The Journal of Systems and Software 137 (2018) 306–321

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

On the use of replacement messages in API deprecation: An empirical study

Gleison Brito[a], Andre Hora[b,*], Marco Tulio Valente[a], Romain Robbes[c]

[a] ASERG Group, Department of Computer Science (DCC), Federal University of Minas Gerais, Brazil
[b] Faculty of Computer Science (FACOM), Federal University of Mato Grosso do Sul, Brazil
[c] SwSE Research Group, Free University of Bozen-Bolzano, Italy

ARTICLE INFO

Article history:
Received 25 March 2017
Revised 12 November 2017
Accepted 8 December 2017
Available online 9 December 2017

Keywords:
API deprecation
Software evolution
Mining software repositories
Empirical software engineering

ABSTRACT

Libraries are commonly used to support code reuse and increase productivity. As any other system, they evolve over time, and so do their APIs. Consequently, client applications should be updated to benefit from better APIs. To facilitate this task, API elements should always be deprecated with replacement messages. However, in practice, there are evidences that API elements are deprecated without these messages. In this paper, we study questions regarding the adoption of deprecation messages. Our goal is twofold: to measure the real usage of deprecation messages and to investigate whether a tool is needed to recommend them. We assess (i) the frequency of deprecated elements with replacement messages, (ii) the impact of software evolution on this frequency, and (iii) the characteristics of systems that deprecate API elements in a correct way. Our analysis on 622 Java and 229 C# systems shows that: (i) on the median, 66.7% and 77.8% of the API elements are deprecated with replacement messages per project, (ii) there is no major effort to improve deprecation messages, and (iii) systems that deprecated API elements with messages are different in terms of size and community. As a result, we provide the basis for creating a tool to support clients detecting missing deprecation messages.

1. Introduction

In software development, it is common practice to implement systems on top of frameworks and libraries (Tourwé and Mens, 2003), taking advantage of their *Application Programming Interfaces* (APIs). This provides several benefits, for example: (i) reduction of development costs and time due to code reuse (Moser and Nierstrasz, 1996), (ii) increase focus on the essential system requirements, since developers do not need to re-implement the services provided by an API (Konstantopoulos et al., 2009), and (iii) increase software quality by using well-adopted, tested and documented code elements. Due to their advantages, APIs may have thousands of client applications. For example, with the support of Boa (Dyer et al., 2013), an infrastructure to support ultra-large-scale software mining on GitHub repositories, we found 143,454 client applications for `java.util.ArrayList`, 63,434 for `android.os.Bundle`, and 50,118 for `org.junit.Test`.

As any software system, frameworks/libraries and their APIs also evolve over time. Naturally, API elements (*i.e.,* public types, methods, and fields) may be renamed, removed, or updated. Consequently, impacted client applications should migrate to benefit from improved API elements (Bogart et al., 2016).

To facilitate client developers making the transition and preserve backward compatibility, API elements should be deprecated with replacement messages. Mechanisms to support API deprecation are provided by most programming languages, such as Java and C#. For example, Java presents two solutions to deprecate types, methods, and fields: using deprecation annotations and/or deprecation Javadoc tags. Both annotations and tags are used to warn developers referencing deprecated API elements. However, the latter may be accompanied by replacement messages to suggest what to use instead. Listing 1 presents an example of deprecated method in Java. In this example, method `getPostParams()` is deprecated with a `@Deprecated` annotation (line 4) and a Javadoc tag `@deprecated` (line 2). This tag contains a replacement suggestion for the deprecated method, which is method `getParams()`.

In practice, previous studies indicate that API elements are commonly deprecated with missing or unclear replacement messages. For example, Robbes et al. (2012) and

* Corresponding author.
E-mail addresses: gleison@dcc.ufmg.br (G. Brito), hora@facom.ufms.br (A. Hora), mtov@dcc.ufmg.br (M.T. Valente), rrobbes@unibz.it (R. Robbes).

https://doi.org/10.1016/j.jss.2017.12.007
0164-1212/

```
/**
 * @deprecated Use {@link #getParams()} instead.
 */
@Deprecated
protected Map<String, String> getPostParams() throws AuthFailureError { ... }
```

Listing 1. Deprecated method in Java - GOOGLE/IOSHED.

Sawant et al. (2016) investigate the impact of API deprecation in software ecosystems. Although it is not their main focus, the authors present preliminary evidences that APIs are usually deprecated without replacement messages. Hora et al. (2015b) investigate the impact of API evolution also at an ecosystem level; their results also highlight evidences that deprecation mechanisms should be more adopted. However, we still lack detailed information of API deprecation adoption. We are unaware about the real scale of this phenomenon, whether it tends to get better (or worse) over time, and characteristics of involved systems.

In this paper, we study a set of questions regarding the adoption of API deprecation messages. We analyze: the frequency of deprecated API elements with replacement messages; the impact of software evolution on the frequency of replacement messages; and 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 these messages. Thus, we propose the following research questions, which are answered in the context of 622 Java and 229 C# systems:

- **RQ1. What is the frequency of deprecated APIs with replacement messages?** We analyse the frequency of deprecated API elements with replacement messages. We detect that 66.7% of the API elements are deprecated with replacement messages per system in Java and 77.8% in C# (median values).
- **RQ2. What is the impact of software evolution on the frequency of replacement messages?** We assess the frequency of replacements messages by comparing multiple releases. Overall, we detect that there is almost no major effort to improve the quality of these messages over time.
- **RQ3. What are the characteristics of software systems with high and low frequency of replacement messages?** We investigate whether system popularity, size, community, activity, and maturity have an impact on the way developers deprecate API elements. We find that systems that follow best deprecation practices are statistically significant different from the ones that do not in terms of size, developing community, and activity.

Overall, we detect at a large-scale level that API elements are commonly deprecated without replacement messages. Therefore, we perform a follow up study to verify the feasibility of designing and implementing a recommendation tool that automatically infers replacement messages by mining real solutions adopted by developers. In this context, we computed a precision of 73%, *i.e.,* we are able to correctly infer missing replacement messages in almost 3/4 of the cases. To evaluate recall, we analyzed three real-world systems, resulting in the following values: 28.2%, 30.7%, and 37.5%. Thus, this suggests that a recommendation tool targeting elements deprecated without replacement messages is indeed possible to be implemented, with good precision and reasonable recall.

This work is an extension of our previous study (Brito et al., 2016). Specifically, this study extends the previous one in three major points: (1) we extend all research questions to investigate API deprecation in the context of C# programming language; (2) we provide new data analysis on RQ2 to investigate the impact of software evolution on the frequency of replacement messages; and (3) we provide a complementary study to assess precision and recall of a recommendation tool to automatically infer replacement messages. Thus, the contributions of this paper are summarized as:

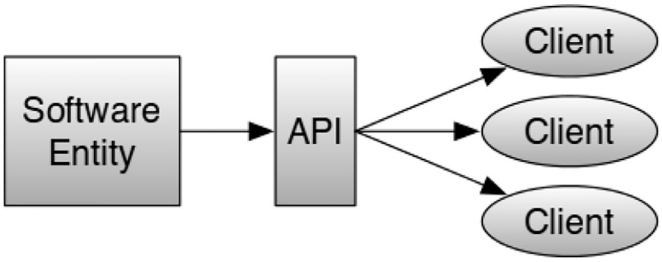

Fig. 1. APIs acting as interfaces between clients and a provider software entity (Montandon et al., 2013).

- We provide a large-scale empirical study covering two programming languages to understand to what extend APIs are deprecated with replacement messages.
- We provide evidences on the benefits of a recommendation tool to assist client developers in the detection of missing replacement messages.

In Section 2, we present the background in the context of APIs and deprecation. We describe our experiment design in Section 3 and we present the experiment results in Section 4. The complementary study on the recommendation tool to infer replacement messages is described in Section 5. Finally, we present related work in Section 6, and we conclude the paper in Section 7.

2. Background

2.1. Application programming interfaces

In this paper, we define *Application Programming Interfaces* (APIs) as interfaces used by software components to communicate with each other, as illustrated in Fig. 1. Examples of successful APIs include Java API,[1] .NET Framework Class Library,[2] and Android API.[3]

In this work, we consider API elements as public/protected types, fields, and methods. Listing 2 shows an API example, which presents an excerpt of the `Stack` class in package `java.util` for Java Platform Standard Edition.[4] In this example, methods `push()`, `pop()`, `peek()`, `empty()`, and `search` are public, thus, they are API elements. By contrast, `serialVersionID()` field is private, thus, it is not an API element. Also, the class `Stack` itself is an API element, because it is public.

2.2. API deprecation

Software systems evolve over time, changing their methods, fields, and types. When an API element changes, the impact propagates to client systems. To mitigate the impact of these changes, libraries and frameworks should use deprecation mechanisms, like

[1] https://docs.oracle.com/javase/8/docs/api/.
[2] http://msdn.microsoft.com/en-us/library/gg145045.aspx.
[3] http://developer.android.com/reference.
[4] http://www.oracle.com/technetwork/java/javase.

```
1  public class Stack<E> extends Vector<E> {
2
3      public E push(E item) {
4          ...
5      }
6      public synchronized E pop() {
7          ...
8      }
9      public synchronized E peek() {
10         ...
11     }
12     public boolean empty() {
13         ...
14     }
15     public synchronized int search(Object o) {
16         ...
17     }
18     private static final long serialVersionUID = 1224463164541339165L;
19 }
```

Listing 2. Example of API elements in class `java.util.Stack` in Java.

```
1  @Deprecated
2  public Database(Context context) { ... }
```

Listing 3. Deprecated method in Java using `@Deprecated` annotation - FACEBOOK/STETHO.

```
1  Note: Path\to\java\file.java uses or overrides a deprecated API.
2  Note: Recompile with -Xlint:deprecation for details.
```

Listing 4. Warning message caused by `@Deprecated` annotation.

```
1  /**
2  * Force a compile-time error on the old method of field definition
3  * @deprecated Use the required method getFieldOrder() instead to
4  * indicate the order of fields in this structure.
5  */
6  protected final void setFieldOrder(String[] fields) { ... }
```

Listing 5. Deprecated method in Java using the Javadoc tag `@deprecated` - JAVA-NATIVE-ACCESS/JNA.

messages to support client developers. In fact, API deprecation is a way to alleviate the impact of API changes. In theory, before being removed, changed, or renamed, API elements should be annotated as deprecated to support client developers making the transition to new ones. Deprecated API elements are kept in the system to preserve backward compatibility, but they should not be used by client developers because they may be removed in the future.

