Empirical Software Engineering
https://doi.org/10.1007/s10664-019-09713-w

To react, or not to react: Patterns of reaction to API deprecation

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

Published online: 28 May 2019

Abstract
Application Programming Interfaces (API) provide reusable functionality to aid developers in the development process. The features provided by these APIs might change over time as the API evolves. To allow API consumers to peacefully transition from older obsolete features to new features, API producers make use of the deprecation mechanism that allows them to indicate to the consumer that a feature should no longer be used. The Java language designers noticed that no one was taking these deprecation warnings seriously and continued using outdated features. Due to this, they decided to change the implementation of this feature in Java 9. We question as to what extent this issue exists and whether the Java language designers have a case. We start by identifying the various ways in which an API consumer can react to deprecation. Following this we benchmark the frequency of the reaction patterns by creating a dataset consisting of data mined from 50 API consumers totalling 297,254 GitHub based projects and 1,322,612,567 type-checked method invocations. We see that predominantly consumers do not react to deprecation and we try to explain this behavior by surveying API consumers and by analyzing if the API's deprecation policy has an impact on the consumers' decision to react.

Keywords Deprecation · Application programming interface · API usage · Java

Communicated by: Alexander Serebrenik

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

Romain Robbes
rrobbes@unibz.it

Alberto Bacchelli
bacchelli@ifi.uzh.ch

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

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

[3] Departments of Informatics University of Zurich, Zurich, Switzerland

1 Introduction

> The most radical possible solution for constructing software is not to construct it at all.
>
> Frederick P. Brooks Jr., 1975

Application Programming Interfaces (APIs) are as close to a "silver bullet" as we have found in Software Engineering (Brooks 1975); Brooks acknowledges as much while revisiting his seminal essay "The Mythical Man-Month" after two decades (Brooks 1995).

However, just as like other software systems, APIs have to evolve and this evolution can have a large impact on its consumers if not done carefully (Robbes et al. 2012; Bogart et al. 2016a).

To smoothen the evolution of their API, producers can rely on the mechanism of *deprecation*. Whenever an API element is found to be inadequate, this element can be marked as deprecated to signal the consumers that its use is discouraged. A message can be added to the deprecation, for example, to suggest a replacement, to encourage consumers to migrate their code to the new version of the API, and to provide a rationale for the change. Software development tools support the deprecation mechanism: Compilers emit warnings when deprecated code is used and IDEs (*e.g.*, Eclipse 2018) visualize the usages of deprecated methods struck through. The API evolution is completed when, after a suitable period of time, the deprecated API element is removed from the API. At this point, any consumer that still uses the deprecated element would be unable to compile their code against the latest version of the API without first removing the calls to this element.

Several studies have shown that both consumers and producers may not behave as expected when it comes to the deprecation mechanism. The reaction of the consumers may be overdue or not happen (Robbes et al. 2012; Sawant et al. 2016, 2018); also, the API producer may not provide clear instructions for replacement or even fail to provide a rationale for the deprecation (Brito et al. 2016b, 2018; Sawant et al. 2018a). Producers may eschew from removing deprecated methods from the API to retain backward compatibility or, oppositely, remove API elements without first deprecating them (Sawant et al. 2018b). They may do so between major versions, or, breaking semantic versioning practices, do it between minor versions of the APIs (Raemaekers et al. 2014). Certain deprecation policies adopted by producers might have an adverse impact on the consumers (Sawant et al. 2018).

The current implementation of the deprecation mechanism in Java 8 has been changed for Java 9 (Marks 2017). The Java language designers who made the call to change the mechanism cite a lack of credibility surrounding the deprecation mechanism as the driver behind the change. According to the Java language designers, API consumers are unaware of whether a deprecated feature they use is going to be ever removed. All this has led API consumers to not taking deprecation seriously, thereby continuing with their use of the deprecated entities.

In this study, we seek to ascertain the scale of reactions or non-reactions to deprecated entities and the diversity of these reaction patterns. There is no current understanding as to how an API consumer can react to a deprecated feature or what the frequency of these reactions and the rationale behind them is. An in-depth understanding of whether consumers react to deprecation would allow us to understand whether consumers take deprecation seriously or whether they allow technical debt to accrue over time by not reacting. Concurrently, we would be able to assess if Java's deprecation mechanism is achieving its stated goal. Additionally, knowing the different kinds of reactions and their frequency allows API producers to understand whether their effort with evolving the API is worthwhile.

- We first conduct a qualitative study (presented in Section 4.1) to analyze the *diversity* (Jansen 2010) of how consumers react to API deprecation. We manually track a sample of 380 deprecated API elements in consumers' code across their lifetime and we observe the following patterns (beyond the expected pattern of replacing with the recommended replacement): non-reaction, deletion, replacement by another API, replacement with an in-house solution, and rollback to a previous API version.
- We then gather quantitative information (Section 4.2) about the *frequencies* of the reaction patterns we previously observed, by means of mining software repositories. Specifically, we quantify the reaction to API deprecation of 50 popular Java APIs, with a process that analyzed 297,254 Java projects on Github. The prevalent finding is that the most common reaction, which constitutes the 88% of the consumers' reactions, is to *not react*.
- We analyze if and how the reaction patterns vary depending on the considered API (Section 4.3). This also allows us to analyze if certain deprecation strategies are associated with specific reaction patterns (Section 4.4). We find that 20 of the APIs affect no consumers with deprecation, a further 18 APIs deprecate elements that they know have limited impacts on the consumers, and APIs that release rarely have fewer reactions than ones that release often.
- Since most consumers do not react to deprecation, we report on a survey of the reasons for non-reaction to deprecation (Section 4.4.2). We analyze 79 responses, and find that the top three reasons reported by respondents are: (i) the lack of a suitable alternative, (ii) the too high cost of reacting, and (iii) no perceived incentive to react since the API does not release frequently.

We conclude discussing the implications of our findings (Section 5), in particular, that deprecation seems to be viewed not seriously by consumers, who rarely react to it. This is in line with the view of the Java language designers.

2 Background: Deprecation in Java

Deprecation as a language feature exists to give API producers a way in which they can indicate that a feature should no longer be used. According to the official Java documentation: "A deprecated class or method is . . . no longer important. It is so unimportant, in fact, that you should no longer use it, since it has been superseded and may cease to exist in the future" (Rose 2017).

The principal idea of having a deprecation mechanism is to allow API consumers to take their time in adapting to API changes (Rose 2017). As the API evolves, some features might be replaced by newer features that are better, faster or more secure. However, simply removing the obsolete functionality would break API consumer code and allow them no transition period. During this transition period, the consumer is given ample indication in the IDE that the feature being used is deprecated, as seen on lines 3 and 4 of Listing 1.

