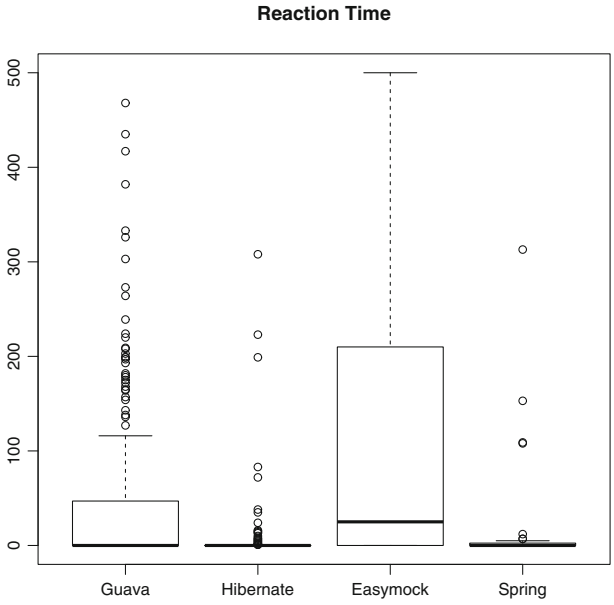


Fig. 6 Days taken by clients to react to a method deprecation once visible

message actually references the fact that a deprecated entity was being reacted to. The other 8 commit messages do not add any information. We also do not see any code comments that might explain the rationale. We can at best speculate that the reaction to deprecation takes place as part of a general code cleanup act.

8 RQ5: Do Affected Clients React Similarly?

This research question seeks to investigate the behavior of third-party API clients when it comes to replacement reactions.

Such an analysis allows us to ascertain whether an approach inspired by Schäfer et al.'s (2008) would work on the clients in our sample. Their approach recommends API changes to a client based on common, or systematic patterns in the evolution of other clients of the same API.

8.1 Consistency of Replacements

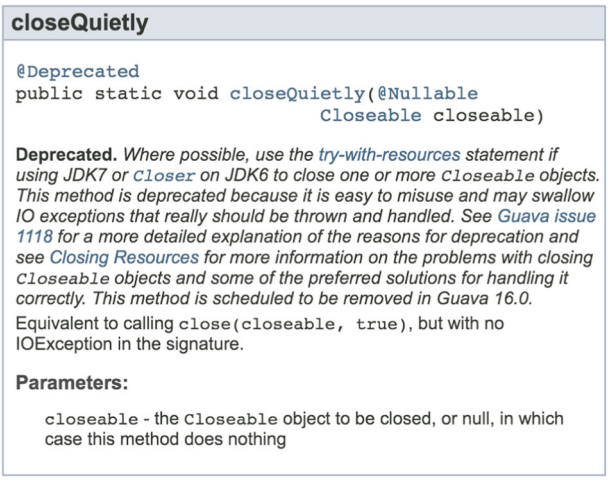
There is no definite way to identify if a new call made to the API is a replacement for the original deprecated call, so we rely on a heuristic: We analyze the co-change relationships in each class file across all the projects; if we find a commit where a client removes a usage of a deprecated method (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 clients show no systematic transitions: There are only 3 distinct methods for which we see 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 the case of Hibernate clients, 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 clients, 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.

8.2 Quality of documentation.

Very few clients react 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 Fig. 7. However, the results we obtained made us hypothesize otherwise. We systematically inspected the Javadoc to see whether deprecated features had documentation on why the feature was deprecated and whether there was an indication of appropriate replacement (or if a replacement is needed).



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

Fig. 7 Example of Javadoc associated with deprecated API artifact

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 an impact on clients; for Spring and Hibernate, we inspected a 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 use 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 of which state that the method is no longer needed and hence can be safely deleted, and 5 deprecated methods do not have a message. Guava 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 provide 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 “This method is scheduled for deletion in June 2013.”).

For Hibernate, all the messages provide a replacement, but most provide no rationale for it. 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 provide their clients with sufficient support to clients concerning deprecation. We found rationales as to why a method was deprecated, but not systematically. Replacement is the most commonly suggested solution; this is in contrast

to the actual behavior of clients who instead prefer removing references to deprecated entities as opposed to replacing them, as reported in Section 4.

9 RQ6: How are Clients Impacted by API Deprecation Policies?

During this study, we noticed that each API has its own way to deprecate features. It seems reasonable to think that this deprecation policy of features may impact a clients' decision to adopt and react to these deprecated features. We thus decided to look at this particular issue in more detail.

9.1 Methodology

To see what different kind of policies of deprecation exist, we first aimed to look at the top 50 APIs that are popularly used by GitHub based Java projects, to get a sufficiently diverse set of APIs, with a sufficient number of clients that may react differently. However, looking at only the top 50 most popular APIs might have one downside: Given that many of the APIs have the same vendor, the deprecation policy adopted by the APIs from the same vendor may be similar. To overcome this limitation, we looked at the top 200 APIs in terms of popularity (ranked based on usage of the API among Java projects on GitHub) and selected the first 50 that had different vendors, this resulted in the APIs listed in Table 6.

Once the APIs had been selected we defined certain criteria to categorize their deprecation behavior on:

Number of deprecated: Number of unique features deprecated during the entire history of the API. The larger the number of features that are deprecated by an API, the larger the chance is that a client using that API will be affected by deprecation.

Percentage of deprecated: Percentage of total features deprecated during the entire history of the API. When an API deprecates a large portion of its features, there is a higher chance that it might deprecate features that are being used.

Time to deprecate: Median time, in days, taken to deprecate a feature from the moment it was introduced in the API. A long time to deprecate can indicate that API developers do not change their API at a fast pace. This fact can be reassuring to clients who are ensured the stability of the API and its features.

Time to remove: Median time, in days, taken to remove a deprecated feature after the moment at which it was deprecated. A short removal time gives API clients a very short window within which they can react to the deprecation of the feature, after which the change becomes a breaking change. A longer removal time may indicate that the API does not perform regular cleanup of its code.

