2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

Do Developers Deprecate APIs with Replacement Messages? A Large-Scale Analysis on Java Systems

Gleison Brito*, Andre Hora*, Marco Tulio Valente*, Romain Robbes†
*ASERG Group, Department of Computer Science (DCC), Federal University of Minas Gerais, Brazil
{gleison.brito,hora,mtov}@dcc.ufmg.br
†PLEIAD Lab, Department of Computer Science (DCC), University of Chile, Santiago, Chile
rrobbes@dcc.uchile.cl

***Abstract*—As any other software system, frameworks and libraries evolve over time, and so their APIs. Consequently, client systems should be updated to benefit from improved APIs. To facilitate this task and preserve backward compatibility, API elements should always be deprecated with clear replacement messages. However, in practice, there are evidences that API elements are usually deprecated without such messages. In this paper, we study a set of questions regarding the adoption of deprecation messages. Our goal is twofold: to measure the usage of deprecation messages and to investigate whether a tool is needed to recommend such messages. Thus, we verify (i) the frequency of deprecated elements with replacement messages, (ii) the impact of software evolution on such frequency, and (iii) the characteristics of systems which deprecate API elements in a correct way. Our large-scale analysis on 661 real-world Java systems shows that (i) 64% of the API elements are deprecated with replacement messages per system, (ii) there is almost no major effort to improve deprecation messages over time, and (iii) systems that deprecated API elements in a correct way are statistically significantly different from the ones that do not in terms of size and developing community. As a result, we provide the basis for the design of a tool to support client developers on detecting missing deprecation messages.**

I. Introduction

Nowadays, it is common practice to implement systems on top of frameworks and libraries [1], taking advantage of their Application Programming Interfaces (APIs) to reuse functionalities [2] and increase productivity [3]. However, as any other software system, frameworks/libraries and their APIs are subjected to evolve over time. Naturally, public types, methods and fields provided by such APIs may be renamed, removed or updated. Consequently, client systems should migrate to benefit from improved API elements.

To facilitate client developers making the transition and preserve backward compatibility, API elements should always be deprecated with replacement messages. Mechanisms to support API deprecation are provided by most of the programming languages. For example, Java has two solutions to deprecate types, methods, and fields: using deprecation annotations and/or deprecation Javadoc tags. Both annotations and Javadoc tags warn developers referencing deprecated APIs, however, the latter may be accompanied by replacement messages to suggest what to use instead. Ideally, APIs should use these mechanisms to assist client developers. In practice, previous studies indicate that API elements are often deprecated with missing or unclear replacement messages [4]–[6]. However, we are still unaware about the size of this phenomenon and whether it tends to get better (or worse) over time.

In this paper, we study a set of questions regarding the adoption of API deprecation messages. We analyze (i) the frequency of deprecated API elements with replacement messages, (ii) the impact of software evolution on such frequency, and (iii) the characteristics of systems which deprecate API elements in a correct way in terms of popularity, size, community, activity and maturity. Our goal is twofold: to measure the usage of deprecation messages and to investigate whether a tool is needed to recommend such messages. Thus, we investigate the following research questions to support our study:

- *RQ1. What is the frequency of deprecated APIs with replacement messages?*
- *RQ2. What is the impact of software evolution on the frequency of replacement messages?*
- *RQ3. What are the characteristics of software systems with high and low frequency of replacement messages?*

In this study, we provide a large-scale analysis on 661 real-world Java systems. Our results show that (i) 64% of the API elements are deprecated with replacement messages per system, (ii) there is almost no major effort to improve deprecation messages over time, and (iii) systems that deprecated API elements in a correct way are statistically significantly different from the ones that do not in terms of size and developing community. As a result, we provide the basis for the design of a tool to support client developers on detecting missing deprecation messages. Thus, the contributions of this paper are summarized as follows:

- We provide a large-scale study to better understand to what extend APIs are being deprecated with replacement messages.
- We provide the motivation to the need of a recommendation tool to assist client developers in the detection of missing replacement messages.

Structure of the paper: In Section II, we present the background in more details. We describe our experiment design in Section III. We present the experiment results in Section IV. Summary and implications are shown in Section V and threats to validity in Section VI. Finally, we present related work in Section VII, and we conclude the paper in Section VIII.

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.99

IEEE computer society

II. Background

We define an API element as a public/protected type, field or method. In theory, before being replaced, API elements should be flagged as deprecated to support client developers making the transition to new ones. Deprecated API elements continue in the system to preserve backward compatibility, but they should not be used by developers because they may be removed in the future.

Java has two solutions to deprecate types, methods, and fields: using the annotation *@Deprecated* (supported since J2SE 5.0), and/or using the Javadoc tag *@deprecated* (supported since J2SE 1.1), as presented in Figure 1.

```
/**
 * Does some thing in old style.
 *
 * @deprecated use {@link #new()} instead.
 */
@Deprecated
public void old() {
// ...
}
```

Fig. 1. Deprecation example using annotation, tag, and replacement message.

The annotation *@Deprecated* causes the compiler to issue a warning when it finds references to deprecated API elements. The Javadoc tag *@deprecated* also warns developers about deprecated elements. However, its associated Javadoc comment may be accompanied by a message to suggest developers what to use instead, *i.e.,* a replacement message. Java deprecation guidelines present two solutions to create these replacement messages:

- *Javadoc 1.1*: Using the annotation *@see* to indicate the replacement API.
- *Javadoc 1.2 and later*: Using the word *use* followed by the annotation *@link* to indicate the replacement API (as shown in Figure 1).