2.2.1. API deprecation in Java

The Java language, since J2SE 5.0, provides a mechanism to deprecate types, methods, and fields, using the `@Deprecated` annotation. This annotation causes the compiler to issue a warning when it finds references to deprecated API elements. Listing 3 shows an example of a deprecated method using the `@Deprecated` annotation. In this example, the `Database` method is deprecated with this annotation (line 1), but it is not included any suggestion for a replacement.

When an element is annotated with the `@Deprecated` annotation, the compiler will issue a deprecation warning if the element is used (*e.g.*,invoked, referenced, or overridden). The compiler will complain as the message shown in Listing 4:

When using deprecation annotations, it is a good practice to document the reasons for the deprecation and/or to recommend alternative API elements. To support this practice, Java provides the Javadoc tag `@deprecated` (supported since J2SE 1.1). The tag should also be used to warn developers about deprecated elements. Listing 5 shows an example of a deprecated method using the `@deprecated` tag (line 3). The tag contains a deprecation message that suggests replacing the deprecated method `setFieldOrder()` (lines 6–8) by the method `getFieldOrder()`.

More specifically, Java documentation recommends the use of two solutions to deprecate elements with replacement messages *i.e.*, message to suggest developers what to use instead, as follows:

- *Javadoc 1.1*: This Javadoc version recommends to use the annotation `@see` to indicate the replacement API.
- *Javadoc 1.2 and later*: These versions recommend to use the word `use` followed by the annotation `@link` to indicate the replacement API.

Listing 6 shows an example of deprecated Java method, according to Javadoc 1.2 and later guideline. Like in Javadoc 1.1, the tag `@deprecated` (line 3) is used to provide documentation to help the developers, but we also see the use of the `use` guideline to indicate the replacement element. Notice the use of the `@link` tag to provide a link to the documentation of the replacement element. The warning message "*Use @link #ScriptSortBuilder(Script, String) instead*" will be presented when developers compile a system that calls the deprecated method `lang()`. This message contains the suggestion for the replacement element and the link for its documentation. Notice that Java deprecation guidelines are not mandatory: developers may adopt other conventions to provide replacement messages, or simply do not use them at all.

2.2.2. API deprecation in C#

The C# programming language has the attribute `Obsolete` to deprecate API elements. Like `@Deprecated` annotations in Java, when an element has the `Obsolete` attribute, the C# compiler issues a message if it is used. The attribute contains two arguments: (i) a message to support developers and (ii) a parameter to define if the use of deprecated element should cause a compiler error. This attribute can be used with no arguments, but it is recommended to include an explanation on the reasons the el-

```
1  /**
2   * The language of the script.
3   * @deprecated Use {@link #ScriptSortBuilder(Script, String)} instead.
4   */
5  @Deprecated
6  public ScriptSortBuilder lang(String lang) { ... }
```

Listing 6. Deprecated method in Java using Javadoc 1.2 guidelines - ELASTIC/ELASTICSEARCH.

```
1  [Obsolete("Use ApiTaskAsync instead", true)]
2  protected virtual void ApiAsync(HttpMethod httpMethod, string path, object
       parameters, Type resultType, object userState) { ... }
```

Listing 7. Deprecated method in C# using `Obsolete` attribute with a replacement message - FACEBOOK-CSHARP-SDK/FACEBOOK-CSHARP-SDK.

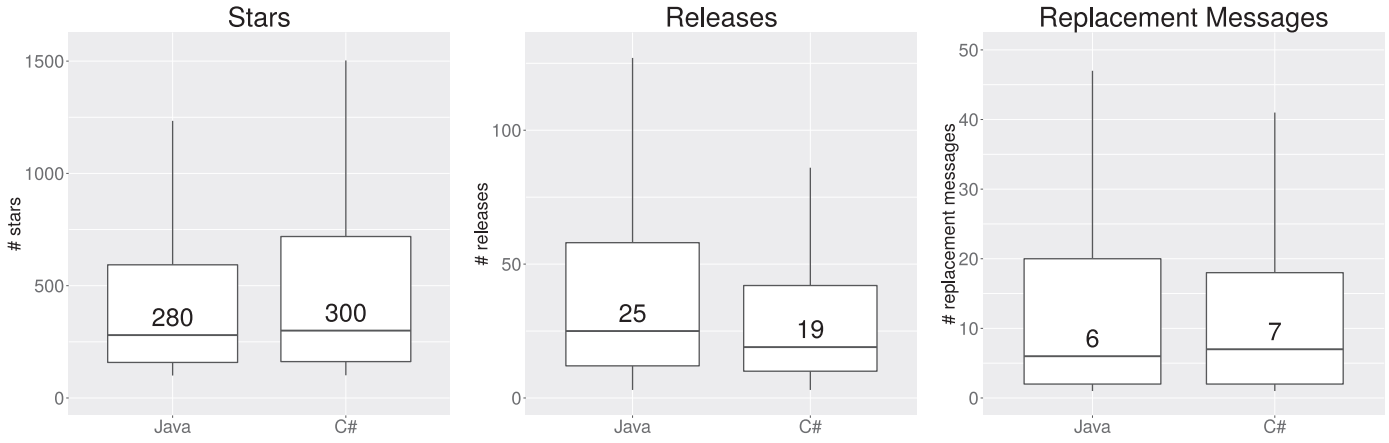

Fig. 2. Distribution of number of stars, number of releases, and number of deprecated API elements in the selected systems.

ement is obsolete and what API element should be used as a replacement. Listing 7 presents an example of a deprecated method in C#, where the `Obsolete` attribute is used with a replacement message to support client systems.

3. Study design

3.1. Selecting case studies

We analyse Java and C# systems hosted on GitHub, the most popular social coding platform nowadays. We use three filtering criteria to select real systems and discard irrelevant ones (Kalliamvakou et al., 2014): number of stars, releases, and deprecated API elements.

1. **Number of stars.** GitHub provides the stargazer button that allows users to show interest on systems. We select systems with 100 or more stars in order to only take into account popular and real-world ones.
2. **Number of releases.** GitHub has the ability to tag specific points in history in order to facilitate release creation.[5] We select systems with three or more public releases (tags) available on GitHub. We use this criterion to assess API deprecation evolution.
3. **Number of deprecated API elements with replacement messages.** We select systems with at least one public/protected deprecated API element with replacement message. We use this criterion to filter out systems without replacement messages, which are not in the scope of our study.

We then selected all the systems that satisfied our filtering criteria: **622 in Java and 229 in C#**. To better characterize them, Fig. 2 presents the distribution of the three aforementioned measures. For Java systems, the number of stars in the first quartile, median, and third quartile is 158, 280, and 593. For C#, the number of stars in the first quartile, median, and third quartile is 162, 300, and 719, respectively. The top-3 Java systems with more stars are ELASTIC/ELASTICSEARCH (12.4K stars), NOSTRA13/ANDROIDUNIVERSALIMAGELOADER (9.7K), and GOOGLE/IOSCHED (7.7K). For C#, the top-3 systems with more stars are DOTNET/COREFX (9.2K), SIGNALR/SIGNALR (5.6K), and DOTNET/ROSLYN (4.5K). Also, for Java, the number of releases in the first quartile, median, and third quartile is 12, 25, and 58, while for C# it is 10, 19, and 42. Finally, for Java, the number of API elements deprecated with replacement messages in the first quartile, median, and third quartile is 2, 6, and 20, and for C# it is 2, 7, and 18.

In addition, we manually classify the selected systems in two categories: *library* and *non-library*. In fact, library developers are expected to take more care when evolving APIs. We define systems in library category after inspecting their description and documentation on GitHub. In this case, we payed special attention in explanations including keywords such as *library, framework, API*, and *interface*. Other systems are classified as non-library. For Java, we classified 342 (54.98%) as libraries and 280 as non-libraries (45.02%). For C#, we detected 119 (51.97%) libraries and 110 (48.03%) non-libraries.

3.2. Extracting deprecated API elements

As a first step to support answering our research questions, we extract all API elements (including types, fields, and methods) with deprecation annotations in Java and C# systems. For Java, we extract their associated Javadoc and, for C#, we extract the Obsolete attribute. Listing 8 presents an example of a deprecated method in Java. In this example, the `@deprecated` annotation (line 6) is used to deprecate method `onModule()`. Additionally, a Javadoc annotation is used to describe a replacement for the deprecated method: instead of calling `onModule()`, ELASTIC/ELASTICSEARCH developers should now call `onIndexModule()`.