Rollbacks: Number of times a deprecated method was marked in a future release as non-deprecated. A rollback may be performed because the API developers changed their mind about deprecating a feature. This would send a confusing signal to the API client as they cannot be sure about the future of the feature that they are using. Ideally, this behavior should be avoided, because it gives no clear indication about the future of a feature.

Percentage of removed: Percentage of deprecated features eventually removed from the API. A high percentage suggests that the API performs a lot of cleanup of its deprecated features. On the other hand, a low percentage indicates that the API is lax about removing deprecated features, thus allowing clients to assume they do not have to react to deprecation.

Table 6 List of 50 APIs selected for RQ6

API Artifact ID	Domain	Popularity	Rank
junit	Testing	67,954	1
slf4j-api	Logging	18,521	2
log4j	Logging	17,421	3
spring-core	Dependency injection	15,086	4
mysql-connector-java	Database	14,333	6
servlet-api	Server	12,044	8
jstl	Server	12,007	9
commons-io	IO utility	10,821	12
guava	Collections	9542	14
hibernate-entitymanager	Object relational mapper	8413	16
logback-classic	Logging	7597	20
mockito-all	Testing	7010	22
commons-lang	Utility	6485	26
jackson-databind	JSON handling	5905	28
httpclient	Server	5584	32
commons-dbcp	Database	5486	34
joda-time	Time utility	5045	36
commons-logging	Logging	4947	37
aspectjrt	Aspect oriented	4685	38
testng	Testing	4485	40
commons-codec	Codec utility	4337	41
commons-fileupload	Fileupload utility	4001	45
h2	Database	3952	46
postgresql	Database	3816	47
hsqldb	Database	3633	49
validation-api	Bean validation	3509	52
commons-collections	Collections	3406	54
json	JSON handling	2952	56
hamcrest-all	Testing	2867	57
cglib	Bytecode generation	2816	60
selenium-java	Browser	2694	61
lombok	Eclipse	2472	65
javassist	Bytecode generation/manipulation	2466	66
jsoup	HTML parser	2333	67
mybatis	Database	2199	72
standard	Tagging library	2150	74
commons-beanutils	Java beans	2147	75
mongo-java-driver	Database	2077	78
poi	File format manipulation	1940	83
commons-cli	Command line utility	1855	84
jersey-client	Server	1844	85
dom4j	HTML parser	1798	87
c3p0	Database	1782	88

Table 6 (continued)

API Artifact ID	Domain	Popularity	Rank
commons-httpclient	HTTP Client	1676	91
primefaces	UI building	1608	95
commons-pool	Object pooling	1583	96
guice	Dependency injection	1556	97
freemarker	Java beans	1531	98
assertj-core	Testing	1527	99
easymock	Testing	1484	100

Number of never-removed: Number of deprecated features that were never removed from the API and that are still present despite being deprecated. An API leaving a lot of its deprecated features never-removed may signal that the client should not worry about their code breaking in the near future. Ideally, this number should be quite high given that the traditional pattern of deprecation is to first deprecate a feature and then after an interval remove it from the API.

Average deprecations per version: Average number of features deprecated per version of the API. A high average number of deprecations per version tells us that the API is very volatile, which might factor into a client's decision on upgrading the version of the API being used.

Average removals per version: Average number of deprecated features removed per version of the API. If the removals per version are high, then this adversely impacts a client's decision to change the version being used as making a change ensures that their code would break.

9.2 Clustering

Using these dimensions, we can run a clustering algorithm on the APIs to see whether clusters emerge and their nature. The most widespread clustering algorithm which fits our model is *k*-means (Lloyd 1982). The *k*-means clustering algorithm aims to partition a set of elements into *k* clusters where each element in the set belongs to a cluster with the nearest mean. One issue with this clustering technique *which is unsupervised*, is the estimation of the number of clusters to be produced. To do so we choose to use the elbow method (Mooi and Sarstedt 2010), that allows for a visual estimation of the value of *k*.

The elbow method looks at the percentage of variance explained as a function of the number of clusters. After a point, adding more clusters in the *k*-means algorithm should not result in a better estimation of the data. Using this technique, we calculated the sum of squares within each cluster for each number of clusters (where the number of clusters varied from 1 to 15) provided to *k*-means algorithm and plotted these sums. Looking at the plot, we determined that the number of clusters that best describes our data is 7. Using this value as our input for the *k*-means algorithm, we could establish what the clusters are, what are the characteristics of these clusters, and which APIs belongs to them.

In Fig. 8, we see a silhouette plot of the clusters that we have obtained from our data. A silhouette plot essentially provides a succinct graphical representation of how well each object lies within its cluster. The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette ranges

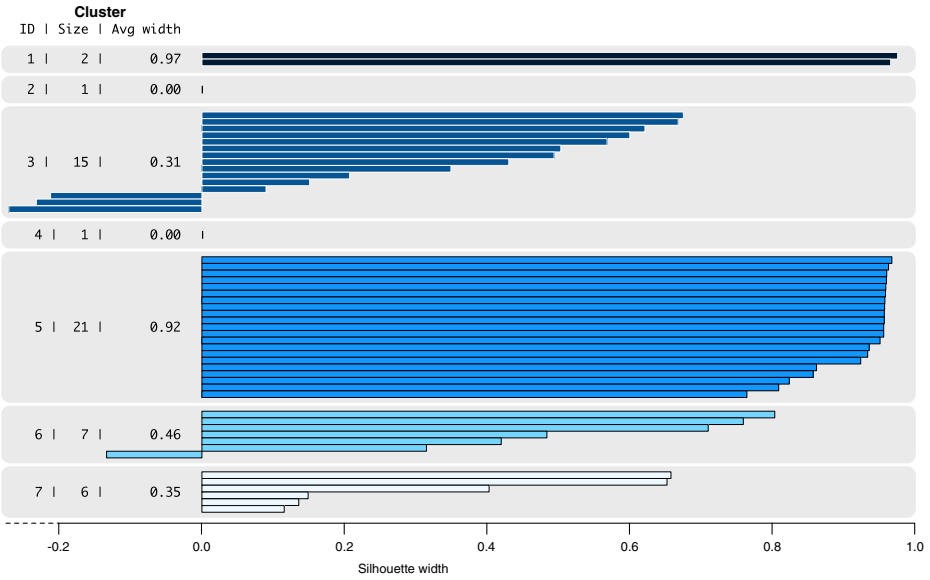


Fig. 8 Silhouette plot of the 7 clusters

from −1 to 1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters. If most objects have a high value, then the clustering configuration is appropriate.