Notice, however, that Java deprecation guidelines are not mandatory to follow: developers may adopt other conventions to create replacement messages, or simply do not use them.

Ideally, an API element should be deprecated with deprecation annotation, deprecation Javadoc tags, and replacement messages in order to support developers migrating to new/better ones. However, in practice, previous studies indicate that API elements are often deprecated with missing or unclear replacement messages [4]–[6]. In this work, we verify at a large-scale level (i) the frequency of deprecated elements with replacement messages in Java systems, (ii) whether this frequency is increasing or decreasing over time, (iii) and the characteristics of systems which are deprecating API elements in a correct way.

III. Experiment Design

A. Selecting Case Studies

We analyze Java systems hosted on the popular social coding platform GitHub. We use three criteria to select the systems: number of stars, releases, and deprecated API elements.

1) **Number of stars.** GitHub provides the feature number of stars that lets users show their interest on systems. We select systems with 100 or more stars in GitHub in order to filter real-world and popular ones.
2) **Number of releases.** We select systems with three or more public releases available on GitHub. We use this criteria to assess API deprecation evolution.
3) **Number of deprecated APIs.** We select systems with one or more public/protected deprecated API elements. We use this criteria to filter out systems with no deprecated element. These systems are not in the scope of our study.

Based on the above filtering criteria, we selected **661** systems. To better characterize such **systems**, Figure 2 presents the distribution of the three criteria. For number of stars, the first quartile, median, and third quartile is 154, 279, and 593, respectively. The top-3 systems with more stars are elastic/elasticsearch (12.4K stars), nostra13/Android-Universal-Image-Loader (9.7K), and google/iosched (7.7K). For number of releases, the first quartile, median, and third quartile is 11, 24, and 57. The top-3 systems with more releases are JetBrains/kotlin (2.9K releases), rstudio/rstudio (2.1K), and freenet/fred (1.9K). Finally, for number of deprecated API elements, the first quartile, median, and third quartile is 3, 12, and 39. The top-3 systems with more deprecated APIs are groovy/groovy-eclipse (2.3K deprecated APIs), openmrs/openmrs-core (1.2K), and OpenGamma/OG-Platform (994).

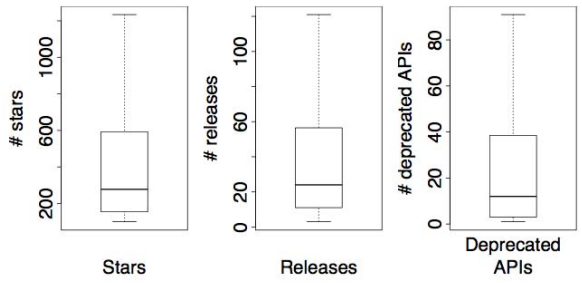

Fig. 2. Distribution of number of stars, releases, and deprecated API elements in the selected systems.

B. Extracting Deprecated API Elements

As a first step to support answering our research questions, we extract all deprecated API elements (types, fields, and methods) and their associated Javadoc from the systems under analysis. As presented in Section II, a Java API element can be deprecated using the annotation *@Deprecated* or the Javadoc tag *@deprecated*. To find deprecated API elements, we implemented a parser based on the Eclipse JDT library to look for deprecation annotations and tags. We restricted our analysis to public and protected API elements because they represent

the external contract to clients. We extracted 5,802 deprecated types, 4,427 deprecated fields, and 26,890 deprecated methods, corresponding to a total of 37,119 deprecated API elements.

C. Extracting Replacement Messages in Deprecated API Elements

When a method is deprecated with the Javadoc tag, it may be accompanied by a replacement message to help client developers. As presented in Section II, Java guidelines present two solutions to create deprecation replacement messages: (i) using the annotation *@see*, or (ii) using the word *use* and the annotation *@link*. In practice, however, developers may adopt other guidelines to create replacement messages or simply do not use any.

To detect alternative guidelines, we extracted deprecation messages in deprecated API elements with the support of the JDT library, and we manually inspected a subset of these messages. As a result of such manual analysis, we detected, in addition to the word *use*, three frequent words/patterns to indicate replacement: *refer*, *equivalent*, *replace** (*i.e., replace*, *replaced*, *replacement*), *see*, *moved*, *instead*, and *should be used*. We also confirmed the two frequently adopted annotations: *@link* and *@see*.

Table I shows the frequency for each replacement guideline as well as message examples. The most adopted guideline is the word *use*: with 17,810 cases, it occurs in 47.9% of the deprecated API elements. In contrast, the least adopted guideline is *should be used*: with 33 cases, it occurs only in 0.09%. Notice that some guidelines may occur together in the same message. For example, *use* commonly happens with *@link*. In total, 22,075 (59.5%) API elements were deprecated with replacement messages from all 37,119. Such data is further explored in Research Question 1, and its evolution is analyzed in Research Question 2.

D. Defining Metrics Likely to Impact API Deprecation

To support answering Research Question 3, about the characteristics of systems that deprecate API elements with replacement messages, we define metrics in five dimensions which are likely to affect development practices: system popularity, size, community, activity, and maturity. The idea is to investigate whether such metrics have an impact on the way developers deprecate API elements. These dimensions and metrics are described below, and summarized in Table II.