[5] http://www.git-scm.com/book/en/v2/Git-Basics-Tagging

```
1 /**
2 * Old-style guice index level extension point
3 *
4 * @deprecated use #onIndexModule instead
5 */
6 @Deprecated
7 public final void onModule(IndexModule indexModule) { ... }
```

Listing 8. Example of obsolete method in Java - ELASTIC/ELASTICSEARCH.

```
1 [Obsolete("Please, use IEqualityComparer insteaded.")]
2 public interface IHashCodeProvider { ... }
```

Listing 9. Example of obsolete method in C# - MICROSOFT/CODECONTRACTS.

Table 1
Regular expressions to identify API elements in C#.

Element	Expression
Field	`(public|protected)\\s+"(\\w+)=(?:\")?(.*?(?=\"?\\s+\\w+=|(?:\"?)\$))`
Property	`(\S+(?:<.+?>)?)(?=\s\w+\s\{get;)`
Method	`((public|protected|static|final|native|synchronized|abstract|transient)+\\s)+[\\$_\\w\\<\\>\\[\\]]*\\s+[\\$_\\w]+\\([^\\)]*\\)?\\s*\\{?[^\\}]*\\}`
Type	`"\\s*(public|protected)\\s+class\\s+(\\w+)\\s+((extends\\s+\\w+)|(implements\\s+\\w+( ,\\w+)*))?\\s*\\{"`

Table 2
Number of deprecated API elements.

Language	Deprecated Types	Deprecated Fields	Deprecated Methods	All Deprecated Elements
Java	5,814	4,521	26,727	37,062
C#	1,277	2,197	4,723	8,197

Listing 9 shows a C# deprecation example: the attribute (line 1) is used to deprecate the type named `IHashCodeProvider`. Note that the `Obsolete` attribute suggests a replacement for the deprecated type. Instead of using `IHashCodeProvider`, MICROSOFT/CODECONTRACTS developers should now use `IEqualityComparer`.

To find deprecated API elements in Java, we implemented a parser based on the Eclipse JDT library to look for deprecation annotations and tags. To find deprecated API elements in C#, we implemented an in-house tool based on lexical analysis to detect deprecation attributes. This tool uses regular expressions to identify methods, fields, and types, checking whether they are deprecated. Table 1 shows the regular expressions used by the tool. In our study, we consider C# property elements as fields.

Finally, we restricted our analysis to public and protected API elements because they represent the external contracts to clients. Moreover, when an entire type is deprecated, we did not consider their contained methods and fields are also deprecated (unless these methods or fields are explicitly deprecated). Table 2 shows the number of public and protected deprecated API elements.

3.3. Extracting replacement messages

In Java, when an element is deprecated using the Javadoc tag, it may be accompanied by a replacement message to help client developers. As presented in Section 2.2, Java guidelines propose two solutions to create deprecation replacement messages: (i) using the annotation `@see`, or (ii) using the word `use` and the annotation `@link`. However, to detect alternative guidelines followed by Java developers, we extracted deprecation messages with the support of the JDT library, and we manually inspected a subset of these messages. Specifically, we randomly selected deprecated messages until we found 200 with replacement messages (regardless of the project); these 200 were the ones inspected to detect the guidelines. As a result of this analysis, we detected seven frequent guidelines to indicate replacement, in addition to `use`: `refer`, `equivalent`, `replace`* (*i.e.,* `replace`, `replaced`, `replacement`), `see`, `moved`, `instead`, and `should be used`. We also confirmed the usage of annotations `@link` and `@see`.

Table 3 shows the frequency of guidelines as well as message examples. The most adopted guideline is `use`, with 17,810 cases (47.9%). In contrast, the least adopted one is `should be used`, with 33 cases, (0.09%). Notice that some guidelines may co-occur in the same message. For example, `use` commonly happens with `@link`. In total, 22,075 (59.5%) API elements were deprecated with replacement messages out of 37,119 Java API elements.

In contrast to Java, we could not find guidelines to define replacement messages for C#. For this reason, we also performed a similar manual analysis to detect replacement guidelines in C#. First, we extracted deprecation messages in deprecated API elements. We looked for occurrences of the attribute `Obsolete` and we manually inspected a subset of these messages. As a result, we detected three frequent guidelines that are used to indicate replacement: `use`, `replace`*, and `instead`. Table 4 presents the frequency of each identified replacement in C#, with examples of replacement messages. The most common guideline is `instead`, with 3,118 cases (51%). In contrast, the least adopted one is `replace`*, with 422 cases (8%). In total, 5,268 (64.3%) API elements are deprecated with replacement messages. This data is further explored in RQ1, and its evolution is analyzed in RQ2.

Table 3
Frequency of replacement guidelines in Java.

Guideline	Frequency	Example
use	17,810 (47.9%)	use encodeURL(String url) instead (Apache Tomcat)
@link	14,852 (40%)	Use @link #setController(DraweeController) instead (Facebook Fresco)
instead	14,173 (38.2%)	Use KEY_LMETA instead (Facebook Nifty)
@see	2,334 (6.3%)	@see #getStartRequests (WebMagic)
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)
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)
equivalent	166 (0.3%)	The @link Iterable equivalent is @link ImmutableSet#of() (Google Guava)
should be used	33 (0.09%)	org.bukkit.entity.minecart.PoweredMinecart should be used instead (Bukkit)

Table 4
Frequency of replacement guidelines in C#.

Guideline	Frequency	Replacement Message Example
instead	3,118 (51%)	This class is obsolete; use class Tree instead (Mono)
use	2,686 (49%)	Use static Add() method instead. (Mono Monomac)
replace*	422 (8%)	Replace it with both GetSupportedInterfaceOrientations (Redth/ZXing.Net.Mobile)

Table 5
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)

3.4. Comparing systems with high and low frequency of replacement messages

3.4.1. Defining metrics possibly impacting API deprecation

To support answering RQ3, about the characteristics of systems that deprecate API elements with replacement messages, we consider metrics in five dimensions that may affect deprecation practices: popularity, size, community, activity, and maturity. The goal is to investigate whether these metrics have an impact on the way developers deprecate API elements. The metrics are described next and summarized in Table 5.

- **Popularity.** This dimension includes metrics that represent how popular is a 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, their developers might have more concerns about their APIs.
- **Size.** This dimension 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 are harder to maintain, therefore, it might 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.** This dimension includes metrics that represent the system community, including *number of contributors, average files per contributor*, and *average API elements per contributor*.[6] The rationale is that systems with larger communities might be easier to maintain, and to keep track of API changes.
- **Activity.** This dimension includes metrics related to the system activity level in terms of *number of commits, number of releases*, and *average days per release*. The rationale is that systems with more activity might respond faster to client complains. Therefore, they may be more likely to improve their APIs.
- **Maturity.** This dimension is about the system age, in *number of days*. The rationale is that older systems are reliable, thus, they may have stable APIs. In contrast, it is natural to expect that newer systems have less stable APIs.

3.4.2. Extracting metrics from case studies

We extracted the proposed metrics from two groups of systems, considering their last release: the ones deprecating API elements in a correct way, by providing replacement messages to deprecated API elements (named *top* systems), and the ones not following this practice (named *bottom* systems). Then, we assessed these groups to verify whether they are statistically different with respect to the proposed metrics. These two steps are detailed next.

Selecting top and bottom systems. We first sorted all systems, in descending order, based on the percentage of deprecated API elements with replacement messages. We selected two groups, top-30% (*i.e.,* systems with the highest percentage of deprecated API elements with replacement messages) and bottom-30% (*i.e.,* systems with the lowest percentage). For Java, each group has 187 systems and for C# each group has 69 systems. Fig. 3 shows the relative distribution of deprecated elements with replacement messages in each group. In the Java systems, the median percentage is 100% for the *top* systems and 17.8% for the *bottom* ones. For C# systems, the median percentage is also 100% for the *top* systems and 42.5% for the *bottom* ones.

Extracting metrics and comparing systems. We extracted the metrics described in the previous subsection for the *top* and *bottom* systems and then compared the obtained values. We first analyse the statistical significance of the difference between the two groups by applying the Mann–Whitney U test at $p\text{-}value = 0.05$. To show the effect size of the difference between the two groups, we compute Cliff's Delta (or d). As in previous studies (Grissom and Kim, 2005; Tian et al., 2015; Linares-Vásquez et al., 2013), we in-

[6] "average files per contributor" = number of files in last release/number of contributors. "average API elements" = number of APIs in last release/number of contributors.

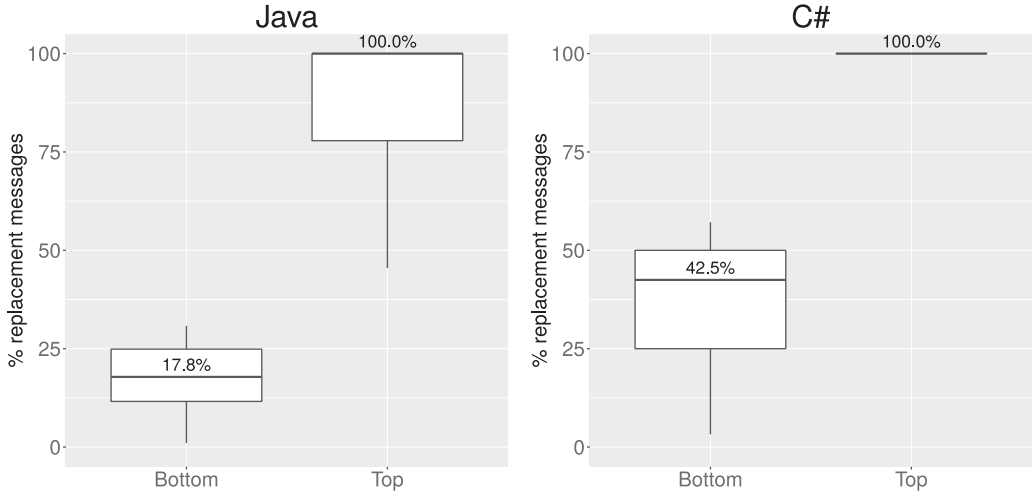

Fig. 3. Distribution of the percentage of deprecated APIs with replacement messages in the top-30% and bottom-30% systems.