Java first introduced deprecation in Java 1.1 as a `@deprecated` Javadoc annotation. This allowed API producers to indicate in the documentation that a feature is deprecated, give a reason behind the deprecation, and possibly indicate an alternative feature to use as seen in Fig. 1. Additionally, some APIs even provided a recommended course of action to deal with the deprecated feature. Subsequently, with the release of Java 5, annotations were added to the Java language, including a source code annotation `@Deprecated`. When a feature is marked with this annotation, the Java compiler throws a warning (Listing 2, 3, 4, 5, 6 and 7).

```
1   public class DateCalculator {
2     public static void main(String[] args) {
3   −       Date date = new Date();
4   −       int day = date.getDay()
5   +       Calendar calendar = Calendar.getInstance();
6   +       int day = calendar.get(Calendar.DAY_OF_WEEK);
7           System.out.println("Today is the " + day + "th of the week.");
8     }
9   }
```

Listing 1 Example of deprecated usage and reaction to it

The Java documentation states that deprecation allows API producers to keep obsolete functionality around for a certain period of time to preserve "backward compatibility" (Rose 2017). Once this period is passed, the obsolete feature can be removed as it would be likely that API consumers already transitioned away from using this obsolete feature. Hence, it is recommended that API consumers react to a deprecated API feature, unless they want to encounter a breaking change later in the evolution of the API. The consumer can react in a number of ways, one such example can be seen on lines 5 and 6 of Listing 1.

However, according to the Java JDK developers, API consumers do not appear to be taking deprecation seriously (Marks 2017). By not removing deprecated features from an API after a transition period has passed, API producers and the Java JDK developers themselves have cheapened the meaning of deprecation. This behavior has prompted few consumers to react to a deprecated feature. Java would like to change this in the upcoming release of Java 9 by enhancing the deprecation mechanism with information about future removal of a deprecated feature.

3 Methodology

In this section, we present the research questions and the research method.

getDay

```
@Deprecated
public int getDay()
```

Deprecated. *As of JDK version 1.1, replaced by* `Calendar.get(Calendar.DAY_OF_WEEK)`.

Returns the day of the week represented by this date. The returned value (`0` = Sunday, `1` = Monday, `2` = Tuesday, `3` = Wednesday, `4` = Thursday, `5` = Friday, `6` = Saturday) represents the day of the week that contains or begins with the instant in time represented by this `Date` object, as interpreted in the local time zone.

Returns:

`the day of the week represented by this date.`

See Also:

`Calendar`

Fig. 1 API documentation for deprecated entity

```
- if (JdkVersion.isAtLeastJava15()) {
      editor = descriptor.createPropertyEditor(this.targetObject);
- } else {
-       Class editorClass = descriptor.getPropertyEditorClass();
-       if (editorClass != null) {
-             editor = (PropertyEditor) BeanUtils.instantiateClass(editorClass);
-       }
- }
```

Listing 2 Deletion of deprecated usage

```
 ImmutableList<String> list = ImmutableList.of("hello");
- list.add("world");
+ ImmutableList<String> list2 = new ImmutableList.Builder<String>()
+                                                 .addAll(list)
+                                                 .add("world")
+                                                 .build();
+ list = list2;
```

Listing 3 Replace with recommended replacement

```
- ImmutableList<String>list = ImmutableList.of("hello");
- list.add("world");
+ MyImmutableList<String> list = MyImmutableList.of("hello");
+ list.add("world");
```

Listing 4 Replace with in-house replacement

```
- ImmutableList<String>list = ImmutableList.of("hello");
- list.add("world");
+ List<String> list = new ArrayList<String>();
+ list.add("hello");
+ Collections.unmodifiableList(list);
+ List<String> list2 = new ArrayList<String>();
+ list2.addAll(list);
+ list2.add("world");
+ Collections.unmodifiableList(list2);
```

Listing 5 Replace with Java replacement

```
1  − ImmutableList<String>list = ImmutableList.of("hello");
2  − list.add("world");
3  + List<String> stringList = new ArrayList<String>();
4  + stringList.add("hello");
5  + UnmodifiableList<String> list = new UnmodifiableList<String>(stringList);
6  + list.add("world");
```

Listing 6 Replace with other third-party API

3.1 Research Questions

The overall goal of this work is to understand the nature of reaction to a deprecated API artifact. This involves understanding how developers react to deprecation, observe the most popular way to react to deprecation, and how API policies are associated with reaction patterns. We structure our work along the following research questions:

RQ$_1$: How do API consumers react to deprecation? We only know that replacing the deprecated feature with its recommended replacement is something that the Java documentation on deprecation recommends. However, there is currently no empirical knowledge on the *diversity* (Jansen 2010) of how consumers react to API deprecation. To that end, we question as to what the possible reaction patterns are.

RQ$_2$: How do API consumers deal with deprecation? Based on the observed reaction patterns, we seek to uncover their frequency in an open source setting. This helps us understand as to how on a large scale API consumers prefer to react to deprecated features. To gain this understanding, we ask two sub-questions. The first attempts to establish the overall upgrade behavior of the consumers with respect to their dependencies and the second benchmarks the frequency of each reaction pattern.

RQ$_{2a}$: How often do consumers upgrade their dependencies?
RQ$_{2a}$: How often does each reaction pattern occur?

RQ$_3$: How do reaction patterns vary across APIs? Once we know the frequency of the observed reaction patterns, we seek to uncover if there is any dominant pattern for the consumers of any of the analyzed APIs. If the majority of the consumers react to deprecation in just one way for an API, we may hypothesize that the behavior of the API

```
1  <dependency>
2          <groupId>com.google.guava</groupId>
3          <artifactId>guava</artifactId>
4  −       <version>14.0</version>
5  +       <version>13.0</version>
6  </dependency>
```

Listing 7 Rollback version of the API

producer may influence this. Furthermore, an insight into the distribution of the reaction pattern for an API can help this API's producer understand how its consumers react to deprecations.

RQ$_4$: What are the reasons behind API consumers not reacting to deprecation? Finally, the results to our previous research questions showed that not reacting is the most common reaction pattern across all consumers of all APIs. With this research question, we would like to investigate this lack of reactions to deprecation. By not reacting, consumers are theoretically allowing technical debt to accrue over time; we would like to uncover the reasons behind this.

3.2 Subject Selection

To understand how API consumers react to the deprecation of features in APIs, we select a set of 50 popular Java APIs and their consumers to study. The popularity of an API is defined by the number of Maven-based Java projects on GitHub that use that API. We restrict ourselves to the Maven ecosystem among Java projects on GitHub because: (1) projects that use Maven can be considered to adhere to the most basic of software engineering principles and (2) Maven-based projects explicitly declare their dependencies in a project object model (POM) file that allows us to establish the API being used and the exact version in use.