- **Popularity.** It includes metrics that represent how popular is the system in GitHub in *number of stars*, *number of watchers*, and *number of forks*. The rationale is that popular systems may have more clients, thus they may have more concerns about their APIs.
- **Size.** It includes metrics related to system size in terms of *number of files* and *number of API elements* (*i.e.,* sum of number of types, fields and methods). The rationale is that larger systems may be harder to maintain, so it should be more difficult to keep track of all API changes. In contrast, smaller systems may be easier to control and to keep track of.
- **Community.** It includes metrics that represent the system community size in *number of contributors*, *average files per contributor*, and *average API elements per contributor*. The rationale is that systems with larger community (*i.e.,* more contributors, but less files and API elements per contributor) may be somehow easier to maintain, thus it should be easier to keep track of API changes.
- **Activity.** It has metrics related to the system activity level in terms of *number of commits*, *number of releases*, *average days per release*. The rationale is that systems with more activity tend to respond faster to client complains. Therefore they may be more likely to improve their APIs.
- **Maturity.** It is about the system age, in number of days. The rationale is that older systems are reliable, thus they may have more stable APIs. In contrast, it is natural to expect that newer systems may have more unstable APIs.

TABLE II
METRICS LIKELY TO IMPACT API DEPRECATION.

Dimension	Metric
Popularity	number of stars
	number of watchers
	number of forks
Size	number of files
	number of API elements
Community	number of contributors
	average files per contributor
	average API elements per contributor
Activity	number of commits
	number of releases
	average days per release
Maturity	age (in number of days)

E. Extracting Metrics from Case Studies

We extracted the proposed metrics from two group of systems: the ones deprecating API elements in a correct way and the ones not doing that. Then, we assessed such groups to verify whether they are statistically different with respect to the proposed metrics. These two steps are detailed below.

Selecting systems and extracting metrics. We sorted all systems, in descending order, based on the percentage of deprecated API elements with replacement messages. We selected two groups, top-25% (*i.e.,* systems with the highest percentage of deprecated API elements with replacement messages) and bottom-25% (*i.e.,* systems with the lowest percentage). Each group has 166 systems. Figure 3 shows the distribution of the percentage of deprecated API elements with replacement messages in each group. The median percentage is 100% for *top* systems and 12.5% for *bottom* systems. Finally, we extracted the metrics described in the previous subsection for the *top* and *bottom* systems.

Assessing selected systems. We compare the values of each metric in *top* and *bottom* systems. We first analyze the statistical significance of the difference between the two groups by applying the Mann-Whitney U test at *p-value* = 0.05. To show the effect size of the difference between the two groups, we compute Cliff's Delta (or *d*). Following the guidelines

TABLE I
FREQUENCY OF REPLACEMENT MESSAGE GUIDELINES IN DEPRECATED API ELEMENTS.

Guideline	Frequency	Replacement Message Example
use	17,810 (47.9%)	`use encodeURL(String url) instead` (Apache Tomcat)
replace*	2,171 (5.8%)	`Replace to getParameter(String, int)` (Dubbo)
refer	1,070 (2.9%)	`property will be removed, refer @link #getEncoded(boolean)` (Actor Platform)
equivalent	166 (0.3%)	`The @link Iterable equivalent is @link ImmutableSet#of()` (Google Guava)
see	777 (2.1%)	`See servlet 3.0 apis like HttpServletRequest.getParts()` (Eclipse Jetty)
moved	224 (0.6%)	`deprecated since 2008-05-28. Moved to stapler` (Eclipse Hudson)
instead	14173 (38.2%)	`Use KEY_LMETA instead` (Facebook Nifty)
should be used	33 (0.09%)	`org.bukkit.entity.minecart.PoweredMinecart should be used instead` (Bukkit)
@link	14,852 (40%)	`Use @link #setController(DraweeController) instead` (Facebook Fresco)
@see	2,334 (6.3%)	`@see #getStartRequests` (WebMagic)

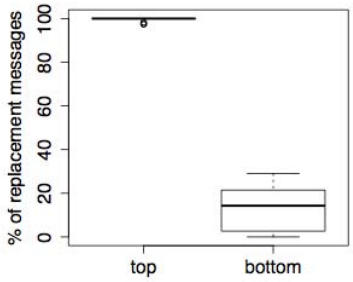

Fig. 3. Distribution of the percentage of deprecated APIs with replacement messages in top-25% and bottom-25% systems.

in [7]–[9], we interpret the effect size values as small for $0.147 < d < 0.33$, medium for $0.33 < d < 0.474$, and large for $d < 0.474$.

IV. RESULTS

A. RQ1. What is the frequency of deprecated APIs with replacement messages?

We analyze the frequency of deprecated API elements with replacement messages for types, fields, and methods in the last release of the cases studies. As presented in Table III, 3,825 (65.9%) deprecated types contains replacement messages. For deprecated fields and methods, these numbers are 2,621 (59.2%) and 15,629 (58.1%), respectively. Considering all deprecated API elements, 22,075 (59.5%) contains replacement messages.

TABLE III
NUMBER OF DEPRECATED API ELEMENTS WITH REPLACEMENT MESSAGES.

Types	Fields	Methods	All
3,825 (65.9%)	2,621 (59.2%)	15,629 (58.1%)	22,075 (59.5%)

Next we present the absolute and relative analysis per system. For types, fields and methods analysis, we consider only systems that have at least one deprecated type, field, and method, respectively.