Table 6
Number of deprecated API elements with replacement messages when considering all projects. Between parentheses, global means.

Language	Types	Fields	Methods	All
Java	3,789 (65%)	2,675 (59%)	15,568 (58.2%)	22,032 (59.4%)
C#	613 (48%)	1,401 (63.8%)	3,254 (68.9%)	5,268 (64.2%)

terpret the effect size values as small for $0.147 < d < 0.33$, medium for $0.33 < d < 0.474$, and large for $d > 0.474$.

4. Results

In this section, we answer and discuss the three research questions proposed in this study. Section 4.1 discusses the frequency of deprecated APIs with replacement messages (RQ1). Section 4.2 investigates the impact of software evolution on the frequency of such messages (RQ2). Section 4.3 describes the system characteristics that may impact on the way developers deprecate API elements (RQ3). Section 4.4 presents some final remarks. Finally, Section 4.5 presents threats to the validity of our study.

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

In this first research question, we analyze the frequency of deprecated API elements with replacement messages in the last release of the Java and C# systems, when considering all projects. As presented in Table 6, in Java, 3,789 deprecated types (65%) contain replacement messages. For deprecated fields and methods, these numbers are 2,675 (59%) and 15,568 (58.2%). When considering all deprecated API elements, 22,032 (59.4%) contain replacement messages. We also present the frequency of deprecated API elements with replacement messages in the C# systems. As noticed in Table 6, 613 deprecated types (48%) in the C# systems contain replacement messages. For deprecated fields and methods, these numbers are 1,401 fields (63.8%), and 3,254 methods (68.9%). Considering all deprecated API elements in C#, 5,268 (64.2%) contain replacement messages. Overall, the results measured for Java and C# are similar, although C# systems have a slight tendency to include more replacement messages, in relative terms. This may be explained by the fact that both languages have explicit and official mechanisms to deprecate API elements with messages, which do not always happen in other programming languages.

Absolute analysis. Fig. 4 shows the distribution of the number of deprecated API elements with replacement messages per system. For types, the median is 2 in Java and 1 in C#. Regarding fields elements, the median is 1 in Java and 2 in C#. For methods, the median is 5 in both languages. Considering all API elements in Java, the first quartile is 2, the median is 6, and the third quartile is 20; for C#, the values are 2, 7, and 18. Therefore, methods are the most frequently deprecated elements with replacement messages in both languages while fields are the least ones in Java and types are the least one in C#.

Relative analysis. Fig. 5 presents the distribution of the relative number of deprecated API elements with replacement messages per system, which is detailed below:

- Types: In Java, the median is 71.4%: there are 114 systems with 100% of their deprecated types with replacement messages. In contrast, we find 57 systems with deprecated types without these messages. For C#, the median is 75%: we detect 61 systems with all deprecated types with replacement messages, and 30 without.
- Fields: In Java, the median is 50%: we identify 74 Java systems with 100% of their deprecated fields with replacement messages and 72 systems without any message. For C#, the median is 75%: there are 26 systems with 100% of fields with replacement messages and 14 systems without any message.
- Methods: In Java, the median is 66.7%: there are 160 Java systems with 100% of their deprecated methods with replacement messages and 24 systems with no replacement messages. For C#, the median is 75%: we found 63 systems with 100% of their deprecated methods with replacement messages and 20 without.
- All: In Java, the first quartile is 33.3%, the median is 66.7%, and the third quartile is 100%; we also found 162 systems (26%) with 100% of their deprecated API elements with replacement messages. For C#, the first quartile is 50%, the median is 77.8%, and the third quartile is 100%; we found 64 systems (28%) with 100% of deprecated API elements with replacement messages.

In summary, in Java, types are the most deprecated elements with replacement messages, followed by methods and fields. The third quartile at 100% for all elements shows that 25% of the systems always deprecate all elements with replacement messages. For C#, like in Java, the third quartile is 100% for all elements, and coincidentally, all three elements have the same median values (75%). We notice that C# systems have a slight trend to include more replacement messages, when compared to the Java ones (me-

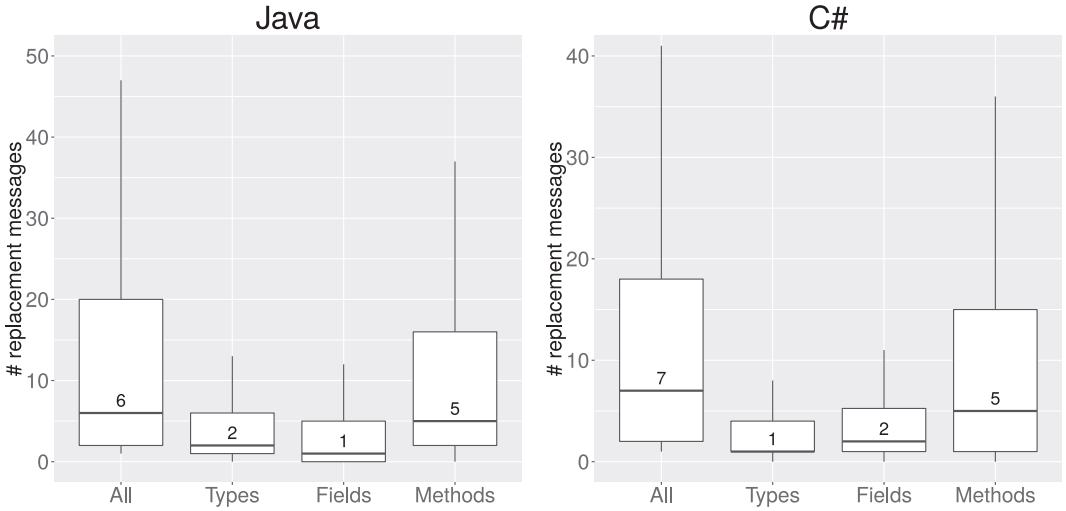

Fig. 4. Absolute distribution of deprecated API elements with replacement messages.

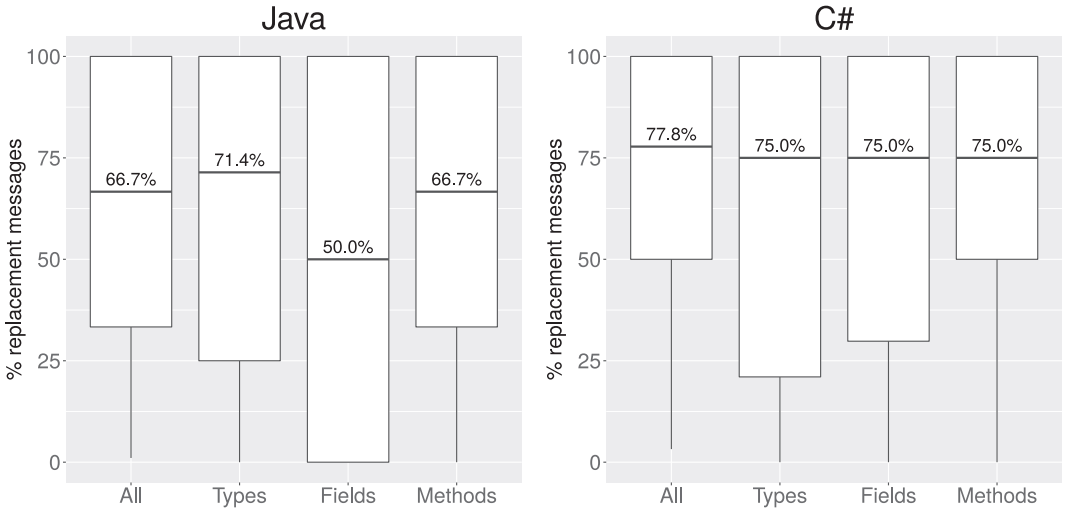

Fig. 5. Relative distribution of deprecated API elements with replacement messages.

dian per project 77.8% against 66.7%). Overall, we also observe a relevant number of API entities deprecated without replacement messages, which may make client application migration more difficult.

Libraries vs. Non-libraries. As described in Section 3.1, we classify the systems in two categories: library and non-library. For Java, we found 342 libraries and 280 non-libraries; in C#, 119 and 110. To compare the categories, we analyze the relative number of deprecated messages per system in each group (Fig. 6). For Java, the first quartile, median, and third quartile are 40%, 71.2%, and 100%, for libraries. For non-libraries, these values are 28.6%, 60%, 86.8%, respectively. For C#, the numbers for libraries are 51.3%, 81.8%, and 97.2%. For non-libraries, these values are 50%, 73.7%, and 100%. The difference between the median values of the two categories is 11.2% (Java) and 8.1% (C#).[7] In both languages, the percentage of replacement messages for libraries is greater than in non-libraries.

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

To study the impact of software evolution on API deprecation, we analyze the frequency of deprecated API elements with replacement messages in several distinct releases of the Java and C# systems. Starting from the first release, we collected all the subsequent ones, considering an interval of at least two months. For each release, we assess the frequency of deprecated API elements with replacement messages that existed in that release, regardless of when they were introduced. We then classify the variation in the percentage of replacement messages of a given system release in four categories: (i) *Decrease Trend*: the percentage always decreases in the analysed releases; (ii) *Increase Trend*: the percentage always increases; (iii) *Variant Trend*: the percentage increases and decreases with no pattern; and (iv) *Stable Trend*: the percentage of replacement messages is constant in all releases. Fig. 7 provides examples of systems in each category.