We download the POM files of all Maven-based Java projects on GitHub. To ensure that all POM files are unique, we do not include forks of projects in our dataset, relying on the aid of GHTorrent (Gousios and Spinellis 2012). This results in a total of 135,729 POM files. Subsequently, we parse each one of these POM files to determine the list of APIs being used in the project. With this data, we classify the most popular Java-based APIs among GitHub clients; for example, JUnit is the most popular API, with 67,954 client Java projects.

We select the top 50 APIs—from different vendors—ranked by popularity in GitHub. Concerning the vendors, we see, for example, that the APIs `spring-core`, `spring-context`, and `spring-test` are all in the top 10 in terms of popularity and that they are all released by the same vendor (*i.e.*, org.springframework). By analyzing clients that use APIs from the same vendor, it is harder to isolate factors stemming from API policies on deprecation when it comes to reaction patterns to deprecation. Hence, we consider at most one API from each vendor.[1]

This selection process results in 50 APIs (a complete list can be found in Appendix B) where the most popular API (JUnit) is used by 67,954 Java projects and the least popular API (jetty-server) is used by 1,362 projects. By targeting these 50 APIs, the total number of API consumers that we analyze in this is 297,254.

3.3 API Usage Data Collection

To understand what features of an API consumers use, one can select from different proposed approaches that collect API usage data, *e.g.*, MAPO by Xie and Pei (2006) and SOURCERER by Bajracharya et al. (2006). We lean on the technique FINE-GRAPE developed by Sawant and Bacchelli (2017). This technique gives us three advantages: (1) it uses Maven-based Java projects, (2) it results in a type-checked API usage dataset, (2) it determines the API usages over the entire history of a given project.

[1] We have more API producers from Apache because Apache is an ecosystem and not a vendor in the traditional sense.

fine-GRAPE only focuses on projects that are under active development[2] *i.e.*, those that have been actively committed to in the last 6 months. We download all 297,254 active projects for the 50 APIs under study and then run the FINE-GRAPE analyzer on the source code of each project. This results in a dataset which contains 1,322,612,567 type-checked API usages across the entire history of the selected API consumers. The usage data we have collected spans from 1997 to 2017. The overall size of the dataset on disk in uncompressed form is 604GB and in compressed zip form is 473GB. It can be found at https://doi.org/10.4121/uuid:cb751e3e-3034-44a1-b0c1-b23128927dd8.

3.4 Determining the Reaction Patterns

There is no empirical knowledge on how API consumers react to deprecated features in an API they use. For example, a consumer might react to deprecation by replacing the deprecated feature with the recommended replacement or by rolling back the version of the API being used so that the feature is no longer deprecated.

Our aim is to create a taxonomy of possible reactions to deprecation. For this purpose, we perform a manual analysis of how an API consumer behaves when a deprecated usage is encountered. We select a sample of 380 usages of deprecated features and manually analyze these in depth. A sample size of 380 ensures a 95% confidence interval and 5% margin of error.

For each usage of a deprecated feature from our sample, we isolate the commit in which the method was originally marked as deprecated, and the consumers' file that uses it. We then look at all commits to the file from the point to see what happens to that usage. To see what changes in the entire project, we isolate the git diffs for each commit.

We analyze each usage and how it evolves over time. We try to decipher the reason behind the introduction of the deprecated usage and the nature and purpose of the API feature being used. Then we look at the documentation of the API to understand the API producers' recommendation (if any) as a reaction to the deprecated feature. We look at the entire history of the file to see what happens to that deprecated usage. If there is a change to it, we note down the nature of the change (a reaction pattern), if there is no change till the end of history we mark it as a non-reaction. The result of this analysis is an empirical understanding of what the API consumer does and the reasons behind the change, which we distill into a taxonomy of reactions to deprecated methods in APIs.

3.5 Quantifying the Reaction Patterns

Once we have an understanding of the various types of reaction patterns that API consumers can adopt, we seek to quantify these patterns. For this, we look at the clients of all 50 APIs and see how those that are affected by deprecation react to deprecation by looking for the reaction patterns found during the manual analysis.

For each API client, we have the method invocations for each file and information on how these invocations evolve over time in each file. This allows us to automatically infer what happens to a deprecated invocation over time. In Section 4.2, we detail the method we apply to automatically recognize and count each reaction pattern.

[2] As indicated by the GHTorrent dataset (Gousios and Spinellis 2012)

3.6 Associating API Evolution to Reactions

We want to see whether and how the evolution policies of APIs are associated with the way in which clients react to deprecation. An API might have a policy to deprecate very few features, thus impacting very few clients; or an API might remove deprecated features very often and this may persuade clients to react to a deprecation just to keep up with the APIs evolution.

We use four dimensions to benchmark the APIs along:

1. Actively releasing: This determines if the API has released a version of the API in the recent past and if it has a history of releasing frequently. APIs that change regularly are more likely to affect a consumer with deprecation as opposed to those that rarely or never release a new version due to the high volatility of features.
2. Deprecated feature removal: This benchmarks if the API has a tendency to remove a deprecated feature or not. APIs that frequently remove deprecated features are more likely to force consumers to react to deprecation due to the risk of a new version introducing breaking changes.
3. Percentage deprecated: This indicates the percentage of the API that has been deprecated on average over each version of the API. We take the average as opposed to the median as we believe that it provides a balanced figure over the entire lifetime of the API. Furthermore, we expect the number of deprecated features in the major version to remain constant in the minor versions of the API (Raemaekers et al. 2014). When a larger proportion of an API is deprecated there is a higher chance that consumers are affected by deprecation as opposed to APIs that deprecate few to none of the features.
4. Breaking changes: This indicates the number of breaking changes the API introduces without first deprecating the feature being removed. If an API has a propensity to introduce breaking changes as opposed to first deprecating a feature and then removing it, fewer consumers are likely to be affected by deprecation as the API does not follow the deprecation protocol.

For each of the dimensions, we define thresholds such that each API can be placed in one bin among the thresholds. Then, we hold a card sort session (Spencer and Warfel 2004) where we cluster APIs with similar evolution traits. The first two authors of the article perform the card sort.

3.7 Understanding Developer Perceptions Regarding Deprecation

Our goal is to gain an understanding as to why we observed certain reaction phenomena. To address this goal we designed a survey made up of 6 questions to send to developers. The questions asked in the survey are based on the observations made during the empirical investigation. We ask developers to rate the frequency (on a five-point Likert scale) with which they have reacted to deprecation in one of the ways identified, and to explain the rationale behind adopting this reaction behavior.