Absolute analysis. Figure 4 shows the distribution of the absolute number of deprecated API elements with replacement messages per system. For types, the first quartile is 1, the median is 2, and the third quartile is 6. Fields present first quartile 0, median 1, and third quartile 5. For methods, the first quartile is 1, the median is 4, and the third quartile is 15. In absolute terms, we note that methods are the most deprecated elements with replacement messages (median 4 per system) while fields are the least one (median 1 per system). Considering all API elements, the 1st quartile is 1, the median is 5, and the 3rd quartile is 18.

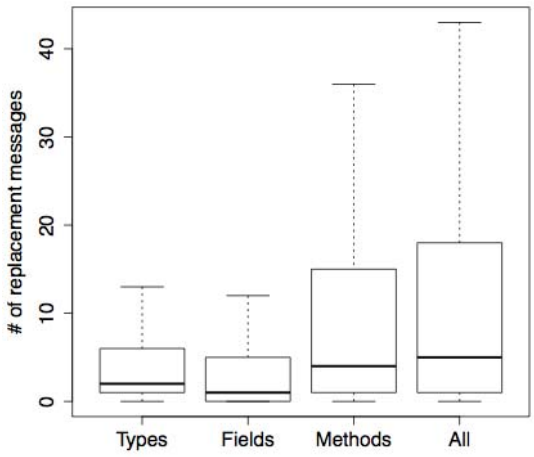

Fig. 4. Absolute distribution of deprecated API elements with replacement messages.

Relative analysis. Figure 5 presents the distribution of the relative number of deprecated API elements with replacement messages per system. For types, the first quartile is 25%, the median is 71.7%, and the third quartile is 100%. There are 118 systems with 100% of types deprecated with replacement messages, such as spring-projects/spring-android, jenkinsci/github-plugin and google/guava. In contrast, there are 63 systems with types deprecated without replacement messages, *e.g.,*caelum/vraptor, cymcsg/UltimateAndroid and spring-projects/spring-roo.

For fields, the first quartile is 0%, the median is 50%,

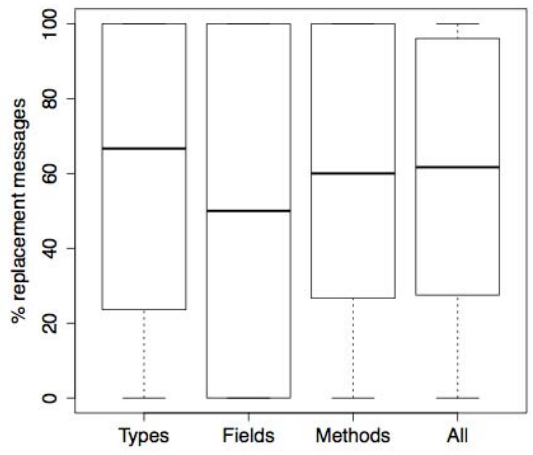

Fig. 5. Relative distribution of deprecated API elements with replacement messages.

and the third quartile is 100%. There are 74 systems with 100% of fields deprecated with replacement messages, such as spring-projects/spring-framework, hibernate/hibernate-orm and apache/tomcat70. We also detect 77 systems with fields deprecated with no replacement messages, such as goldmansachs/gs-collections, phonegap/phonegap-app-developer and spring-projects/spring-webflow.

For methods, the first quartile is 28.6%, the median is 61.5%, and the third quartile is 100%. There are 162 systems with 100% of their deprecated methods with replacement messages, such as square/picasso, nostra13/Android-Universal-Image-Loader and eclipse/jgit. We also find 58 systems in which all deprecated methods do not have replacement messages, such as JPMoresmau/eclipsefp, spring-projects/spring-data-neo4j, and hibernate/hibernate-validator.

We note that, according to the median, types are the most deprecated elements with replacement messages (median 71.7% per system) while fields are the least one (median 50% per system). The third quartile at 100% for types, fields and methods shows that 25% of the systems always deprecate all elements with replacement messages. In contrast, the first quartile at 0% for fields shows that 25% of the systems never deprecate fields with replacement messages.

When considering all API elements, the first quartile is 28.6%, the median is 64%, and the third quartile is 97.5%. That is to say, considering the median, around 2/3 of the API elements are deprecated with replacement messages while 1/3 lacks such messages. There are 164 systems (24.8%) with 100% of deprecated API elements with replacement messages, such as code4craft/webmagic, google/guice and bumptech/glide.

Summary: According to the median, 64% of the API elements are deprecated with replacement messages per system. This percentage is 71.7% for types, 50% for fields, and 61.5% for methods, suggesting that developers are usually more concerned with types and less with fields. We see that 25% of the systems deprecate types, fields and methods with replacement messages. However, other 25% never provide replacement messages for deprecated fields.

B. RQ2. What is the impact of software evolution on the frequency of replacement messages?

In order to verify the impact of software evolution on deprecation, we analyze the frequency of deprecated API elements with replacement messages in two distinct releases of the systems. We compare the first publicly available release with the last one (*i.e.,* the same of Research Question 1). Considering the first and last releases, we find 10,798 and 22,075 deprecated API elements with replacement messages, respectively.

Absolute analysis. Figure 6 shows the distribution of the absolute number of deprecated API elements with replacement messages per system, in the analyzed releases. In the first release, the first quartile is 1, the median is 3, and the third quartile is 17. In the last release, the first quartile is 1, the median is 5, and the third quartile is 18. Notice that the median of the absolute number of deprecated API elements with replacement messages increases over time (from 3 to 5). In fact, this is expected due to the natural evolution of the systems which are likely to provide more features.