Table 7 presents the number of systems in each category. For Java, 40 systems (6%) are in the decrease category, 198 systems (32%) in the increase category, 245 systems (40%) in variant category, and 139 systems (22%) are stable. The values for C# are 25

[7] We found statistically significant difference with medium effect size in the Java comparison ($p\text{-value} < 0.01$ and effect-size = 0.35).

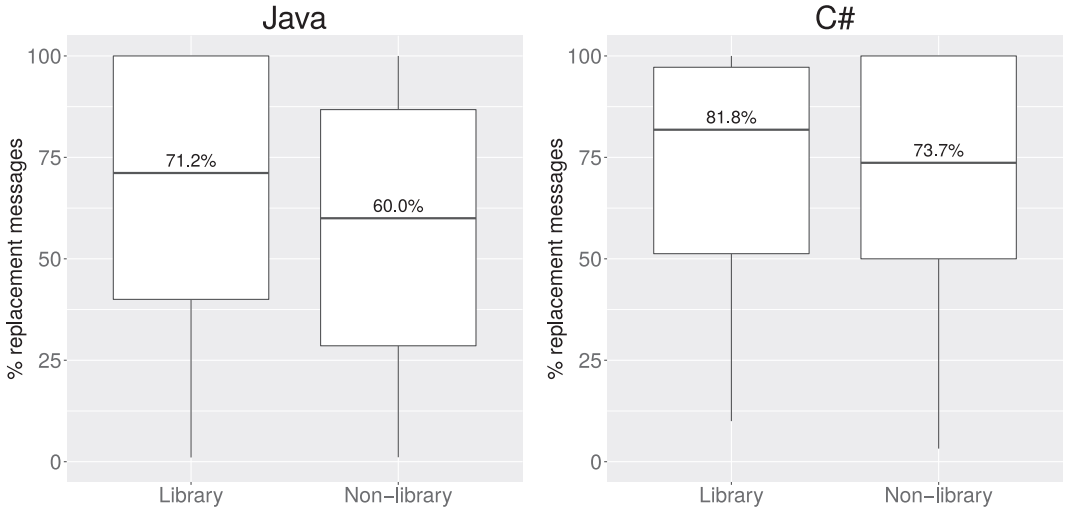

Fig. 6. Relative distribution of deprecated API elements with replacement messages in library and non-library systems.

Table 7
Number of systems in each evolution category.

Language	Decrease	Increase	Variant	Stable
Java	40 (6%)	198 (32%)	245 (40%)	139 (22%)
C#	25 (11%)	59 (26%)	83 (36%)	62 (27%)

systems (11%) for decrease category, 59 systems (26%) for increase, 83 systems (36%) for variant, and 62 systems (27%) for stable. Thus, most systems are variant, while a small amount has a decrease trend in the percentage of replacement messages. Only a minority of the systems loses the quality of their deprecation messages over time.

In addition, Fig. 8 shows the distribution of the number of system releases classified in each evolution category. We performed this analysis to reveal how the number of releases impacts in the evolution of replacement messages. For Java, the values are 18 (decrease), 18 (increase), 14 (stable), and 48 (variant). The values in C# are 22 (decrease), 14 (increase), 11 (stable), and 40 (variant). In both languages, stable systems have the lowest number of releases on the median, while variant systems have the highest number of releases. This may explain the fact the latter ones vary the number of deprecated API elements, for example, by removing them more aggressively.

In summary, 32% of the Java systems increase the amount of replacement messages, while only 6% decrease. For C#, 26% of the systems increase this percentage, and 11% decrease. Overall, most systems increase and decrease the percentage of replacement messages, showing their deprecation quality are not maintained over time.

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

In this research question, we assess the last release of our case studies to investigate whether system popularity, size, community, activity, and maturity have an impact on the way developers deprecate API elements, as indroduced in Section 3.4. We perform this investigation by comparing *top* and *bottom* systems; *top* systems have 100% of their API elements deprecated with replacement messages, while *bottom* barely do that. Table 8 presents the metrics and their respective *p-values* and *d* applied on *top* and *bottom* systems, for both languages. 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.

For Java, the selected *top* and *bottom* systems are statistically significant different with at least a small effect size in 7 out of the 12 metrics, including all size and activity metrics as well as number of contributors and average files per contributor in community. The effect size is large in one metric (number of commits), medium in three (number of files, number of API elements, and number of contributors), and small in three (number of contributors, average files per contributor, and average days per release). Regarding the C# systems, 8 metrics present statistical significance: all popularity and size metrics, and number of contributors, number of commits, and number of releases. The effect size is large in one metric (number of contributors), medium in six (all popularity and size metrics, and number of commits), and small in one (number of releases). In the following, we investigate each dimension.

- **Popularity.** In the Java systems, we detect that there is no difference in *top* and *bottom* systems with respect to the popularity metrics. In contrast, for C# systems, we note that all popularity metrics have statistical difference. According to the relationship column, popular C# systems provide less replacement messages in deprecated APIs.
- **Size.** For both languages, we observe that *top* systems are smaller than *bottom* ones, as measured both in *number of files* and *number of API elements* (notice the "– " on the relationship column). In fact, it is intuitive to consider that smaller systems are easier to maintain and to keep track of API elements, which also facilitates the provision of replacement messages.
- **Community.** We can see that *top* systems have less contributors than *bottom* ones, for both languages. This result is somehow related to the previous one: it is expected that smaller systems have less contributors. However, Java systems with fewer *files per contributor* are more likely to have replacement messages. For C#, *top* systems also have fewer contributors than *bottom* ones. In opposite to Java, the ratio of *files per contributor* has no significance.
- **Activity.** For the activity dimension, we observe that *top* systems have less commits and releases than *bottom* ones in both languages. An explanation is that as *bottom* ones have more code changes, they may be more likely to degrade their APIs. We also notice that Java systems released in short period are more likely to deprecate APIs with replacement messages.

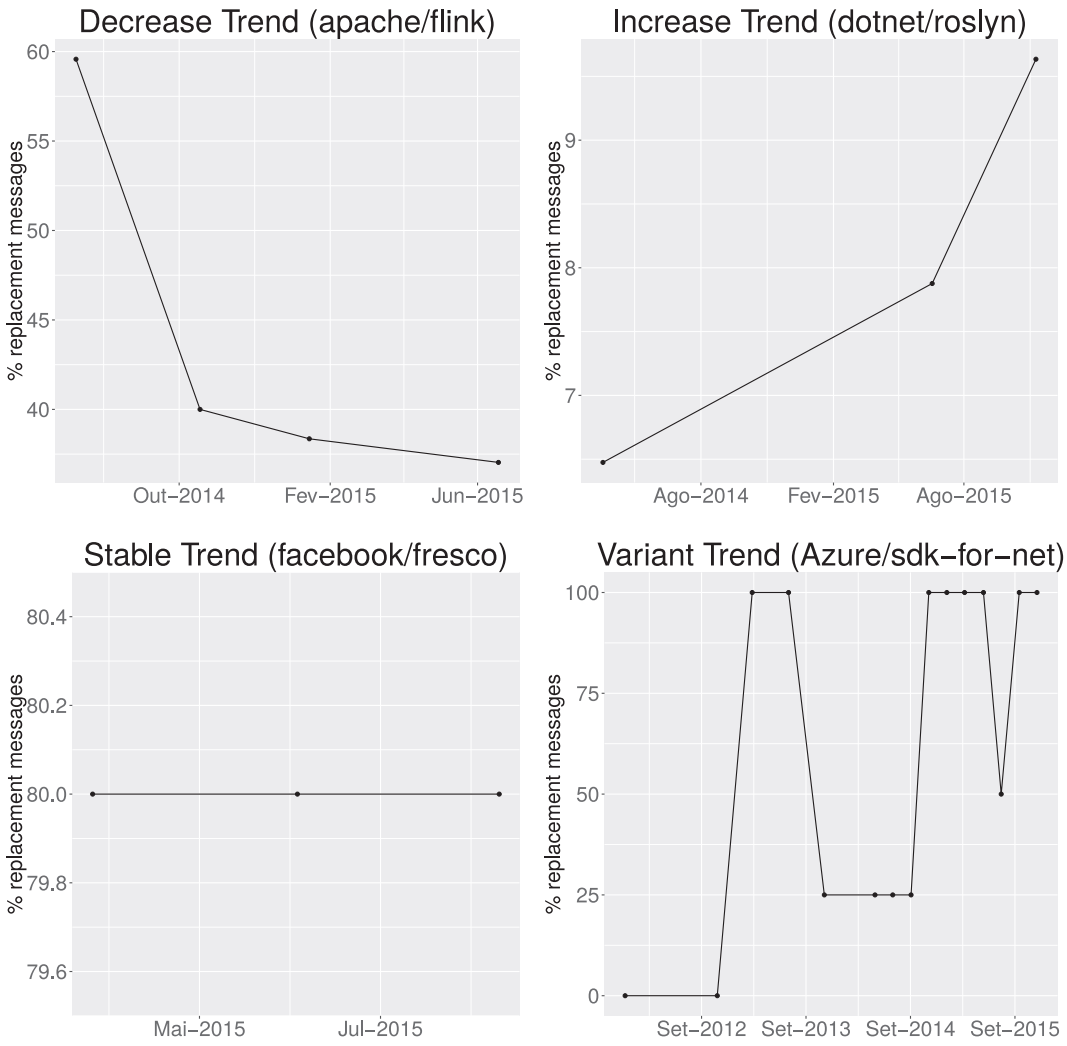

Fig. 7. Examples of systems in each evolution category.