We aimed to reach as many Java developers who work on both industrial projects and open source projects. To achieve this goal, we spread the survey on Java developer forums (*e.g.*, Java code ranch), Reddit communities and Twitter. The survey was in the field for a period of 6 months. We obtained 79 responses and a further 88 developers started the survey but did not see through to completion.

28% of developers in our survey work on open source projects, the rest work on industrial/proprietary projects. Our respondents are primarily developers, with 4 respondents who

also work on research. All our respondents are experienced, with the average number of years of experience being 12. The origin of our respondents is not limited to one geographical location, we have responses from Europe, North America, South America, Australia, and Asia.

4 Results

4.1 RQ1: Reaction Patterns to Deprecation

We manually analyze 380 usages of deprecated API artifacts across consumers of 50 APIs. Based on this analysis, we observed seven reaction patterns (RPs) to deprecation. We describe these patterns, following the same order in which we discovered them in the manual analysis:

RP1: **No reaction** – API consumers do not do anything with the reference to the deprecated feature in their code base. The reference remains in the source code till the latest available version of the consumer code.

RP2: **Delete invocation** – API consumers react by removing the invocation to the deprecated feature, without replacing it with the replacement recommended by the API producers or any other functionality.

RP3: **Replace with recommended replacement** – API consumers replace the deprecated API element with the alternative proposed by the API producers.

RP4: **Replace with in-house replacement** – API consumers remove the invocation made to a deprecated feature and replace it with a functionality that they themselves create.

RP5: **Replace with Java replacement** – API consumers replace the deprecated invocation with an equivalent functionality provided by the Java Development Kit (JDK).

RP6: **Replace with other third-party API** – API consumers choose to switch API and replace the deprecated invocation with a non-deprecated one from the API to which they switch.

RP7: **Rollback version of the API** – API consumers rollback the version being used such that the used feature is no longer marked as deprecated.

> RQ_1. The manual inspection of 380 usages of deprecated API artifacts across consumers of 50 APIs lead to the discovery of six reaction patterns, in addition to 'Replace with recommended replacement'.

4.2 RQ2: Dealing with the Deprecation of a Feature

After having identified the possible reaction patterns (RPs), we investigate their occurrence among all the clients.

4.2.1 RQ2a: Version Upgrade Behavior

We begin our investigation by looking into how many of the API consumers in our dataset have changed the version of the library that they use.

We compute the percentage of consumers that upgrade the version of the dependency in our dataset. Overall in our dataset we see that not many consumers upgrade the version of

the API. None of the APIs has more than 13% of its consumers upgrade their dependency version. For APIs such as 'standard' and 'dom4j' the percentage of consumers upgrading version is less than 5%. For the widely popular APIs such as 'slf4j-api' and 'junit', only 12% or less of the consumers upgrade.

To triangulate this unexpected finding with another source, we asked to our survey respondents (see Section 3.7) whether they upgrade the version of the API that they use. Among the respondents, 31% of the API consumers indicated that they always upgrade the version of the API being used, while a majority of 69% indicate that they only do this occasionally or never. We asked this 69% to rank (on a five point likert scale) the frequency with which one or more motivations behind not upgrading the version of the API has applied to them. These motivations are a result from previous work, literature on deprecation and documentation on deprecation. The results of this can be seen in Fig. 2.

The upgrade cost, in terms of time or money, is the most common reason (42% of consumer rating it as 'almost always' to 'always') for not having upgraded. A 41% of the consumers reported not having upgraded ('almost always' to 'always') when everything in the current version of the API worked just fine. This is in line with what Sawant et al. (2018b) found. Breaking changes in the new version only stopped 32% of consumers from upgrading; in fact, 48% of the API consumers are neutral about this. Conversely, deprecation is seen as an even smaller barrier to upgrading, with only 9.7% consumers indicating that it has stopped them from upgrading the version of the API. A 22% of the responding consumers indicate that they have a policy to freeze the version of the API that they use (52% of consumers actually indicate that they have no such policy).

API consumers also provided us with additional reasons to not upgrade. One consumer indicated that management in the company that he worked in did not allow for dependency upgrades. Additionally, when a project reached a stable point it was no longer needed to upgrade the dependency.

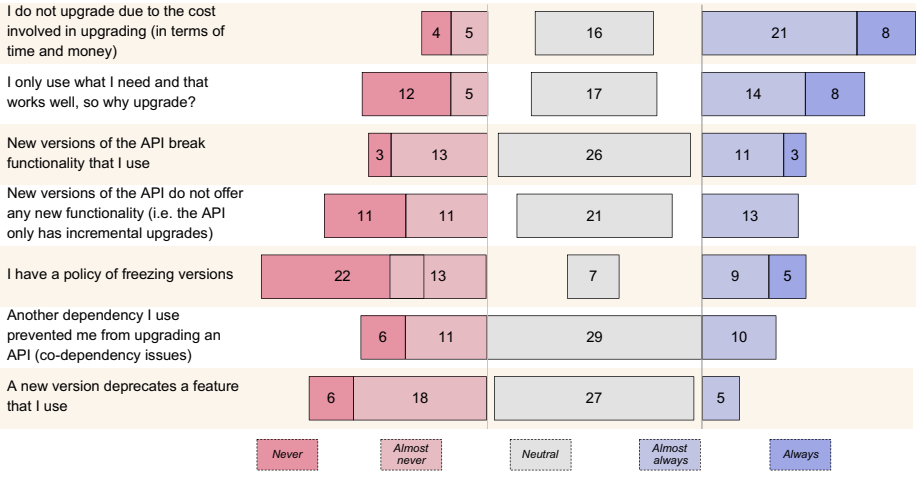


Fig. 2 Reasons behind not upgrading a dependency

Table 1 Breakdown of number of reactions per reaction pattern

Reaction pattern	Number of overall occurrences	number of unique consumers
No reaction	146,076	8,910
Delete invocation	1,015	218
Replace with recommended replacement	36	7
Replace with in-house replacement	702	31
Replace with Java replacement	17	3
Replace with other third-party API	15,236	641
Rollback version of the API	2,134	193

4.2.2 RQ2b: Frequency of Reaction Patterns

After having investigated the overall upgrading behavior of API consumers, we look into how API consumers react to deprecation by analyzing how frequently they adopt one or more of the reaction patterns that we found in RQ_1.

For each RP, we present: (1) the results gathered during the manual analysis conducted to answer RQ1 (in terms of both RP occurrence and the qualitative description of clients' behavior), (2) the dedicated heuristic we devised to automatically detect whether the RP takes place,[3] and (3) the number of overall occurrences and unique consumers of the RP across the 297, 254 API consumers pertaining to the 50 considered Java APIs.