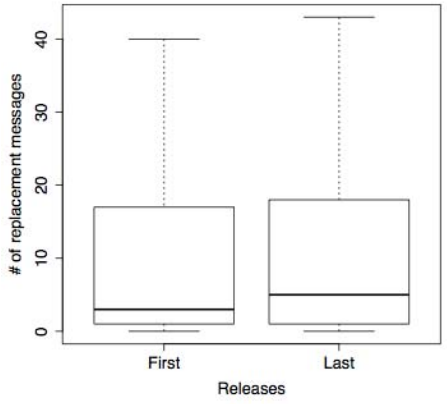

Fig. 6. Absolute distribution of deprecated API elements with replacement messages in the two analyzed releases.

Relative analysis. Figure 7 presents the distribution of the relative number of deprecated API elements with replacement messages per system, in the analyzed releases. In the first release, the first quartile is 10.5%, the median is 59.3%, and the third quartile is 95.4%. In the last release, the first quartile is 28.6%, the median is 64%, and the third quartile is 97.5%. Notice that the median of relative number of deprecated API elements with replacement messages increases only 4.7% (from 59.3% to 64%). That is to say, according to the median, there is almost no effort from developers to provide deprecation messages. Similarly, the third quartile remains stable, increasing only 2.1% (from 95.4% to 97.5%), meaning that systems whose developers are concerned with replacement

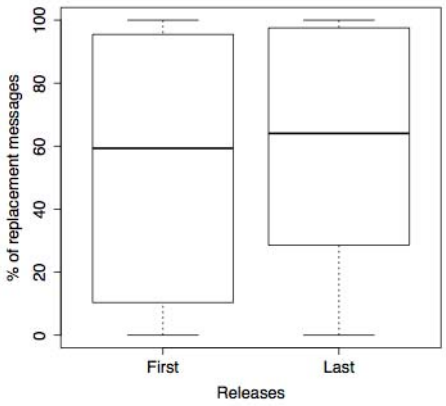

Fig. 7. Relative distribution of deprecated APIs with replacement messages in the two analyzed releases.

messages tend to be like that since beginning. In contrast, by analyzing the evolution of first quartile, we observe significant changes. The first quartile increased by 18.1% (from 10.5% to 28.6%), meaning that 25% of the systems are increasingly by adopting replacement messages.

Overall, from the first to the last release, 152 systems (23%) increased the relative number of deprecated API elements with replacement messages. Figure 8 shows the distribution of the relative number of deprecated messages with replacement messages for such systems. The first quartile is 0% for the first release, and 27.4% for last release. The median is 20% against 57.7%, and the third quartile is 53% against 80.2%. Examples of systems in this category include spring-projects/spring-boot (increased from 0% to 55%) and Netflix/eureka (increased from 0% to 64%).

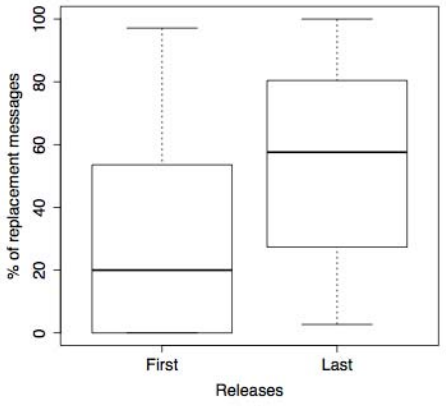

Fig. 8. Relative distribution of systems increasing percentage of replacement messages.

In contrast, 123 systems (18.6%) decreased the number of deprecate API elements with replacement messages. Figure 9 shows the distribution for such systems. The first quartile is 57.9% for the first release, and 29.2% for the last release. The median is 85.1% against 50%, and the 3rd quartile is 100% against 71.3%. Examples of systems in this category include aptana/Pydev (decreased from 91.6% to 50%) and spring-projects/spring-framework (decreased from 98% to 68%).

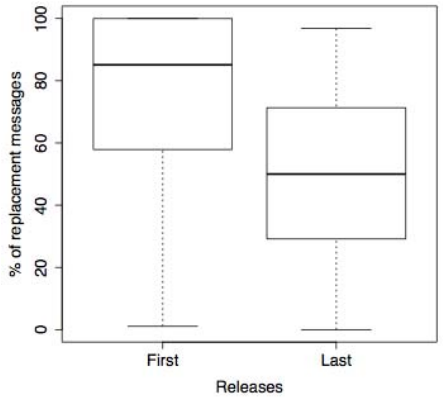

Fig. 9. Relative distribution of systems decreasing percentage of replacement messages.

Finally, we observe that in 386 systems (58.4%) the number of deprecated methods with replacement messages remains stable. Examples include spring-projects/spring-batch and mcxiaoke/android-volley, both with 100%

Summary: On the median, the relative number of deprecated API elements with replacement messages remain almost constant (from 59.3% to 64%), showing that there is no major effort to improve the ratio of deprecation messages. Overall, for 23% of the systems, the number of deprecation with replacement messages increases (from 20% to 57.7%) while for 18.6%, this number decreases (from 85.1% to 50%).