Looking at Fig. 8 we see that our clusters are separate from each other. In most cases, the silhouette values are positive and high. There are only 4 negatives in each cluster, thus indicating that most objects in each cluster are matched with the others. We see that there are two clusters with just one API each (Apache commons-collections and Java), these APIs appear to be outliers in our dataset, in particular, this was expected in the case of the JDK API.

9.3 Results

We use the dimensions listed above to characterize an API's policy of deprecating its features. The goal is to create a categorization of the APIs that are under consideration. Analyzing the clusters, we derive the defining characteristics of a project that falls in each cluster, and how the clients of such a project might potentially react. A summary of this can be found in Table 7. To not bias ourselves, we choose to not look at the APIs that we have already studied so that we do not allow our previous results to dictate the properties of each cluster. In the following we describe the characteristics of each cluster and discuss their potential implications:

Cluster 1: In this cluster (2 elements), the deprecation times and the removal times are both more than 10 years, as opposed to cluster 7 where the deprecation and removal times were less than 3 years. This implies that in this case when the APIs in this cluster deprecate or remove a feature, it is often those features that were introduced early in the API that are affected. However, these APIs also

Table 7 Summary of clusters

Cluster	Characteristic	Number of APIs
1	Very high deprecation time and removal time	2
2	Large portions of the API are deprecated	1
3	Deprecated features are removed with urgency	15
4	Deprecate very little and take a very long time to do so	1
5	Deprecate a lot at the time of introduction, most likely for experimental features	21
6	Deprecates a lot of features, many of which that are not too old	7
7	Features are not deprecated or removed very easily	6

deprecate very few features in total and less than 6 features per version. Thus, the scope of a client being affected by deprecation is minimal at best.

Cluster 2: This cluster has only one element. This API (Apache Commons-collections) is the one out of all 50 APIs that deprecates the largest percentage (30%) of its API in its history. The median times for deprecation and removal are also really low for this API, and in many instances, features are introduced in a deprecated state. This API is also very good at cleaning up its code base by removing 83% of all deprecated features at some future release. Given the large usage of deprecation in this API, we can conclude that clients have to be wary about what feature they use.

Cluster 3: The APIs that fall into this cluster (15 elements) usually have a high median time to deprecate a feature (more than 2 years), but on the other hand have a short period to remove the feature (less than 30 days). This gives a client limited opportunity to react to a deprecation as the deprecated entity will likely be removed in an upcoming release. This might influence a client to not upgrade the version of the API being used.

Cluster 4: There is just one element in this cluster: The JDK API. This cluster is characterized by the fact that a very small percentage of the API is deprecated (less than 1%), and for the features that have been deprecated, there are 621 rollbacks in deprecation in its lifetime. The median time to deprecate is less than 3 years, and in the event there are removals of a deprecated entity, then the time to remove is very long, more than 6 years. The number of deprecations per version is high in comparison to other clusters, so is the number of features removed per version. Given that a very small percentage of the API is deprecated and the removal time being high, we suppose that clients of this API are not affected by deprecation to a large extent and those that are affected do not react at all.

Cluster 5: In this cluster (21 elements), we see the most number of APIs. All these APIs in general do not deprecate a large percentage of their APIs. However, when they do, it is generally done immediately after the introduction of the feature. This generally affects only those features that have been introduced later in the API's life. Thus, only those clients that adopt the latest feature are going to be the ones that are affected to a great extent. But given the fact that very few features are deprecated, we assume the number of affected clients to be very low for these APIs.

Cluster 6: In this cluster (7 elements), the APIs have a low median deprecation time and median removal time both under 40 days. At the same time, the projects in this cluster deprecate quite a lot of methods per version. Many features that are new and introduced in one release are generally deprecated and rendered obsolete in the immediate future. This kind of behavior may discourage a client from adopting new features of the API, due to the fear of being forced to update in the near future. Thus, for the APIs here we should see minimal clients affected by deprecation as most might not even adopt the features that get deprecated.

Cluster 7: This cluster (6 elements) is characterized by median removal and deprecation times higher than 3 years, thus giving the clients of the APIs in this category the safety of not having to worry about their code breaking in any manner. This policy may not incentivize clients of the API to react to a deprecated feature.

Based on the cluster definitions we make the following hypotheses, which can be tested in future work:

Hypothesis 1: Clients of cluster 1, 4 and 7 do not react to deprecated entities as opposed to clients of APIs that belong to other clusters.

Hypothesis 2: Clients of cluster 3 will react to deprecation with a low reaction time, otherwise, their code will break.

Hypothesis 3: Clients of cluster 6 will not adopt newer features since these features are the ones that are generally deprecated.

Hypothesis 4: Clients of cluster 5 will not be affected by deprecation due to the fact a lot of features are experimental and marked as deprecated due to this.

Hypothesis 5: Clients of cluster 2 will be affected by deprecation regularly due to the fact that the API deprecates a lot.

In Table 8 we see in the category into which each of the APIs that we study fits into. In the case of Guava and Hibernate, both fall into cluster 3. We expect to see their clients not upgrading to the latest version of the API in most cases. This is reflected in the results of RQ0 as well, where we see that both for Guava and Hibernate the latest releases has very minimal adoption, whereas older releases have been adopted to a much larger extent.