We test the validity and accuracy of our heuristics by running them on the source code files of the 380 samples that were manually analyzed in Section 4.1. Our heuristics are able to identify the correct reaction pattern in 100% of the cases. Moreover, this analysis also confirmed the exhaustiveness of the patterns created in RQ_1: None of the analyzed cases let emerge new patterns.

We then analyze the fallibility of our heuristics to see whether they incorrectly classify a pattern (*i.e.*, establish the false positive rate). We manually analyze 100 cases of the automated classification for each of RP1, RP2, RP4, RP6 and RP7. For RP3 and RP5, we analyze all the cases since there is a limited number. For RP1–3, RP5, and RP7 we do not see any false positives. In the case of RP4 we see 7 instances where the replacement of deprecated feature with an in-house replacement did not make sense as the functionality being replaced was not the same. In the case of RP6 we observed 18 instances where the third-party API replacement does something completely different to the original API. However, looking deeper at these 18 cases we found that in 4 of these cases the developers had made a conscious choice to change functionality hence a one to one mapping was not needed. For the other 14 cases we could not precisely establish a rationale.

Overall, over 297, 254 API consumers, we see a total of 9, 317 projects that are affected by deprecation and react in one of the ways we have found. Over these 9, 317 projects, we see 165, 216 usages of deprecated methods to which reactions take place. The occurrence of each reaction pattern is summarized in Fig. 3 and Table 1.

[3]To automatically infer if a reaction pattern takes place, we start by going through the history of every file that uses a deprecated feature, every time we see that the number of deprecated features being used is decreasing, we attempt to see why the number of deprecated features being used has gone down, using the specific heuristic.

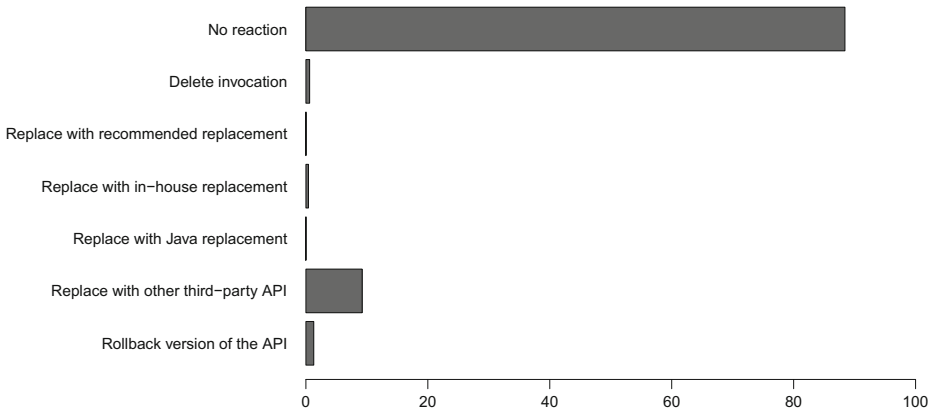


Fig. 3 Percentage distribution of reaction patterns

RP1: **No reaction**

Qualitative analysis:

Total occurrences: 290 (76%) *Unique consumers:* 221 (88%)

In the manually analyzed sample set, not reacting to a deprecated functionality is the most popular reaction pattern. On the one hand, this behavior may be explained by the fact that the cause of deprecation was not severe (we inspected the cause and there were no security or performance issues); on the other hand, this behavior is also unexpected, since all the deprecated entities that were not reacted to had recommended replacements and were well documented.

Detection heuristic: We look at the first version of a file that contains a deprecated feature and then the last version of the file, we see if the references to deprecated features have been removed in the last version. In the event that these have not been removed, we mark it as a 'non reaction'.

Quantitative analysis:

Total occurrences: 146,076 (88%) *Unique consumers:* 8,910 (95%)

Also in the large scale analysis, we found that the most frequent pattern is to not react to deprecation (88% of the time, for the majority of the projects–8,910 out of 9,317). This is despite the fact that the APIs that we consider are all popular and well documented mainstream Java APIs. Looking deeper at these non-reactions we notice that in 55% of the cases the files containing invocations to deprecated features not being reacted to do not change. This might be due to the fact that the file is already stable and requires no more changes. We cannot ascertain whether this code is being executed currently, however, given the active nature of the projects selected we do expect that the code is still being used in some manner. Due to this behavior, consumers simply might not notice that they are using a deprecated feature.

RP2: **Delete invocation**

Qualitative analysis:

Total occurrences: 23 (6%) *Unique consumers:* 4 (1.5%)

Deleting a deprecated invocation occurs frequently among the API consumers, in the manually analyzed dataset. We investigated the deprecated methods that have been removed: In two cases the deprecated feature was supposed to be removed,

since its usage was no longer needed; in the rest, the feature has a recommended replacement, however, the consumers delete the reference. In none of the cases do developers give any rationale behind the deletion.

Detection heuristic: When going through the different versions of a file containing a reference to a deprecated feature, if the number of references to deprecated features reduces and no new invocation is added in the same location in its place either from the same API or any third-party API, we mark it as a 'deleted invocation' with no replacement.

Quantitative analysis:
Total occurrences: 1,015 (0.6%) *Unique consumers:* 218 (2%)
Deleting and not replacing the invocation is also seen in over $1,000$ cases in the large scale analysis. Some of these deletions with no cause might stem from the fact that the API required the deprecated method to be handled in that manner (as we have seen in the qualitative analysis), however, it is reasonable to expect that this might not always be true (as we have also seen in the qualitative analysis).

RP3: **Replace with recommended replacement**
Qualitative analysis:
Total occurrences: 12 (3%) *Unique consumers:* 2 (0.8%)
Replacing a deprecated invocation with its recommended replacement is unpopular amongst API consumers. In our manually analyzed sample set, all the deprecated methods are documented and provide clear instructions as to how the deprecation should be handled. This makes it all the more surprising that we see very few reactions of this nature. Looking at the commit message for those API consumers that have actually replaced the deprecated invocation with the recommended replacement, they simply state that some changes were made due to the upgrade in the API version being used. In fact, in many cases, the reaction to deprecation was performed at the same time as the upgrade to the version of the library. Some API consumers react to deprecation immediately after noticing the deprecation.

Detection heuristic: For the API in question, we create a set of package names in the API over the entire history of the API, to verify whether an invocation added to a file belongs to one of these packages. In a version of a file in which a deprecated feature is removed, we check whether a new invocation is added to the same API in its place. If there is a new invocation made to the same API, we mark it as a 'replacement'.