C. RQ3. What are the characteristics of software systems with high and low frequency of replacement messages?

In this research question we investigate whether system popularity, size, community, activity, and maturity have an impact on the way developers deprecate API elements. We perform that by comparing *top* and *bottom* systems; *top* systems have 100% of their API elements deprecated with replacement messages, while *bottom* barely do that. Table IV presents the metrics and their respective *p-values* and *d* applied on *top* and *bottom* systems (see subsection III-E). Metrics in bold have *p-value* < 0.05, and $d > 0.147$, *i.e.,* they are statistically significant different with at least a small effect size in *top* and *bottom* systems. We find that the selected *top* and *bottom* systems are statistically significant different with at least a small effect size in 7 out of the 12 metrics: all size and community metrics as well as number of commits and releases in activity. Effect size is large in 2 metrics (number of API elements and number of commits), medium in 2 (number of files and average API elements per contributor), and small in 3 (number of contributors, average files per contributor, and number of releases). In the following we investigate each dimension.

TABLE IV
METRICS AND THEIR RESPECTIVE *p-values* AND *d* ON *top* AND *bottom* SYSTEMS. BOLD MEANS *p-value* < 0.05 (STATISTICALLY SIGNIFICANT DIFFERENT), AND $d > 0.147$ (AT LEAST A SMALL EFFECT SIZE). RELATIONSHIP: "+" = *top* SYSTEMS HAVE SIGNIFICANTLY HIGHER VALUE ON THIS METRIC. "-" = *bottom* SYSTEMS HAVE SIGNIFICANTLY HIGHER VALUE ON THIS METRIC.

Dimension	Metric	p-value	d-value	Relationship
Popularity	number of stars	0.846	0.150 (negligible)	+
	number of watchers	0.130	0.09 (negligible)	+
	number of forks	0.043	0.09 (negligible)	+
Size	**number of files**	**< 0.001**	**0.421 (medium)**	-
	number of API elements	**< 0.001**	**0.656 (large)**	-
Community	**number of contributors**	**< 0.001**	**0.265 (small))**	-
	average files per contributor	**< 0.001**	**0.273 (small)**	-
	average API elements per contributor	**< 0.001**	**0.317 (medium)**	-
Activity	**number of commits**	**< 0.001**	**0.466 (large)**	-
	number of releases	**0.033**	**0.169 (small)**	-
	average days per release	0.435	0.095(negligible)	-
Maturity	age (in number of days)	0.199	0.062 (negligible)	-

- **Popularity.** We detect that there is no difference in *top* and *bottom* systems with respect to the popularity metrics *number of stars*, *watchers*, and *forks*. Therefore, we can conclude that system popularity has no statistical effect.
- **Size.** We observe that *top* systems are smaller than *bottom* ones both in *number of files* and *number of API elements* (notice the "-" on the relationship column). In fact, smaller systems tend to be easier to maintain and to keep track of API elements, facilitating the control of replacement messages.
- **Community.** We see that *top* systems have less contributors than *bottom* ones. This result is somehow related to the previous one: it is expected that smaller systems have less contributors. When we check the ratio of *files per contributor* and *API elements per contributor*, we notice that relative numbers are significant. Systems with less files and API elements per contributor are more likely to have replacement messages.
- **Activity.** For activity dimension we observe that *top* systems have less commits and releases than *bottom* ones. An explanation is that *bottom* systems have more code changes, thus they may be more likely to degrade their APIs.
- **Maturity.** Similarly to popularity, we could not find relevant difference between *top* and *bottom* systems with respect to their maturity (*i.e.,* age in number of days). Older systems were expected to be more stable and to provide better APIs. However, the result shows that system age has no effect on the way developers deprecate API elements with replacement messages.

As an example, we present in Table V a comparison between a *top* and a *bottom* system. The *top* system is linkedin/parseq: it has 100% (32) of its API elements deprecated with replacement messages. The *bottom* one is apache/hive: it has 16.9% (23 from a total of 136) of its API elements deprecated with replacement messages. In fact, size, community and activity aspects of both systems are clearly distinct: linkedin/parseq is easier to manage when comparing to apache/hive.

Summary: Top systems are statistically significantly different from bottom projects in 7 out of 12 metrics. Top systems are smaller in terms of number of files and API elements but have more contributors per files and API elements. System popularity and maturity seems to have no effect on the way developers deprecate API elements with replacement messages.

TABLE V
COMPARISON BETWEEN A TOP AND BOTTOM SYSTEM.

Metric	median values	linkedin/parseq (top system)	apache/hive (bottom system)
number of files	668	339	13,001
number of API elem	4,803	2,269	83,045
number of contrib	18	12	39
avg. files per contrib	42.9	28.2	333.3
avg. API elem/contrib	260.8	189	2,129
number of releases	24	20	76
number of commits	1,232	263	6,848

V. SUMMARY AND IMPLICATIONS

From our analysis on 661 real-word Java systems, we provide insights into the adoption of replacement messages. Research Question 1 shows that 64% of the API elements are deprecated with replacement messages per system. The percentage is 71.7% for types, 50% for fields, and 61.5% for methods, suggesting that developers are more concerned with types and less with fields. Research Question 2 presents that the proportion of deprecated API elements with replacement messages does not get much better over time (from 59.3% to 64%). Thus that there is no major effort to improve deprecation messages. In this case, the number of deprecation with replacement messages increases for 23% of the systems while it decreases for 18.6%. Finally, Research Question 3 shows that top systems are statistically significantly different from bottom projects in several of the considered metrics. Top systems are smaller in terms of number of files/API elements and community (less contributors, but more contributors per files and API elements). In contrast, system popularity and maturity have no effect on the way developers deprecate API elements with replacement messages.