Spring fits into the seventh cluster, where the clients are not expected to be impacted by deprecation. We see that reflected in the results of RQ1, where Spring has the least

Table 8 API deprecation characteristics

API	Spring	Hibernate	Guava	Easymock	Java
Number deprecated	551	110	529	95	1910
Percentage deprecated	11%	1%	3%	4%	1%
Time to deprecate	1454	1043	620	0	927
Time to remove	1456	698	228	1557	3456
Rollbacks	10	0	47	2	629
Percentage removed	45%	100%	60%	90%	59%
Number unremoved	295	0	172	7	152
Average deprecations per version	4	0	11	7	239
Average removals per version	2	0	6	6	141
Cluster	1	2	2	3	7

number (and percentage) of clients affected by deprecation. This is to the credit of the Spring developers who have adopted a policy that does not have any adverse impact on its clients.

Easymock falls into cluster 6, and here we expect to see a minimal number of clients to be affected, given that only new features are deprecated, whereas the ones that were originally introduced are not necessarily deprecated by the API. However, we see from RQ1 that there are many clients affected by deprecation. This might indicate that it is not only new features that are being deprecated in Easymock. This result makes Easymock an imperfect fit for the cluster. Looking at Fig. 8, this does indeed appear to be the case, all elements in cluster 6 do not appear to fit with each other to make it a homogeneous cluster. Given that we did not allow our cluster definition to be defined by the data that we already had in place, it is to be expected that for some of these APIs, the cluster fit would be imperfect.

10 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 including the JDK API, 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. For Guice, we could find no data due to Guice's aggressive deprecation policy. In the case of the JDK API, we found very few clients to be affected, but none of them changed versions, thus we could not analyze their reaction data. 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 these 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.

As a final step, we looked at how the different APIs deprecate their features and how such a deprecation policy can impact a client. We clustered 50 APIs based on certain characteristics (such as the number of deprecated API elements, and the time to remove deprecated API elements), and documented the patterns that emerged in seven clusters. For each cluster, we define its primary characteristic and predict the behavior of a client that uses an API that belongs to the clusters, leading to five hypotheses that can be confirmed or infirmed in future work. We see that in the case of Guava, Hibernate and Spring clients our clusters fit perfectly. However, in the case of Easymock, the fit is not as good. This suggests that further investigation is needed in this case.

11 Discussion

We now discuss our main findings and contrast them with the findings of the Smalltalk study we expand upon. Based on this, we give recommendations on future research directions.

11.1 Comparison with the Deprecation Study on Smalltalk

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

11.1.1 Proportion of Deprecated Methods Affecting Clients

Both studies found that only a small proportion of deprecated methods affects clients. In the case of Smalltalk, this proportion is below 15%, but 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 (Begel and Zimmermann 2014), the functionalities API developers deprecate are mostly unused by the clients, thus deprecation causes few problems. Nevertheless, this also suggests that most 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.

11.1.2 Not Reacting to Deprecation

Despite the differences in the deprecation mechanisms and warnings, most 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., one Hibernate client would have to change 17,471 API invocations).

One reason behind the non-reaction to deprecation might be that some of these deprecated entities find themselves in dead-code regions of API client code as opposed to essential parts. This might impact the client's decision to react. However, the impact of this is hard to determine given that the cost of executing thousands of API clients in representative execution scenarios is prohibitive (assuming it is even possible in the first place).

We see that in many cases the preferred way to react to deprecation is by deleting the invocation. This reaction pattern might be due some APIs advising that the deprecated feature need not be used anymore and can be safely deleted with no replacement. The impact of this might be quite high given our findings in Section 8.

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.

11.1.3 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.

11.1.4 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 Section 11.4 we detail some of the first steps in this direction, clearly emerging from the findings in our study.

11.2 Comparison Between Third-Party APIs and the JDK API

We observe that deprecations in the JDK do not affect the JDK clients to a large degree. Only 4 out of 58 projects are affected by deprecation and all these 4 introduced a call to the deprecated artifact despite knowing that it was deprecated. Such a low proportion of JDK clients being affected was an unexpected finding, we rationalize it with the following hypotheses:

11.2.1 Early Deprecation of Popular JDK Features

Some of the more popular or used features of the JDK that have been deprecated, were deprecated in JDK 1.1 (e.g., the Java Date API). For these features, replacement features have been readily available for a long time. As a sanity check, we looked for the usages of the `Date` class in our database on API usages that was mined from GitHub based data. There we see that only 47 projects ever use this class out of 65,437 Java based projects. This indicates that almost all clients already use the replacement features instead of the features that have been deprecated a long time ago.

11.2.2 Nature of Deprecated Features

Manually analyzing the list of features deprecated in the JDK, we found that many of these features belong to the `awt` and `swing` sub-systems. Both these sub-systems provide GUI features for developers. The nature of the projects hosted on Maven Central is such that most of these projects do not provide a graphical interface as they are, in most cases, intended to be used as libraries. Nevertheless, the analysis of the 65,437 GitHub clients also shows the same behavior, thus mitigating the risk of a sample selection bias. Other than just GUI features, the JDK also has internal features and security features that have been deprecated. These are not intended for public use, hence, we do not see these among the projects that we investigate.

11.2.3 Nature of Projects

Our dataset contains all the projects from Maven Central. The fact that a project is released in an official site such as Maven Central indicates that high level of adherence to Software engineering practices among its developers. Given that these projects are in the public eye, and free for all to reuse, developers of these projects must have made every effort to ensure high code quality. This might have resulted in us seeing such low usage of deprecated features. Moreover, our dataset contains information on over 56,000 projects, and for each project, we have data on each release. However, we do not have any information at commit level. This might prevent us from detecting real time usage of a deprecated artifact and any reaction that might take place. All usages of deprecated features might have been taken out by the time the release is made to Maven Central. Thus, we might miss some deprecation information. Nevertheless, results from the 65,437 GitHub clients are in line with the findings from Maven Central.