Quantitative analysis:
Total occurrences: 36 (0.02%) *Unique consumers:* 7 (0.07%)
Replacing a feature with its recommended replacement is the second least frequent way in which API consumers reacts. In practice, only 7 API consumers choose to react in this manner. These 7 API consumers have replaced the deprecated feature in 36 (0.02%) cases and they do not react in any other manner to deprecation; furthermore, they replace all invocations being made to deprecated features. Thus, similarly to the qualitative analysis, we observe that consumers that do react to deprecation "as intended" tend to be systematic about it.

RP4: **Replace with in-house replacement**
Qualitative analysis:
Total occurrences: 2 (0.5%) *Unique consumers:* 1 (0.4%)

In two cases, the deprecated invocation is replaced by some functionality developed by the API consumer itself. Both cases belong to the Hibernate API. The API consumers have in each case replaced a database mapping invocation with their own wrapper around the database. The reason we found evidence for is that the Hibernate API was changing too much and it was not deemed worth keeping up with the changing API.

Detection heuristic: We start by looking at all the files and packages in a given project and create a set of package names that the project itself has defined, this is the list of packages from which a feature can be imported. When in a version of a file we see that a deprecated feature is removed, we look to see if it has been replaced by a feature pertaining to one of these in-house packages.

Quantitative analysis:
Total occurrences: 702 (0.4%) *Unique consumers:* 31 (0.3%)
Replacing with in-house functionality is done in 702 cases. Thus, in a non-negligible number of cases, API consumers have taken the effort to implement functionality that has been provided, yet deprecated by an external API.

RP5: **Replace with Java replacement**
Qualitative analysis:
Total occurrences: 6 (1.5%) *Unique consumers:* 1 (0.4%)
In our sample set, many of the APIs extend the functionalities of the Java API or provide alternatives to existing Java libraries. These are seen in the case of the Guava API's consumers. The consumer replaced references to the Guava Map class with those to the Java Map class. There was no reasoning given by the API consumer in the commit or in any pull request as to why the change was made. We speculate that using functionality from the Java API was deemed to be easier and safer than using deprecated features from the Guava API.

Detection heuristic: Java libraries start with one of three prefixes: `java`, `sun` and `javax`. If we see a deprecated feature is removed and—in its place—we see a new invocation being made to a method which belongs to a class whose package name starts with one of the prefixes, we infer that this reference to a deprecated feature has been replaced by a feature from Java and mark it as such.

Quantitative analysis:
Total occurrences: 17 (0.01%) *Unique consumers:* 3 (0.03%)
The least frequent way (17 cases or 0.01% of the time) for an API consumer to react is to replace the deprecated functionality with an invocation to a Java API feature. This could be explained by the fact that all third-party APIs seek to offer functionalities beyond the Java API building on top of it.

RP6: **Replace with other third-party API**
Qualitative analysis:
Total occurrences: 28 (7%) *Unique consumers:* 13 (5%)
More than one API may provide the same functionality. For example, Easymock and Mockito are both libraries that allow developers to mock objects in test cases. We see in 5 cases, all of which pertain to consumers of commons-collections API, deprecated features are replaced by functionality from the Guava API. The primary reason is that the commons-collections API had become obsolete, while Guava is more modern and provides more updated functionality.

Detection heuristic: There is no way to determine a list of package names for third-party replacement APIs as it is hard to understand which APIs can replace a certain API, thus making a complete list of replacement packages a non-trivial endeavor. Instead, we rely on data we have collected for all the aforementioned replacement based reaction patterns as for each pattern we create lists of packages to which a replacement method could belong to. When a replacement method does not belong to a class from any of those packages, we infer that it belongs to another third-party API.

Quantitative analysis:

Total occurrences: 15,236 (9%) *Unique consumers:* 641 (7%)
The second most frequent way (7%) in which API consumers (641 consumers out of 9,317) react to deprecation is by replacing a reference to a deprecated feature with a third-party API feature. This behavior might be prevalent for the consumers of some APIs as opposed to others as not all APIs have competitors.

RP7: **Rollback version of the API**
Qualitative analysis:
Total occurrences: 19 (5%) *Unique consumers:* 8 (3%)
Deprecations in an API are visible to API consumers when they upgrade the version of the API being used. At this point, a consumer can choose to stick with this version or rollback to the previous version. In one case (consumer of JUnit) in our sample set, we see this to be the case. The reasoning was that the new version had deprecated certain features being used. Needless to say, this defeats the purpose of deprecation.

Detection heuristic: When going through various versions of a file containing a reference to a deprecated feature ordered chronologically, if the file no longer contains deprecated features, we check whether the API consumer has rolled back the version of the API. We also ensure that the method invocation has not been removed from the source code. If both conditions are met, we mark it as a rollback.

Quantitative analysis:

Total occurrences: 2,134 (1%) *Unique consumers:* 193 (2%)
Rolling back the version of the API being used is also seen 1.2% of the time. This may indicate that several API consumers do not wish to take the effort to adapt to a new version of an API, but prefer to stick with an older version.

To challenge our aforementioned findings concerning frequency of reaction patterns, we ask in our survey (which targets API consumers from a different setting) which one of these reaction patterns they have most frequently adopted. Figure 4 reports the results.

69% of API consumers in our survey reported to always react by replacing the deprecated invocation with the recommended replacement from the API. This is in direct contradiction of the trends seen in GitHub data. While only 20% of consumers indicate that they do no react to a deprecated entity, in fact, 44% actually indicate that not reacting to deprecation is something that they would not consider as acceptable behavior. This again contradicts the behavior observed on GitHub. However, this could be explained by our survey respondents answer our questions to conform with what they perceive as acceptable behavior (known as social desirability bias Fisher 1993).

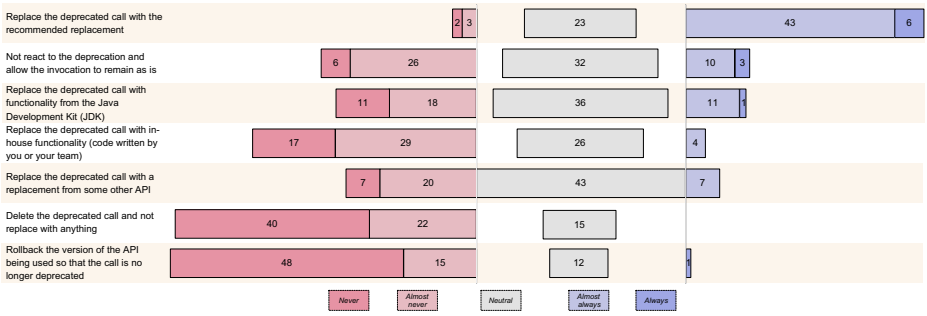


Fig. 4 API consumers' preferred way to react to deprecation

The other 5 reaction patterns all receive less than 17% of support from consumers. In all the cases, the majority of consumers indicate that they do not react in such a manner. We see that rolling back the version of the API and deleting a deprecated call with no replacement are by far the most negatively viewed reaction patterns with 80% and 75% of consumers indicating that they would never adopt such a pattern.