Table 8
Metrics and their respective *p-values* and *d* on *top* and *bottom* systems. Bold values mean *p-value* $< .05$ (statistically significant different), and $d > 0.147$ (at least a small effect size). Level of significance for d-values: L = large, M = medium, S = Small, N = negligible. Rel. = relationship: "+" = *top* systems have significantly higher value on this metric. "− " = *bottom* systems have significantly higher value on this metric.

Dimension	Metric	Java			C#		
		p-value	d-value	Rel.	*p*-value	d-value	Rel.
Popularity	Number of stars	.674	0.142 (N)	+	**.004**	**0.407 (M)**	−
	Number of watchers	.018	0.04 (N)	+	**<.001**	**0.454 (M)**	−
	Number of forks	.028	0.08 (N)	+	**.003**	**0.447 (M)**	−
Size	Number of files	**<.001**	**0.459 (M)**	−	**.034**	**0.419 (M)**	−
	Number of API elements	**<.001**	**0.374 (M)**	−	**<.001**	**0.414 (M)**	−
Community	Number of contributors	**<.001**	**0.376 (M)**	−	**.009**	**0.537 (L)**	−
	Avg. files per contrib.	**<.001**	**0.227 (S)**	−	.648	0.160 (N)	−
	Avg. API elem. per contrib.	<.001	0.123 (N)	−	.303	0.110 (N)	−
Activity	Number of commits	**<.001**	**0.563 (L)**	−	**.009**	**0.342 (M)**	−
	Number of releases	**.001**	**0.206 (S)**	−	**.014**	**0.244 (S)**	−
	Avg. days per release	**.004**	**0.182 (S)**	−	.201	0.292 (S)	−
Maturity	Age (in number of days)	.253	0.009 (S)	+	.102	0.363 (M)	-

- **Maturity.** For both languages, we could not find relevant differences between *top* and *bottom* systems with respect to their maturity (*i.e.,* age in number of days). Although, someone might expect older systems to be more stable and to provide better APIs, in fact, we concluded that system age has no effect on the way developers deprecate API elements with replacement messages.

In summary, for Java, top systems are statistically different from bottom ones in 7 out of 12 metrics. Top systems tend to be smaller in terms of number of files and API elements, but have more contributors per file. For C#, we found significance in 8 metrics; popularity and size impact the way developers deprecate their APIs. In both languages, system maturity has no effect on the way developers deprecate API elements with replacement messages.

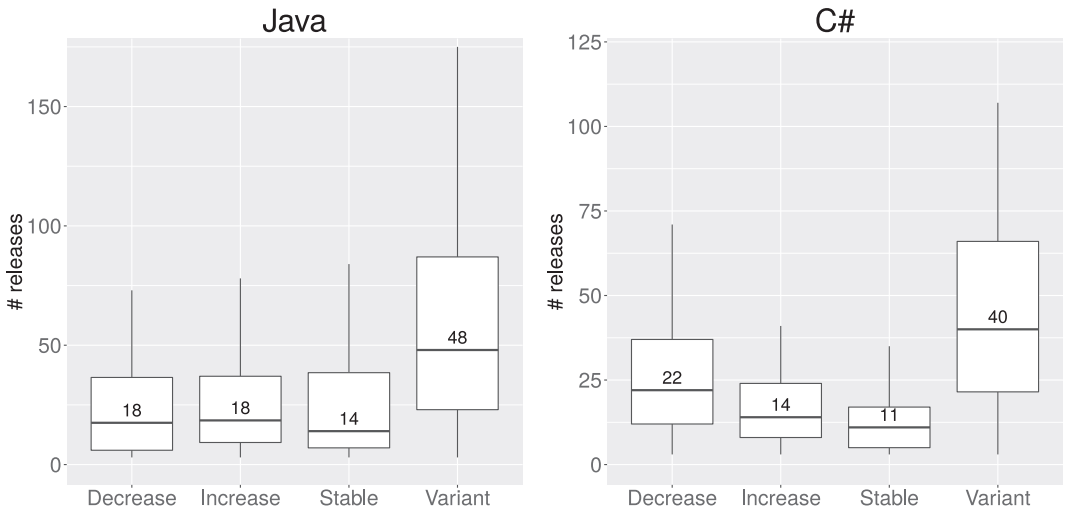

Fig. 8. Number of system releases classified in each evolution category.

4.4. Discussion and implication

From our analysis on 622 Java and 229 C# systems, we discuss the major findings and implications. RQ1 shows that 66.7% of the API elements are deprecated with replacement messages per system for Java, on the median. This percentage is 71.4% for types, 50% for fields, and 66.7% for methods, suggesting that developers are more concerned with types and less with fields. RQ1 presents that 25% of the systems always deprecate all types, fields, and methods with replacement messages. For C#, the values for fields, methods, and types are 75%. When considering all API elements, the median percentage of replacement messages is 77.8%. Finally, RQ1 also shows a comparison between libraries and non-libraries systems. In Java systems, values for libraries are 71.2%, and for non-libraries, 60%. For C#, the numbers are 81.8%, and 73.7%.

RQ2 presents that 32% of the analyzed Java systems increase the number of replacement messages, while only 6% decrease. The percentages are not very different for C#: 26% of the systems increase this percentage, and 11% decrease. Overall, most systems are placed on the variant category where they increase and decrease the percentage of replacement messages over time.

Finally, RQ3 shows that top systems (*i.e.,* the ones with more quality in their replacement messages) are statistically significant different from bottom projects (*i.e.,* the ones with less quality in their replacement messages) in several of the considered metrics. Top systems in Java are smaller in terms of number of files, API elements, and community (they have less contributors, but more contributors per file). Surprisingly, system popularity and maturity have no effect on the way developers deprecate API elements with replacement messages. For C#, top systems are also smaller in terms of files, API elements, community and have less activity.

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 (Wu et al., 2010; Robbes et al., 2012). In fact, there is an effort in the literature to understand the impact of software evolution on APIs (McDonnell et al., 2013; Robbes et al., 2012; Hora et al., 2015b; Xavier et al., 2017) and to detect how this impact can be alleviated by mining client reactions (Dagenais and Robillard, 2008; Schäfer et al., 2008; Kim and Notkin, 2009; Wu et al., 2010; Meng et al., 2012; 2013; Bogart et al., 2016; Sawant et al., 2016; Hora et al., 2017). 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, and that this situation does not get much better over time, we present an implication of our findings:

> *Implication*: A recommendation tool should 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 Section 5, we design and assess a prototype of this recommendation tool to support client developers by automatically inferring missing replacement messages.

4.5. Threats to validity

4.5.1. Construct validity

Construct validity is related to whether the measurement in the study reflects real-world situations. One threat of our study is that deprecated API elements may be incorrectly classified as having or not having replacement messages. To assess this threat, we performed two analyses to assess false-positives and false-negatives. First, we manually analysed 500 randomly selected deprecation messages classified as having replacement messages (regardless of the project). For Java, we detected 4 false-positives ($<1\%$), *i.e.,* messages incorrectly classified as including replacement information. For C#, we found 1 ($<1\%$) false-positive. Second, we also manually analysed 500 randomly selected messages classified as not having replacement messages (again, regardless of the project) For Java, we detected 26 (5%) false-negatives, *i.e.,* messages incorrectly classified as not including replacement information. For C#, we found 15 (3%) false-negatives. Therefore, the risk of false-positives and false-negatives in our classification is very low.

4.5.2. Internal validity

This threat is related to uncontrolled aspects that may affect experimental results.

Findings Validation. We paid special attention to the appropriate use of statistical machinery (*i.e.,* Mann–Whitney test and Cliff's Delta effect size) when reporting our results in RQ3. This reduces the possibility that these results are due to chance.

Correlation is not Causation. In RQ3, we examined whether there are metrics associated with top and bottom systems. Notice, however, that correlation does not imply causation. Thus, more advanced statistical analysis, *e.g.,*causal analysis (Couto et al., 2014), can be adopted to further extend our analysis.

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

C# In-house Tool Implementation. Another possible threat is the possibility of errors in our C# in-house tool to identify deprecated elements. To evaluate this threat, we manually analysed the tool precision and recall in three systems: FACEBOOK-CSHARP-SDK/FACEBOOK-CSHARP-SDK (26 deprecated elements), ANTARIS/RAZORENGINE (62 deprecated elements) and AZURE/AZURE-STORAGE-NET (107 deprecated elements). For the three systems, the tool found all deprecated elements (both precision and recall are 100%). Thus, the risk of this threat is also low.

4.5.3. External validity

External validity is related to the possibility to generalize our results. We focused on the analysis on 622 Java and 229 C# open-source systems, which are representative case studies. These systems are hosted in GitHub, the most popular code repository nowadays. Despite these observations, our findings cannot be directly generalized to other systems, specifically to systems implemented in other programming languages or commercial ones. Future replications should address this issue.

5. Inferring missing replacement messages

5.1. Motivation

We detected that the median number of replacement messages with deprecated API elements per project is 66.7% for Java and 77.8% for C#. This shows that a large number of replacement messages are missing. Moreover, the percentage of API elements deprecated with replacement messages barely increases over time. This scenario shows that developers may benefit from an approach to find missing replacement messages, which could improve their maintenance practices. Specifically, it might be possible to design and implement a recommendation tool that automatically infers replacement messages by mining real solutions adopted by developers (as briefly presented in the implication in Section 4.4). That is, even when there is no explicit replacement message, API clients may take their own decisions to replace a deprecated API element by another one. Thus, a recommendation tool could learn these decisions, particularly when a common decision is followed by many clients.