Maintaining API elements in large and complex systems is not a simple task, but may involve several developers with different level of knowledge, making it difficult to keep consistency during their evolution [4], [6], [10]. In fact, there is an effort in the literature to understand the impact of software evolution on APIs [4], [6], [11] and to detect how such impact can be alleviated (*e.g.,* [10], [12]–[20]) by mining client reactions. However, this is not performed in the context of API deprecation. Thus, together with the fact that API elements are usually deprecated without replacement messages per system (36% per system), and that such situation does not improve over time, we present two implications of our findings:

> *Implication 1*: A recommendation tool can be constructed to assist client developers by automatically inferring missing replacement messages. These messages can be inferred by mining client system reactions, *i.e.,* learning the solution adopted by clients when there is no replacement messages.

In contrast, it often happens that API elements are deprecated with replacement messages per system (64% per system). In such cases, developers point out how old API elements should be replaced. This information provides the basis for the following implication:

> *Implication 2*: The quality of a recommendation tool to detect missing messages can be assessed by its correctness in identifying deprecated API elements *with* replacement message. In other words, replacement messages of deprecated API elements can be used as an oracle for measuring accuracy of the tool in detecting valid messages.

VI. Threats to Validity

Construct Validity. The construct validity is related to whether the measurement in the study reflects real-world situations.

Classification of deprecation messages. One threat of our study is that messages may be incorrectly classified as having or not having replacement messages. In order to assess this threat we performed two analyses. First, we manually analyzed 500 randomly selected deprecation messages classified as having replacement messages. We detected 4 ($<$1%) false-positives, *i.e.,* we classified as they having replacement messages but they have not. Second, we manually analyzed 500 randomly selected deprecation messages classified as not having replacement messages. In this case, we detected 26 (5%) false-positives, *i.e.,* we classified as they not having replacement messages but they have. Therefore, in both cases the risk of wrong classification is low, so this threat is reduced.

Evolution analysis. We only analyzed the first and the last release for each system. These two releases do not characterize the entire evolution of the case studies. However, they do provide a general overview of their evolution, because the last release represents the newest one while the first release represents the oldest one.

Internal Validity. The internal validity is related to uncontrolled aspects that may affect the experimental results.

Findings Validation. We paid special attention to the appropriate use of statistical machinery (*i.e.,* Mann-Whitney test, Cliff's Delta effect size) when reporting our results in Research Question 3. This reduces the possibility that such results are due to chance.

Association is not Causation. In Research Question 3, we examined whether there are metrics associated with top and bottom systems. Notice, however, that association does not imply causation [21]. Thus, more advanced statistical analysis, *e.g.,* causal analysis [22], can be adopted to further extend our analysis.

Parser Implementation. A possible threat is the possibility of errors in the implementation of our AST parser, which detects deprecated API elements. However, as the implementation is based on JDT (a library developed by Eclipse), the risk of this threat is very reduced.

External Validity. The external validity is related to the possibility to generalize our results. We focused on the analysis of 661 open-source and real-world Java systems, therefore they are credible and representative case studies. Such systems are stored in GitHub, the most popular code repository nowadays, thus their source code are easily accessible. Despite these observations, our findings—as usual in empirical software engineering—cannot be directly generalized to other systems, specifically to systems implemented in other programming languages or commercial ones. Therefore, our study should be carried out on other systems, possibly written in other languages or commercial ones.

VII. Related Work

We separate related work in two categories, the first one about the impact of API evolution and second one in the context of API evolution analysis.

A. API Evolution Impact

McDonnell *et al.* [11] investigate API stability and adoption on a small-scale Android ecosystem. The authors found that Android APIs are evolving fast and client adoption is not following the evolution pace. Also in the Android context, Linares-Vásquez [23] analyze how API changes trigger questions and activity in StackOverflow. Results suggest that Android developers normally have more questions when the API behavior is modified.

In a large-scale study, Robbes *et al.* [4] investigate the impact of API deprecation in an ecosystem, written in the dynamic typed programming language Smalltalk. They detected that some API deprecation have large impact on the ecosystem and that the quality of deprecation messages should be improved. The authors show evidence that APIs are usually deprecated with missing and unclear messages, however their focus is on impact analysis, so they do not deep investigate deprecation messages themselves.

In a recent work, Hora *et al.* [6] studied the impact of API replacement and improvement (*i.e.,* not API deprecation) on a large-scale ecosystem also written in Smalltalk. The results of

this study also confirm the large impact on client systems, and hints that deprecation mechanisms should be more adopted.

B. API Evolution Analysis

Several approaches were proposed to support API evolution and reduce the efforts of client developers. Henkel and Diwan [24] propose CatchUp, a tool that uses a modified IDE to capture and replay refactorings related to API evolution. Chow and Notkin [25] present an approach that is supported by API developers: they annotate changed methods with replacement rules that will be used to update client systems. Hora *et al.* [19], [20] propose APIEvolutionMiner and apiwave, tools to support keeping track of API evolution and popularity.

Kim *et al.* [26] help to automatically infer rules from structural changes, computed from modifications at or above the level of method signatures. Kim *et al.* [14] propose LSDiff, a tool to support computing differences between two versions of one system. In this case, the authors take into account the body of the method to infer rules, improving their previous work [26]. Nguyen *et al.* [27] propose LibSync, atool that uses graph-based techniques to help developers migrate from one framework version to another. Dig and Johnson [28] support developers to better understand the requirements for migration tools. For instance, they found that 80% of the changes that break client systems are refactorings.