> ***RQ***$_2$**.** A small minority (less than 13%) of API consumers update their API version. Furthermore, the most common (88%) reaction to deprecation in our sample from GitHub is 'No reaction'; this result is not confirmed by the answers collected in our survey, whose respondent say to often (69%) 'Replace with recommended replacement'.

4.3 RQ3: Variance of Reaction Patterns Across APIs

By answering RQ2, we have analyzed the distribution of reaction patterns across all the consumer projects considered in our dataset, regardless of the API producer. Now we move our attention to investigating whether the distribution of reaction patterns varies across the consumers of different APIs. We differentiate between those APIs that affect several consumers vs. those that do not.

In Fig. 5 we see the percentage of consumers affected by deprecation. We define thresholds to create logical groupings of APIs based on what proportion of consumers are affected by deprecation. We define the thresholds for specific categorization by manually analyzing the graphs,[4] which is similar to the elbow method utilized when determining clusters (Bholowalia and Kumar 2014). We choose such a methodology over using quartiles as it allows us to take into account large changes in values and assign appropriate buckets to these values as opposed to lumping all values within a quartile in the same bucket.

Unaffecting consumers: 0% of consumers are affected by deprecations done by the API
Minimally affecting consumers: $< 2\%$ of consumers are affected by deprecations

[4] https://www.xaprb.com/blog/2015/11/07/setting-thresholds-with-quantiles/

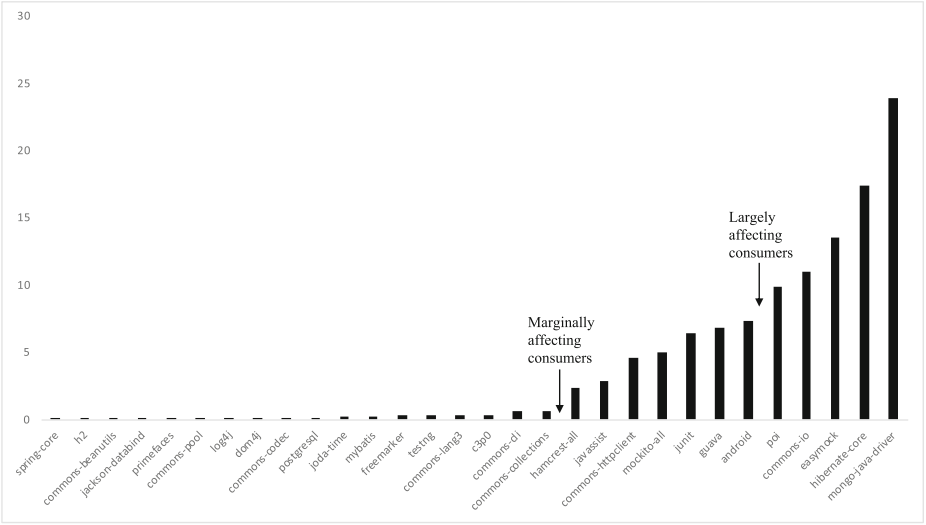


Fig. 5 Percentage of consumers affected per API

Marginally affecting consumers: between 2% and 7% of consumers are affected by deprecations

Largely affecting consumers: $>$ 7% of consumers are affected by deprecated features

Figure 6 shows a breakdown of the percentage of API producers belonging to each of these categories. Predominantly APIs do not affect their consumers, with 20 (40%) APIs never affecting any consumer and 18 (36%) APIs affecting less than 2%. Six (12%) APIs affect between 2% and 7% consumers and a further six (12%) affect more than 7% of the consumers.

Unaffecting consumers: For 20 APIs (40%), we observe that no consumer is affected by deprecation, because our sampled API consumers do not use any of the deprecated features. For the other 30 APIs, consumers are affected to various degrees.

Minimally affecting consumers: Considering the distribution of reaction patterns across the 18 APIs that minimally affect their consumers ($<$ 2%), consumers of 12 of these APIs predominantly do not react to deprecation. For other 3 APIs, replacing references

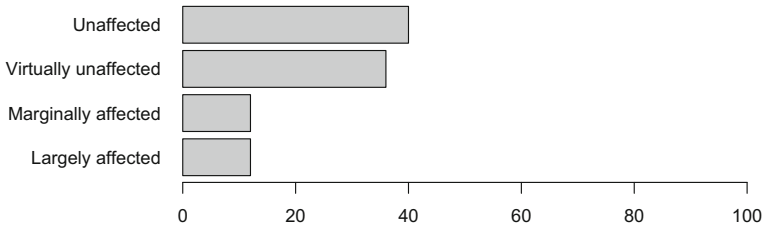


Fig. 6 Distribution of the sampled APIs based on degree to which their consumers are affected by deprecation

to deprecated entities with references to a third-party API (RP6) is the dominant reaction pattern. In the case of 'commons-lang3' (seen in Fig. 7), some consumers react by replacing deprecated features with features from the same API (RP3). Despite this fact, replacing with a third-party API is still the most common reaction pattern amongst consumers of this API. For the 'joda-time' API we see four different kinds of reaction patterns: no reaction (RP1), deletion (RP2), rollback and replacing with third-party API (RP6), as seen in Fig. 7. The consumers of the last API from this set, *i.e.*, 'commons-pool', always rollback the version of the API being used (RP7), as visible in Fig. 7. This seem to indicate that for the consumers of this API, deprecation acts as a deterrent to upgrading the version of the API being used.

Marginally affecting consumers: Among the APIs that marginally affect their consumers, the predominantly popular way to react to deprecation is by not reacting at all: For all the APIs, in over 60% of the cases consumers do not react to deprecation (RP1). There is a little bit of variance in terms of reaction patterns in the cases where a reaction does actually take place. In the case of 'guava', 'hamcrest-all', and 'junit', in 15-20% of the cases consumers have reacted by replacing a deprecated feature with a third-party API (RP6). In the case of the other 3 APIs that fall in category ('javassist', 'mockito', and 'commons-httpclient'), in over 25% of the cases features are replaced with third-party API ones (RP7).