11.2.4 Documentation of the JDK

The JDK API is the best-documented API out of the ones that we have studied in this paper. They have detailed reasons behind every deprecation, thus allowing a developer to make an informed choice on reacting to the deprecation. This documentation also mentions the replacement feature that should be used in the event that a developer would like to react to the deprecated feature. Java is also one of the most popular languages in the world (http://www.tiobe.com/tiobe_index 2017; http://pypl.github.io 2017), thus leading to the generation of a large amount of community-based documentation (e.g., Stackoverflow, blog posts, and books) that provide a developer with every aid imaginable to use the Java API in the right manner. Also, Java is one of Oracle's most important projects and the company ensures that there are plenty of programming guides available on its own website. This amount of developer support could be one of the reasons why we see very few projects who are affected by deprecation.

11.2.5 Deprecation Policy in JDK

The Java developers have made a commitment to not removing deprecated features in any current or future release (http://www.oracle.com/technetwork/java/javase/compatibility-417013.html#incompatibilities 2017). However, the JDK developers recommend removing all the deprecated features as soon as possible. The main reason they keep deprecated features is to ensure backward source code compatibility with previous versions. This does not act as an incentive for a developer to change the version of the JDK being used, hence, it might result in fewer projects changing the JDK version and being affected by deprecation.

11.3 Impact of Deprecation Policy

We see in RQ6 that different APIs deprecate their features in different manners. They all differ in terms of time taken to deprecate a feature or remove a feature or the number of features that are removed from the API. Based on different characteristics that we define, we find that we can cluster 50 APIs into 7 distinct clusters, each with their own defining characteristic.

We see that in the case of Guava, Spring, and Hibernate, the clients react as predicted by the clusters in which these APIs found themselves. However, in the case of Easymock, we do not see the expected behavior among its clients. This tells us that the clusters are not perfect in every case and may have to be expanded upon by studying more APIs. However, what we do see is that the deprecation policy adopted by an API does indeed have an impact on its clients, thus providing API developers with an insight into how the deprecation policy they adopt affects a client and what policy they should adopt in the event they want to minimize the impact on their client.

11.4 Future Research Directions

Below we enumerate a couple of promising future lines of research worth pursuing:

11.4.1 If it ain't Broke, don't Fix it

We were surprised that so many projects did not update their API versions. Those that often do it slowly, as we saw in the cases of 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 despite 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 for these choices, thus indicating how future implementations of deprecation can give better incentives to clients of deprecated methods.

11.4.2 Further Investigating the Deprecation Polices

We see that different APIs do actually adopt different deprecation strategies and this appears to have an impact on the clients of these APIs. However, we have been only able to discuss this for the APIs in our analysis. As a further step, one could assess the impact of deprecation policies on the clients for all the APIs that were used to make the clustering. This would reinforce the idea of deprecation policies and their impact on a client.

11.4.3 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.

11.4.4 Volume of Available Documentation

In the case of popular APIs such as the JDK API or JUnit, there is a large amount of documentation that is available. This might impact the reaction pattern to deprecation given that there is likely some document artifact that addresses how to react to a deprecated entity. On the other hand, for smaller or less popular APIs which do not have as much community documentation or vendor based documentation, support for the developer might not be available. Overall, the volume of API documentation might impact the reaction pattern to deprecation, a fact that warrants future investigation.

11.5 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 have some innacuracies. 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. (2014). 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 Kalliamvakou et al. 2014); 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.

Projects on Maven central need not always follow the same name. There are occasions where a project has renamed itself to another artifact ID or to another group ID. There is no automated way to keep up with these renames, as this information is unavailable on Maven central. Due to this, for certain project we might miss out on their evolution and thus might missclassify their reaction pattern w.r.t JDK deprecations.

The projects on Maven Central were expected to adhere to semantic versioning when releasing different versions of their projects. However, previous work by Raemaekers et al. (2014) shows that only 50% of the projects on Maven Central use the versioning system in the right way. Due to this, we can not and did not rely on version numbers of the various projects under study to ensure that they are ordered in the right way when we analyze their evolution. To overcome this we use the release date as a way to order the versions. However, this might not be accurate given that in some cases a relase pertaining to major version 3 might be made after a release to major version 4. This might impact the accuracy of our results, however, a manual inspection showed that the former case happens in a very small percentage of cases, thus the impact on our results appeared to be negligible.

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 (Chacon 2009). However, we may be missing reactions to API deprecations that have not yet been merged in the main branch.

12 Related Work

12.1 Studies of API Evolution

Several works study or discuss API evolution, the policies that regard it, and their impact on API developers and clients.

When it comes to an API, one of the first decisions is what to leave out. Many projects have so-called *internal APIs*, that, despite being publically accessible, are reserved for internal usage by the project (Businge et al. 2013a). They are not intended to be used by clients and may change without warning from one release to the next. Businge et al. (2013a) found that 44% of 512 Eclipse plugins that they analyzed used internal Eclipse APIs, a finding echoed in a larger study by Hora et al. (2016), that found that 23.5% of 9,702 Eclipse client projects on GitHub used internal APIs. Shedding more light on the issue, a survey by Businge et al. (2013b) found that 70% of 30 respondents used internal APIs because they couldn't find a public API with the equivalent functionality, and re-implementation of the functionality would be too costly. Hora et al. (2016) observed that some internal APIs were later promoted to public APIs, and presented an approach to recommend internal APIs for promotion. These findings agree with our study, in that they also show that maintainability often takes a back to functionality, a fact reflected in the unwillingness to update APIs, which can lead to considerable technical debt.

Studies closely related to this paper (i.e., Robbes et al. 2012 and Hora et al. 2015), that deal with deprecation policies of APIs and their impact on API clients have been performed on the Pharo ecosystem. The first study focused on API deprecations and their impact on the entire Pharo ecosystem. The second study focused on API changes that were not marked as deprecations beforehand. They look at the APIs policy to change features and the impact these changes have on the client.

Brito et al. (2016) analyze deprecation messages in more than 600 Java systems, finding that 64% of deprecated methods have replacement messages. This implies that API clients are provided with support when reacting to deprecation.