For example, suppose that type T from an API A is deprecated without a replacement message. Consider also that $C_1, C_2, \ldots, C_n$ are clients of the API A that reference the deprecated type T. In this context, suppose also that along their version history most clients C_i replaced these references to another type T'. Therefore, in this case, a recommendation tool can suggest that a deprecated message like "use type T' instead" should be added to the declaration of T.

In this section, we investigate the feasibility of designing and implementing such a tool. To this purpose, we rely on data provided by APIWAVE (Hora and Valente, 2015), which is a tool to assist client developers on evolving their systems to newer or improved APIs. APIWAVE provides data about API migration at type level, which is mined from the type usage differences between two versions of a class. By mining the import statements of these versions, the tool infers that a type T was replaced by a type T'. The current version of APIWAVE includes data about the evolution of top-1,000 most popular GitHub Java projects, from which 320K packages and types are extracted. Finally, the tool provides a ranking with the most common API migrations, which is used to support the study described in this section.

5.2. Study design

5.2.1. Dataset

To support the proposed study, we collect the top-3,000 API migration rules, as provided by APIWAVE. These rules have the format: $T \rightarrow T'$, expressing that type T (left side) is commonly replaced by type T' (right side), in the 1,000 GitHub projects mined by APIWAVE. Next, we detail examples of the detected rules:

- *junit.framework.Assert* → *org.junit.Assert*. This is the most popular rule. This migration occurred in JUNIT-TEAM/JUNIT, in version 4.0. Type *Assert* was moved to package *org.junit*;
- *org.neo4j.helpers.Function* → *org.neo4j.function.Function*. This migration occurred in NEO4J/NEO4J, in version 2.3.3. Type *Function* was moved to *org.neo4j.function*;
- *org.sonar.api.BatchComponent* → *org.sonar.api.BatchSide*. This migration happened in version 5.2 of SONARSOURCE/SONARQUBE. The new type improves some functions implemented by the deprecated one.

From the top-3,000 provided evolution rules, we found 720 where the left side corresponds to a type available in our original dataset of 622 Java systems; these are exactly the types investigated in this section. For example, the APIWAVE dataset includes the following rule: *org.neo4j.helpers.Function* → *org.neo4j.function.Function*, and our dataset includes the type *org.neo4j.helpers.Function*, which matches the left side of this rule.

Considering these 720 rules, there are 44 rules whose left side is a deprecated type. For example, the APIWAVE dataset includes the rule: *org.apache.commons.logging.Log* → *org.slf4j.Logger*, where *org.apache.commons.logging.Log* is a deprecated type. In fact, APIWAVE may produce rules not related to API deprecation. For example, the rule *java.util.List* → *java.util.Collection* is discarded because *List* is not in our dataset, and it is clearly not in the context of API deprecation.

Considering the 44 evolution rules where the left side is a deprecated type, we found that 32 types have a replacement message, while 12 types do not have such messages. For example, the right side of the evolution rule *org.hibernate.criterion.Expression* → *org.hibernate.criterion.Restrictions* corresponds exactly to the replacement message found in *org.hibernate.criterion.Expression*, as shown in Listing 10 (line 3).

5.2.2. Research questions

In this study, we assess whether a recommendation tool can be designed and implemented. Thus, we investigate two research questions:

RQA. What is the tool precision to recommend replacement messages?

We define precision as *TP/(TP+FP)*, where TP (True Positive) happens when a recommendation provided by APIWAVE matches the replacement message in a deprecated type; and FP (False Positive) happens when a recommendation provided by APIWAVE does not match the replacement message in a deprecated type.

RQB. What is the tool recall to recommend replacement messages?

We also calculate *recall* = *TP/(TP+FN)*, where FN (False Negative) happens when a replacement message in a deprecated type is not covered by APIWAVE data. In this case, we restrict our analysis to three popular systems: SONARSOURCE/SONARQUBE, JUNIT-TEAM/JUNIT, and GOOGLE/GUAVA. This is performed because we need to know all relevant elements (*i.e., TP+FN*) to compute recall, which requires manual assessment of all deprecated elements of a given system.

```
1 /**
2  * Factory for Criterion objects.  Deprecated!
3  * @deprecated Use {@link Restrictions} instead
4  */
5 @Deprecated
6 public final class Expression extends Restrictions { ... }
```

Listing 10. Example of replacement message (line 3) that matches an evolution rule inferred by APIWAVE - HIBERNATE/HIBERNATE-ORM.

```
1 /**
2  * A set of assert methods.  Messages are only displayed when an assert fails.
3  *
4  * @deprecated Please use {@link org.junit.Assert} instead.
5  */
6 @Deprecated
7 public class Assert { ... }
```

Listing 11. True positive example - JUNIT-TEAM/JUNIT.

```
1 /**
2  * @deprecated Use {@link android.support.v7.app.AppCompatActivity} instead.
3  */
4 @Deprecated
5 public class ActionBarActivity extends AppCompatActivity { ... }
```

Listing 12. False positive example - ANDROID/PLATFORM_FRAMEWORKS_SUPPORT.

```
1 /**
2  * @since 2.2
3  * @deprecated since 5.2 use {@link BatchSide} annotation
4  */
5 @Deprecated
6 public interface BatchComponent { ... }
```

Listing 13. Example of true positive in SONARSOURCE/SONARQUBE.

5.3. Results

RQA. What is the tool precision to recommend replacement messages?

We compute a precision of 73%: 32 out of 44 recommendations are true positives. As an example of true positive, we have the rule *junit.framework.Assert* → *org.junit.Assert*. The replacement message for the type in the left side matches the type in the right side of the rule, as shown in Listing 11 (line 4).

As an example of false positive, we present the rule *android.support.v7.app.ActionBarActivity* → *android.app.Activity*. The real replacement message for this type does not match the one suggested by this rule, as shown in Listing 12 (line 2). As noted, developers who deprecated this type are suggesting the usage of the alternative type *AppCompatActivity*. This shows that client applications may sometimes adopt other solutions than the ones pointed by replacement messages.

RQB. What is the tool recall to recommend replacement messages?

The analysis on SONARSOURCE/SONARQUBE reveals recall of 28.2%: we found recommendations for 13 out of 46 replacement messages. As an example of true positive, we show the *org.sonar.api.BatchComponent* case. The rule *org.sonar.api.BatchComponent* → *org.sonar.api.BatchSide* is the most popular one for the *org.sonar.api.BatchComponent* type. The replacement message for this type matches this rule, as presented in Listing 13 (line 3).

For JUNIT-TEAM/JUNIT, the recall is 30.7%: we detect recommendations for 4 out of 13 replacement messages. The rule *org.junit.matchers.JUnitMatchers* → *org.hamcrest.junit.JUnitMatchers* is the most popular one for the type *org.junit.matchers.JUnitMatchers*, and indeed represents an example of true positive. The replacement message for this type matches this rule, as shown in Listing 14 (line 2).

Finally, for GOOGLE/GUAVA, the recall is 37.5%: a recommendation tool based on APIWAVE rules would provide 3 out of 8 replacement messages. As an example of true positive, we have the rule *com.google.common.base.Objects.ToStringHelper* → *com.google.common.base.MoreObjects.ToStringHelper*. This rule matches the replacement message found in the source code, as presented in Listing 15 (line 3).

5.4. Final remarks

The presented study shows promising results, with good precision (73%) and reasonable recall (28%, 30.7%, and 37.5%), suggesting that a recommendation tool for elements deprecated without messages can help developers on finding alternatives API elements. Notice, however, that the low recall values are justified by the heuristic and dataset used by APIWAVE. More specifically, when comparing two versions of a class, APIWAVE extracts evolution rules from cases where only one type is removed and only one type is added in the import statements. The positive side of this approach is that it is more likely to obtain rules with higher precision. However, this comes at the cost of producing fewer rules. Moreover, APIWAVE mines a limited number of client projects (top-1000 most popular GitHub projects), thus, naturally, it may miss some evolution rules. Future studies may adopt different heuristics and increase the dataset to improve precision and recall.

6. Related work

In a large-scale study, Robbes et al. (2012) investigate the impact of API deprecation in a Smalltalk ecosystem. Later, they extended this study to the Java programming language (Sawant et al., 2016; 2017). The authors detected that some API deprecation have large impact on the ecosystem and that the quality of deprecation messages should be improved. The authors show evidences that APIs are sometimes deprecated with missing and unclear messages, however, their focus is on impact analysis, so they do not deeply investigate deprecation messages themselves. Hora et al. (2015b) studied the impact of API replacement and improvement

```
/**
 * @deprecated use {@code org.hamcrest.junit.JUnitMatchers}
 */
@Deprecated
public class JUnitMatchers { ... }
```

Listing 14. Example of true positive in JUNIT-TEAM/JUNIT.

```
/**
 * @deprecated Use {@link MoreObjects.ToStringHelper} instead.
 */
@Deprecated
public static final class ToStringHelper { ... }
```

Listing 15. Example of true positive in GOOGLE/GUAVA.

(*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.

McDonnell et al. (2013) 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 et al. (2014) analyze how API changes trigger questions and activity on StackOverflow. Results suggest that Android developers normally have more questions when the API behavior is modified. Recently, Bogart et al. (2016) studied how developers reason about and apply changes in the context of three software ecosystems: Eclipse, R/CRAN, and Node.js/npm. The authors state differences in practices, polices, and tools applied when performing/avoiding a breaking change. They present that breaking changes are rare in Eclipse; R/CRAN values consistency; and breaking changes in Node.js/npm are necessary for progress and innovation.