Dagenais and Robillard [12] present SemDiff, a tool that suggests replacements for API elements based on how it adapts to its own changes. Schafer *et al.* [13] propose to mine API usage change rules from client systems. Wu *et al.* [10] present AURA, an approach that combines call dependency and text similarity analyses to produce evolution rules. Meng *et al.* [15] propose a history-based matching approach (named HiMa) to support framework evolution. In this case, rules are extracted from the revisions in code history together with comments recorded in the evolution history of the framework. Hora *et al.* [16], [17] focus on the extraction of API evolution rules that only make sense for a system or domain under analysis.

Finally, studies also address the problem of discovering the mapping of APIs between different platforms that separately evolved. For example, Zhong *et al.* [29] focus on the mapping between Java and C# APIs while Gokhale *et al.* [30] study the mapping between JavaME and Android APIs.

In summary, related studies are intended to better understand API evolution and to propose solutions to API migration. None of them, however, study API evolution in the context of API deprecation and their replacement messages.

VIII. Conclusion

This paper presented a large-scale empirical study about the adoption of replacement messages of deprecated API elements. We focused one three questions: (i) the frequency of deprecated API elements with replacement messages, (ii) the impact of API evolution on such frequency, and (iii) the characteristics of systems correctly deprecating API elements. Our goal was to investigate the need of a recommendation tool to assist developers in the detection of missing replacement messages. The study was performed in the context of 661 popular and real-world Java systems. We reiterate the most interesting conclusions from our experiment results:

- 64% of the API elements are deprecated with replacement messages per system.
- The proportion of deprecated API elements with replacement messages does not get much better over time.
- Systems that deprecate API elements in a correct way are smaller and they have proportionally more contributors. In contrast, system popularity and maturity have no impact.
- A recommendation tool can be constructed to assist client developers by automatically inferring missing replacement messages. Replacement messages of deprecated API elements can be used as an oracle for measuring accuracy of the tool.

As future work, we plan to extend this research to analyze systems implemented in other programming languages. We also plan to categorize the systems under analysis to understand their differences regarding API deprecation. Finally, we plan to improve the evolution analysis by taking into account several releases in the selected case studies to better characterize the impact software evolution on API deprecation.

Acknowledgment

This research is supported by CNPq and FAPEMIG.

References

[1] T. Tourwé and T. Mens, "Automated support for framework-based software," in *International Conference on Software Maintenance*, 2003.

[2] D. Konstantopoulos, J. Marien, M. Pinkerton, and E. Braude, "Best principles in the design of shared software," in *International Computer Software and Applications Conference*, 2009.

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

[4] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? The case of a smalltalk ecosystem," in *International Symposium on the Foundations of Software Engineering*.

[5] D. Ko, K. Ma, S. Park, S. Kim, D. Kim, and Y. Le Traon, "API Document Quality for Resolving Deprecated APIs," in *Asia-Pacific Software Engineering Conference*, 2014.

[6] A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse, and M. T. Valente, "How do developers react to API evolution? the Pharo ecosystem case," in *International Conference on Software Maintenance and Evolution*, 2015.

[7] R. Grissom and J. Kim, "Effect sizes for research: A broad practical approach," *Lawrence Erlbaum Associates Publishers*, 2005.

[8] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? a case study on free android applications," in *International Conference on Software Maintenance and Evolution*, 2015.

[9] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: a threat to the success of android apps," in *Joint meeting on foundations of software engineering*, 2013.

[10] W. Wu, Y.-G. Gueheneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *International Conference on Software Engineering*, 2010.

[11] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *International Conference on Software Maintenance*.

[12] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *International Conference on Software engineering*, 2008.

[13] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *International Conference on Software engineering*, 2008.
[14] M. Kim and D. Notkin, "Discovering and Representing Systematic Code Changes," in *International Conference on Software Engineering*, 2009.
[15] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *International Conference on Software Engineering*, 2012.
[16] A. Hora, N. Anquetil, S. Ducasse, and S. Allier, "Domain Specific Warnings: Are They Any Better?" in *International Conference on Software Maintenance*, 2012.
[17] A. Hora, N. Anquetil, S. Ducasse, and M. T. Valente, "Mining System Specific Rules from Change Patterns," in *Working Conference on Reverse Engineering*, 2013.
[18] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *International Conference on Software Engineering*, 2013.
[19] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "APIEvolutionMiner: Keeping API evolution under control," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*.
[20] A. Hora and M. T. Valente, "apiwave: Keeping track of api popularity and migration," in *International Conference on Software Maintenance and Evolution*, 2015, http://apiwave.com.
[21] C. Couto, P. Pires, M. T. Valente, R. Bigonha, and N. Anquetil, "Predicting software defects with causality tests," *Journal of Systems and Software*, vol. 93, 2014.
[22] R. D. Retherford and M. K. Choe, *Statistical models for causal analysis*. John Wiley & Sons, 2011.
[23] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do API changes trigger stack overflow discussions? a study on the Android SDK," in *International Conference on Program Comprehension*, 2014.
[24] J. Henkel and A. Diwan, "Catchup!: Capturing and replaying refactorings to support API evolution," in *International Conference on Software Engineering*, 2005.
[25] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *International Conference on Software Maintenance*, 1996.
[26] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *International Conference on Software Engineering*, ser. ICSE '07, 2007.
[27] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.
[28] D. Dig and R. Johnson, "The role of refactorings in API evolution," in *International Conference on Software Maintenance*, 2005.
[29] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API mapping for language migration," in *International Conference on Software Engineering*.
[30] A. Gokhale, V. Ganapathy, and Y. Padmanaban, "Inferring likely mappings between APIs," in *International Conference on Software Engineering*.