While neither Brito's study nor ours look extensively into the reasons why API elements are deprecated, other works did. Hou and Yao (2011) studied release notes of the JDK, AWT and Swing APIs, looking for rationales for the evolution of the APIs. In the case of deprecated API elements, several reasons were evoked: Conformance to API naming conventions, naming improvements (increasing precision, conciseness, fixing typos), simplification of the API, coupling reduction, improving encapsulation, or replacement of functionality. Of note, a small portion of APIs elements was deleted without replacements. Our manual analysis of deprecation messages found that was also the case in the APIs that we studied. We found relatively few rationales for deletion of API elements, with the most common reason being that the method concerned was no longer needed.

The versioning policy adopted by an API might give an API client an indication as to what kind of changes could be expected in that version. To that end, Raemaekers et al. investigated the relation among breaking changes, deprecation, and the semantic versioning policy adopted by an API (Raemaekers et al. 2012). 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.

The evolution policy of Android APIs has been extensively studied. McDonnell et al. (2013) investigate stability and adoption of the Android API on 10 systems; the API changes are derived from Android documentation. They found that the Android API's policy of evolving quickly leads to clients having 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 (Linares-Vásquez et al. 2014), not API clients directly. Bavota et al. (2015) 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.

Web based API evolution policies have also been studied. Wang et al. (2014) 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 web APIs, we find for example the one by Espinha et al. (2015), who study 43 mobile client applications depending on web APIs and how they respond to web API evolution.

APIs also break. Dig and Johnson studied and classified the API breaking changes in 4 APIs (Dig and Johnson 2006); they did not investigate their impact on clients. They found that 80% of the changes were due to refactorings. Cossette and Walker (2012) studied five Java APIs to evaluate how API evolution recommenders would perform in the cases of API breaking changes. They found that all recommenders handle a subset of the cases, but that none of them could handle all the cases they referenced.

In a large-scale study of 317 APIs, Xavier et al. (2017) found that for the median library, 14.78% of API changes break compatibility with its previous versions and that the frequency of these changes increased over time. However, not that many clients were impacted by these breaking changes (median of 2.54%). On the topic of breaking APIs, Bogart et al. (2016) conducted interviews with API developers in 3 software ecosystems: Eclipse, npm, and R/CRAN. They found that each ecosystem had a different set of values that influenced their policies and their decisions of whether to break the API or not. In the Eclipse ecosystem, developers were very reluctant to break APIs, strongly favoring backward compatibility; some methods were still present after being deprecated for more than 10 years. The R/CRAN ecosystem places emphasis on the ease of installation of packages by end-users, so packages are extensively tested to see if they work. As a result, API developers notify their clients of the coming API changes, and may also coordinate with them to quickly resolve issues, a finding also echoed by Decan et al. (2017). Finally, the npm ecosystem values ease of change for API developers. Following semantic versioning, breaking the API can be done by changing the major version number of the package. Since packages stay in the repository, clients are free to upgrade or not when this happens.

Regarding the clusters of APIs that we found, APIs in clusters 1, 4, and 7 seem wary of imposing too much work on their clients, and as such seem closer to the strategy employed in the Eclipse ecosystem. APIs in Cluster 5 behave somewhat similarly, at least for older API elements. On the other hand, the APIs in clusters 2, 3, and 6 are much less wary of deprecating entities, similarly, to the npm and R/CRAN ecosystems.

Bogart et al. (2016) also detail some strategies clients use to cope with change, including actively monitoring APIs for changes, doing so reactively, and limiting the number of dependencies to APIs. The latter can go to the extreme of keeping a local copy of the API to avoid migrating to a newer version in the case of R/CRAN. Another behavior is observed by Decan et al. (2016) in the case of R/CRAN: the rigid policy of forcing all packages to work together can become burdensome. Indeed, since packages have to react to API changes, the coordination and reaction costs can be excessive. As a result, an increasing number of R packages are now primarily found on GitHub, not CRAN, meaning that the developers are not affected by R/CRAN's strict policies.

Decan et al. (2017) also investigated the evolution of the package dependencies in the npm, CRAN, and RubyGems ecosystems. An interesting finding in the context of this paper is the increasing tendency in the npm and (to a lesser extent) RubyGems packages to specify maximal version constraints. This means that some package maintainer specifies a maximal version number of the packages they depend on, to shield themselves from future updates

that might force them to react to breaking changes. This strategy is complementary to the strategies documented by Bogart et al. mentioned above. They note that this behavior was not observed in R/CRAN, where a single version of each package—the latest—is stored at any given time, so specifying a specific version is of limited usefulness; package maintainers have to update anyways. In this study, we found that a large number of API clients did not update their API version (the exception being Spring), which seems to go along the lines of the behavior observed by Decan.

12.2 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 (2006), 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. (2010) 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 and Pletcher (2010, 2011) 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. (2011) mine usages of popular Java APIs by crawling SourceForge to create a corpus of usage examples that form 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.

12.3 Supporting API Evolution

Beyond empirical studies on API evolution, researchers have proposed several approaches to support API evolution and reduce the efforts of client developers. Chow and Notkin (1996) present an approach where the API developers annotate changed methods with replacement rules that will be used to update client systems. Henkel and Diwan (2005) propose CatchUp!, a tool using an IDE to capture and replay refactorings related to the API evolution. Dig et al. (2007) propose a refactoring-aware version control system for the same purposes.

Dagenais and Robillard observe the framework's evolution to make API change recommendations (Dagenais and Robillard 2008), while Schäfer et al. observe the client's evolution (Schäfer et al. 2008). Wu et al. present a hybrid approach (Wu et al. 2010) that includes textual similarity.

Nguyen et al. (2010) propose a tool (LibSync) that uses graph-based techniques to help developers migrate from one framework version to another.

Finally, Holmes and Walker notify developers of external changes to focus their attention on these events (Holmes and Walker 2010).

13 Conclusion

We have presented an empirical study on the effect of deprecation of Java API artifacts on their clients. This work expands upon 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), the scale of the dataset (25,357 vs 2600 clients) and the nature of the dataset (third- party APIs vs third-party and language APIs).

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. In the case of the JDK API, we saw that only 4 clients were affected, and all of them were affected by deprecation because they introduced a call to the deprecated entity at the time it was already deprecated, thereby limiting the probability of a reaction from these clients.