Several approaches are proposed to support API evolution and reduce the efforts of client developers. Henkel and Diwan (2005) propose CatchUp, a tool that uses a modified IDE to capture and replay refactorings related to API evolution. Chow and Notkin (1996) 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 and Valente (2015) propose apiwave, a tool to support keeping track of API evolution and popularity.

Kim et al. (2007) help to automatically infer rules from structural changes, computed from modifications at or above the level of method signatures. Kim and Notkin (2009) 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 (Kim et al., 2007). Nguyen et al. (2010) propose LibSync, a tool that uses graph-based techniques to help developers migrate from one framework version to another. Dig and Johnson Dig and Johnson (2005) 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 (2008) present SemDiff, a tool that suggests replacements for API elements based on how it adapts to its own changes. Schäfer et al. (2008) propose to mine API usage change rules from client systems. Wu et al. (2010) present AURA, an approach that combines call dependency and text similarity analyses to produce evolution rules. Meng et al. (2012) 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. Other studies focus on the extraction of API evolution rules that only make sense for a system or domain under analysis (Hora et al., 2012; 2015a).

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.

7. Conclusion

This paper presented an empirical study about the adoption of replacement messages on deprecated API elements. We focused on three major questions: (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 correctly deprecating API elements. The study was performed in the context of 622 Java and 229 C# popular and real-world systems. We reiterate the most interesting findings:

- 66.7% of the API elements in Java are deprecated with replacement messages per system (on the median). For C#, this value is 77.8%.
- Overall, the percentage of deprecated API elements with replacement messages does not improve over time.
- Systems that deprecate API elements in a correct way tend to be smaller and they have proportionally more contributors.

To investigate the practical application of our findings, we evaluated the feasibility of a recommendation tool designed to infer replacement messages by mining solutions adopted by developers. This preliminary investigation showed promising results, with good precision (73%) and reasonable recall for three real-world systems (28%, 30.7%, and 37.5%). This suggests that a recommendation tool for elements deprecated without messages can help developers on finding alternative API elements.

As future work, we plan to extend our analysis to other programming languages. For example, we can compare systems implemented in statically and dynamically typed languages. We also plan to categorize the systems under analysis regarding other characteristics (*e.g.,*domain, corporate backed) to reveal and understand differences regarding API deprecation. Moreover, in addition to looking at each of system characteristics in isolation, one can build a regression model on the percentage of deprecation with replacement messages with each characteristic as an independent variable. Finally, as previously discussed, further work includes the design and implementation of a recommendation tool to assist client developers by automatically inferring missing replacement messages in deprecated types, methods, and fields.

Acknowledgment

This research is supported by CNPq and FAPEMIG.

References

Bogart, C., Kästner, C., Herbsleb, J., Thung, F., 2016. How to break an API: cost negotiation and community values in three software ecosystems. In: International Symposium on Foundations of Software Engineering (FSE), pp. 109–120.

Brito, G., Hora, A., Valente, M.T., Robbes, R., 2016. Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 360–369.
Chow, K., Notkin, D., 1996. Semi-automatic update of applications in response to library changes. In: International Conference on Software Maintenance (ICSM), pp. 359–368.
Couto, C., Pires, P., Valente, M.T., Bigonha, R., Anquetil, N., 2014. Predicting software defects with causality tests. J. Syst. Software 93, 24–41.
Dagenais, B., Robillard, M.P., 2008. Recommending adaptive changes for framework evolution. In: International Conference on Software Engineering (ICSE), pp. 481–490.
Dig, D., Johnson, R., 2005. The role of refactorings in API evolution. In: International Conference on Software Maintenance (ICSM), pp. 389–398.
Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N., 2013. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In: International Conference on Software Engineering (ICSE), pp. 422–431.
Grissom, R., Kim, J., 2005. Effect Sizes for Research: A Broad Practical Approach. Lawrence Erlbaum Associates Publishers.
Henkel, J., Diwan, A., 2005. Catchup!: capturing and replaying refactorings to support API evolution. In: International Conference on Software Engineering (ICSE), pp. 274–283.
Hora, A., Anquetil, N., Ducasse, S., Allier, S., 2012. Domain specific warnings: are they any better? In: International Conference on Software Maintenance (ICSM), pp. 441–450.
Hora, A., Anquetil, N., Etien, A., Ducasse, S., Valente, M.T., 2015. Automatic detection of system-specific conventions unknown to developers. J. Syst. Software 109, 192–204.
Hora, A., Robbes, R., Anquetil, N., Etien, A., Ducasse, S., Valente, M.T., 2015. How do developers react to API evolution? The Pharo ecosystem case. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 251–260.
Hora, A., Robbes, R., Valente, M.T., Anquetil, N., Etien, A., Ducasse, S., 2017. How do developers react to API evolution? A large-scale empirical study. Software Qual. J. 1–31.
Hora, A., Valente, M.T., 2015. apiwave: keeping track of API popularity and migration. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 321–323.
Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D., Damian, D., 2014. The promises and perils of mining GitHub. In: Working Conference on Mining Software Repositories (MSR), pp. 92–101.
Kim, M., Notkin, D., 2009. Discovering and representing systematic code changes. In: International Conference on Software Engineering (ICSE), pp. 309–319.
Kim, M., Notkin, D., Grossman, D., 2007. Automatic inference of structural changes for matching across program versions. In: International Conference on Software Engineering (ICSE), pp. 333–343.
Konstantopoulos, D., Marien, J., Pinkerton, M., Braude, E., 2009. Best principles in the design of shared software. In: International Computer Software and Applications Conference (COMPSAC), pp. 287–292.
Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., Poshyvanyk, D., 2013. API change and fault proneness: a threat to the success of android apps. In: International Symposium on Foundations of Software Engineering (FSE), pp. 477–487.
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: International Conference on Program Comprehension (ICPC), pp. 83–94.
McDonnell, T., Ray, B., Kim, M., 2013. An empirical study of API stability and adoption in the android ecosystem. In: International Conference on Software Maintenance (ICSM), pp. 70–79.
Meng, N., Kim, M., McKinley, K.S., 2013. Lase: locating and applying systematic edits by learning from examples. In: International Conference on Software Engineering (ICSE), pp. 502–511.
Meng, S., Wang, X., Zhang, L., Mei, H., 2012. A history-based matching approach to identification of framework evolution. In: International Conference on Software Engineering (ICSE), pp. 353–363.
Montandon, J.E., Borges, H., Felix, D., Valente, M.T., 2013. Documenting APIs with examples: lessons learned with the apiminer platform. In: Working Conference on Reverse Engineering (WCRE). Citeseer, pp. 401–408.
Moser, S., Nierstrasz, O., 1996. The effect of object-oriented frameworks on developer productivity. Computer 29 (9), 45–51.
Nguyen, H.A., Nguyen, T.T., Wilson Jr., G., Nguyen, A.T., Kim, M., Nguyen, T.N., 2010. A graph-based approach to API usage adaptation. In: International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 302–321.
Robbes, R., Lungu, M., Röthlisberger, D., 2012. How do developers react to API deprecation? The case of a smalltalk ecosystem. In: International Symposium on the Foundations of Software Engineering (FSE), pp. 1–11.
Sawant, A.A., Robbes, R., Bacchelli, A., 2016. On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 400–410.
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.
Schäfer, T., Jonas, J., Mezini, M., 2008. Mining framework usage changes from instantiation code. In: International Conference on Software Engineering (ICSE), pp. 471–480.
Tian, Y., Nagappan, M., Lo, D., Hassan, A.E., 2015. What are the characteristics of high-rated apps? A case study on free android applications. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 301–310.
Tourwé, T., Mens, T., 2003. Automated support for framework-based software. In: International Conference on Software Maintenance (ICSME), pp. 148–157.
Wu, W., Gueheneuc, Y.G., Antoniol, G., Kim, M., 2010. Aura: a hybrid approach to identify framework evolution. In: International Conference on Software Engineering (ICSE), pp. 325–334.
Xavier, L., Hora, A., Valente, M.T., 2017. Historical and impact analysis of API breaking changes: a large scale study. In: 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 138–147.

Gleison Brito received his MSc degree in Computer Science from the Federal University of Minas Gerais (2016). His main research interests include software repository mining, and empirical software engineering.

André Hora is an Assistant Professor in the Computer Science Faculty at Federal University of Mato Grosso do Sul (UFMS), Brazil. Previously, he was a Post-doctoral fellow in the ASERG Group, UFMG, Brazil. He received a PhD degree from the University of Lille-1/Inria, France, in 2014. His main research interests include software maintenance and evolution, software repository mining, and empirical software engineering.

Marco Tulio Valente received his PhD degree in Computer Science from the Federal University of Minas Gerais, Brazil (2002), where he is an Associate Professor in the Computer Science Department, since 2010. His research interests include software architecture and modularity, software maintenance and evolution, and software quality analysis. He is a "Researcher I-D" of the Brazilian National Research Council (CNPq). He also holds a "Researcher from Minas Gerais State" scholarship, from FAPEMI G. Valente authored more than 60 refereed papers in international conferences and journals. Currently, he heads the Applied Software Engineering Research Group (ASERG), at DCC/UFMG.

Romain Robbes is an Associate Professor at Free University of Bozen-Bolzano, Italy. 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 and Mining Software Repositories. He authored more than 60 papers on these topics at top software engineering venues (ICSE, FSE, ASE, EMSE, ECOOP, OOPSLA), received best paper awards at WCRE 2009 and MSR 2011, and was the recipient of a Microsoft SEIF award 2011.