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 respects 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. In this study, we also studied how clients of a language API (JDK API) are affected by deprecation, and we see that in contrast to Smalltalk APIs, clients are rarely affected by deprecation. We also went further and looked at the impact of deprecation policies on the reactions of clients, and found that an API's policy on deprecation may have a major role to play in a client's decision to react. 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.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

Asm bytecode manipulator. http://asm.ow2.org/. Accessed on 7 April 2016

Bavota G, Linares-Vasquez M, Bernal-Cardenas CE, Penta MD, Oliveto R, Poshyvanyk D (2015) The impact of api change-and fault-proneness on the user ratings of android apps. IEEE Trans Softw Eng (TSE) 41(4):384–407

Begel A, Zimmermann T (2014) Analyze this! 145 questions for data scientists in software engineering. In: Proceedings of the 36th ACM/IEEE international conference on software engineering, ICSE '14. ACM, pp 12–23

Black A, Ducasse S, Nierstrasz O, Pollet D, Cassou D, Denker M (2009) Pharo by example. Square Bracket Associates

Bogart C, Kästner C, Herbsleb JD, Thung F (2016) How to break an API: cost negotiation and community values in three software ecosystems. In: Proceedings of the 24th ACM SIGSOFT international symposium on foundations of software engineering (FSE), pp 109–120

Brito G, Hora A, Valente MT, Robbes R (2016) 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, p to appear

Brooks FP (1975) No silver bullet. Software state-of-the-art, pp 14–29

Businge J, Serebrenik A, van den Brand MG (2013a) Eclipse api usage: the good and the bad. Softw Qual J 23(1):107–141

Businge J, Serebrenik A, van den Brand M (2013b) Analyzing the eclipse API usage: putting the developer in the loop. In: Proceedings of the 17th european conference on software maintenance and reengineering, CSMR, pp 37–46

Chacon S (2009) Pro git. Apress

Chow K, Notkin D (1996) Semi-automatic update of applications in response to library changes. In: Proceedings of international conference on software maintenance (ICSM), pp 359–368

Cossette BE, Walker RJ (2012) 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, p 55

Dagenais B, Hendren L (2008) Enabling static analysis for partial java programs. ACM Sigplan Notices 43(10):313–328

Dagenais B, Robillard MP (2008) Recommending adaptive changes for framework evolution. In: Proceedings of 30th international conference on software engineering (ICSE), pp 481–490

Decan A, Mens T, Claes M, Grosjean P (2016) When github meets CRAN: an analysis of inter-repository package dependency problems. In: Proceedings of the 23rd IEEE international conference on software analysis, evolution, and reengineering (SANER), pp 493–504

Decan A, Mens T, Claes M (2017) An empirical comparison of dependency issues in OSS packaging ecosystems. In: Proceedings of the 24th IEEE international conference on software analysis, evolution and reengineering (SANER), pp 2–12

Dig D, Johnson RE (2005) The role of refactorings in api evolution. In: ICSM 2005: proceedings Of the 21st international conference on software maintenance, pp 389–398

Dig D, Johnson R (2006) How do apis evolve? A story of refactoring. J Softw Maint Evol Res Pract 18(2):83–107

Dig D, Manzoor K, Johnson R, Nguyen TN (2007) Refactoring-aware configuration management for object-oriented programs. In: 29th international conference on software engineering, pp 427–436

Easymock api repository. https://github.com/easymock/easymock. Accessed on 7 April 2016

Espinha T, Zaidman A, Gross H-G (2015) Web api fragility: how robust is your mobile application? In: Proceedings of the 2nd international conference on mobile software engineering and systems (MOBILESoft). IEEE, pp 12–21

Fraser SD, Brooks Jr FP, Fowler M, Lopez R, Namioka A, Northrop L, Parnas DL, Thomas D (2007) 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, pp 1026–1030

Gousios G (2013) The ghtorrent dataset and tool suite. In: Proceedings of the 10th working conference on mining software repositories, MSR 2013, pp 233–236

Gousios G, Vasilescu B, Serebrenik A, Zaidman A (2014) Lean GHTorrent: Github data on demand. In: Proceedings of the 11th working conference on mining software repositories, pp 384–387

Guava api repository. https://github.com/google/guava. Accessed on 7 April 2016

Guice api repository. https://github.com/google/guice. Accessed on 7 April 2016

Henkel J, Diwan A (2005) Catchup!: capturing and replaying refactorings to support API evolution. In: Proceedings of 27th international conference on software engineering (ICSE), pp 274–283

Hibernate api repository. https://github.com/hibernate/hibernate-orm. Accessed on 7 April 2016

http://www.github.com. Last Accessed February 2017

http://www.oracle.com/technetwork/java/javase/compatibility-417013.html#incompatibilities. Last Accessed February 2017

Holmes R, Walker RJ (2010) Customized awareness: recommending relevant external change events. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, vol 1. ACM, pp 465–474

Hora A, Robbes R, Anquetil N, Etien A, Ducasse S, Valente MT (2015) How do developers react to api evolution? The pharo ecosystem case. In: IEEE international conference on software maintenance and evolution (ICSME), 2015. IEEE, pp 251–260

Hora AC, Valente MT, Robbes R, Anquetil N (2016) When should internal interfaces be promoted to public? In: Proceedings of the 24th ACM SIGSOFT international symposium on foundations of software engineering (FSE), pp 278–289

Hou D, Pletcher DM (2010) 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, RSSE '10. ACM, pp 26–30

Hou D, Pletcher DM (2011) An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In: IEEE 27th international conference on software maintenance, ICSM 2011, Williamsburg, VA, USA, September 25–30, 2011, pp 233–242

Hou D, Yao X (2011) Exploring the intent behind API evolution: a case study. In: Proceedings of the 18th working conference on reverse engineering (WCRE), pp 131–140

Ingalls D, Kaehler T, Maloney J, Wallace S, Kay A (1997) Back to the future: the story of squeak, a practical smalltalk written in itself. In: Proceedings of the 12th ACM SIGPLAN conference on object-oriented programming (OOPSLA) 1997, pp 318–326

Johnson RE, Foote B (1988) Designing reusable classes. Journal of Object-Oriented Programming 1(2):22–35

Juzgado NJ, Vegas S (2011) The role of non-exact replications in software engineering experiments. Empir Softw Eng 16(3):295–324

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. In: Proceedings of the 11th working conference on mining software repositories. ACM, pp 92–101

Lämmel R, Pek E, Starek J (2011) Large-scale, ast-based api-usage analysis of open-source java projects. In: Proceedings of ACM symposium on applied computing (SAC), p 1317

Lienhard A, Renggli L (2005) SqueakSource-Smart Monticello Repository. European Smalltalk User Group Innovation Technology Award, August

Linares-Vásquez M, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D (2014) 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, pp 83–94

Lloyd S (1982) Least squares quantization in pcm. IEEE Trans Inf Theory 28(2):129–137

Lungu M, Robbes R, Lanza M (2010) Recovering inter-project dependencies in software ecosystems. In: ASE'10: proceedings of the 25th IEEE/ACM international conference on automated software engineering, ASE '10, pp 309–312

McDonnell T, Ray B, Kim M (2013) An empirical study of API stability and adoption in the android ecosystem. In: Proceedings of 29th IEEE international conference on software maintenance (ICSM). IEEE, pp 70–79

Mileva YM, Dallmeier V, Zeller A (2010) Mining api popularity. In: Testing–practice and research techniques. Springer, pp 173–180

Mooi E, Sarstedt M (2010) Cluster analysis. Springer

Nagappan M, Zimmermann T, Bird C (2013) Diversity in software engineering research. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering. ACM, pp 466–476

Nguyen HA, Nguyen TT, Wilson G. Jr., Nguyen AT, Kim M, Nguyen TN (2010) A graph-based approach to api usage adaptation. In: Proceedings of ACM international conference on object oriented programming systems languages and applications, pp 302–321

PYPL popularity of programming language. http://pypl.github.io. Accessed on 19 Feb 2017

Raemaekers S, van Deursen A, Visser J (2012) Measuring software library stability through historical version analysis. In: 28th IEEE international conference on software maintenance (ICSM). IEEE, pp 378–387

Raemaekers S, van Deursen A, Visser J (2014) Semantic versioning versus breaking changes: a study of the maven repository. In: IEEE 14th international working conference on source code analysis and manipulation (SCAM), 2014. IEEE, pp 215–224

Robbes R, Lungu M, Röthlisberger D (2012) 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, p 56

Sawant AA, Bacchelli A (2015) A dataset for api usage. In: Proceedings of 12th IEEE working conference on mining software repositories (MSR). IEEE, pp 506–509

Sawant AA, Bacchelli A (2016) Fine-grape: fine-grained api usage extractor – an approach and dataset to investigate api usage. Empir Softw Eng 22(3):1348–1371

Schäfer T, Jonas J, Mezini M (2008) Mining framework usage changes from instantiation code. In: Proceedings of 30th international conference on software engineering (ICSE), pp 471–480

Spring api repository. https://github.com/spring-projects/spring-framework. Accessed on 7 April 2016

Tiobe index. http://www.tiobe.com/tiobe_index. Accessed on 19 Feb 2017

Wang S, Keivanloo I, Zou Y (2014) How do developers react to restful api evolution? Service-Oriented Computing pp 245–259

Wu W, Guéhéneuc Y-G, Antoniol G, Kim M (2010) Aura: a hybrid approach to identify framework evolution. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, vol 1. ACM, pp 325–334

Xavier L, Brito A, Hora AC, Valente MT (2017) Historical and impact analysis of API breaking changes: a large-scale study. In: Proceedings of the 24th IEEE international conference on software analysis, evolution and reengineering, (SANER), pp 138–147

Xie T, Pei J (2006) Mapo: mining api usages from open source repositories. In: Proceedings of the 2006 international workshop on mining software repositories (MSR). ACM, pp 54–57

Anand Ashok Sawant is a PhD student working in the Software Engineering Research Group at the Delft University of Technology, The Netherlands. His current research interest is in API usage and API adoption. His work involves mining of API usage from open source repositories, conducting empirical studies on usage data and qualitative studies, all of which aim to further the understanding of API usage and improve the usage of APIs.

Romain Robbes is an Associate Professor at the Free University of Bozen-Bolzano, in the SwSE research group, where he works since April 2017. Before that, he was an Assistant, then Associate Professor at the University of Chile (Computer Science Department), in the PLEIAD research lab. He earned his PhD in 2008 from the University of Lugano, Switzerland and received his Master's degree from the University of Caen, France. His research interests lie in Empirical Software Engineering, including, but not limited to, Mining Software Repositories. He authored more than 80 papers on these topics, including top software engineering and programming languages venues such as ICSE, FSE, ASE, EMSE, ECOOP, or OOPSLA, received best paper awards at WCRE 2009 and MSR 2011, and was the recipient of a Microsoft SEIF award 2011. He has served in the organizing and program committees of many software engineering conferences (ICSE, MSR, WCRE, ICSME, OOPSLA, ECOOP, IWPSE, and others) and serves on the Editorial Boards of EMSE and JSS.

Alberto Bacchelli is SNFS Professor of Empirical Software Engineering at the University of Zurich, Switzerland, where he leads the Zurich Empirical Software Engineering Team (ZEST). His primary research interest is in empirical software engineering. His focus is on developing and applying data science processes and techniques to conduct empirical experiments on complex knowledge-based activities, particularly the development of software, and improve their state of the practice. His goal is understanding and improving complex knowledge- based activities, such as software engineering, with processes, practices, and tools informed by data and actors' needs, and by engaging with other disciplines such as social sciences. He tackles research questions with both qualitative and quantitative research approaches. Prior to joining UZH, he have worked four years as assistant professor at Delft University of Technology (TU Delft), The Netherlands, in the Software Engineering Research Group (SERG).