Largely affecting consumers: Among those APIs that affect consumers the most, 'hibernate-core' has affected 17% (1,391 out of 7,983) of its consumers and 'mongo-java-driver' affects 24% (496 out of 2,077) of its consumers; deprecations in 'hibernate-core' and 'mongo-java-driver' are more exposed to the consumers than the deprecations in most other APIs. In the final set we see that for 'android', 'poi', 'commons-io', and 'easymock', consumers predominantly (∼80% of the cases) do not react (RP1). For these APIs, the alternative reaction patterns is typically to react by replacing the deprecated

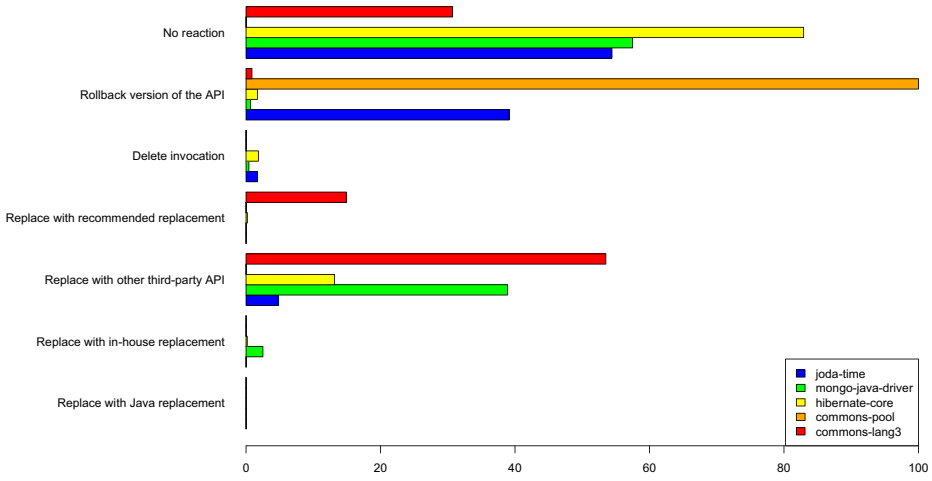


Fig. 7 Percentage breakdown of frequency of reaction patterns for the 'joda-time', 'mongo-java-driver', 'hibernate-core', 'commons-pool' and 'commons-lang3' APIs

features with a feature from a third-party API (RP7). Out of this set of APIs, only the consumers of 'hibernate-core' (in Fig. 7) show cases of replacing a deprecated feature with its recommended replacement (RP3). For this API the consumers appear to display the most varied reaction patterns. This may be explained by the large number of consumers (over 1,000) affected by deprecation. For 'mongo-java-driver' (the API that affects the largest percentage of consumers seen in Fig. 7), in over 40% of the cases the consumers prefer to start using a new API (RP6).

> RQ_3. Over 75% of the APIs marginally or don't affect consumers with deprecation. Only 2 APIs (hibernate-core and mongo-java-driver) affect more than 15% of their consumers. For these APIs we see reactions, where consumers abandon the API in favor of another.

4.4 RQ4: Explaining the Non-Reactions

We see that not reacting to deprecation is by far the most popular reaction pattern. This is an unexpected finding and we delve deeper into our data and survey developers to gain a thorough understanding behind this phenomenon.

We do this in two ways: (1) we analyze the impact of the API's evolution strategy in the consumers' reaction pattern and (2) we ask API consumers what motivates them to not react to deprecation.

4.4.1 Consumers' Reactions and API Deprecation Policies

In most cases, API consumers do not react to the deprecation of an API artifact; in some instances where a reaction does take place, the nature of these reactions can be diverse. We would like to investigate whether the API's deprecation and evolution strategies are associated with the consumer's behavior toward deprecation.

In Section 3.6 we list the four dimensions along which we measure the behavior concerning an API's evolution; similar to Section 4.3, for each of the dimensions we define the thresholds for specific categorization by manually analyzing the graphs. The thresholds are detailed in Table 2.

Table 2 Dimensions of API evolution and thresholds

Dimension A: Actively releasing			
$\not{A}$: last release >5 years ago	A^-: releases every >12 months	A: releases every 4-12 months	A^+: releases every 0-3 months
Dimension R: Removal of deprecated features (percentage)			
$\not{R}$: no deprecated methods removed	R: 0-50% of deprecated methods removed	R^+: 50-75% of deprecated methods removed	R^{++}: >75% of deprecated methods removed
Dimension D: Deprecated features (percentage)			
D^-: 0 - 2% of methods deprecated	D: 3-9% of methods deprecated	D^+: 10-15% of methods deprecated	D^{++}: >16% of methods deprecated
Dimension B: Breaking changes (cardinality)			
$\not{B}$: 0 breaking changes	B: 1-90 changes that break methods	B^+: >90 changes that break methods	

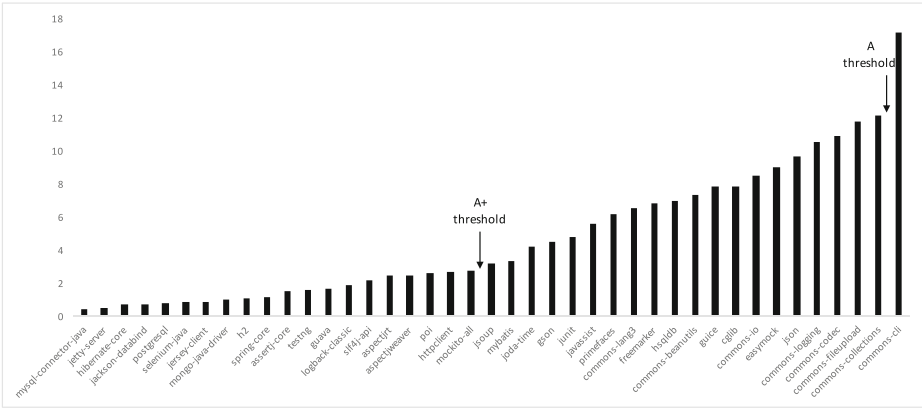


Fig. 8 Activeness of APIs, excluding inactive APIs

We start by looking at the release activeness of the APIs. We notice that some APIs have not been active for more than 5 years and denote these as inactive. For the rest, we define thresholds based on the number of releases per months. A breakdown of this metric per API along with the chosen threshold points can be seen in Fig. 8.

We look at the percentage of deprecated features that an API removes in its lifetime. For some APIs, no deprecated features are removed. For the rest, we define thresholds based on the graph seen in Fig. 9.

We also take into account the percentage of the API that is deprecated over its lifetime. For all our APIs there is at least one feature that has been deprecated. We capture the frequency with which features are deprecated in different bins, the thresholds for which can be seen in Fig. 10.

Finally, we look at the number of breaking changes that an API introduces. The thresholds defined for this can be seen in Fig. 11. We scored each of the APIs based on the categories defined in Table 2 and conducted a card sort (as described in Section 3.6). This card sort resulted in nine API deprecation strategies (DSs), each with its own defining characteristic. In the following, we provide more details about each emerged DS, in terms of its description along with the aforementioned dimensions, which APIs fit in the pattern and an analysis of the behavior of the consumers of the APIs adapting this DS.

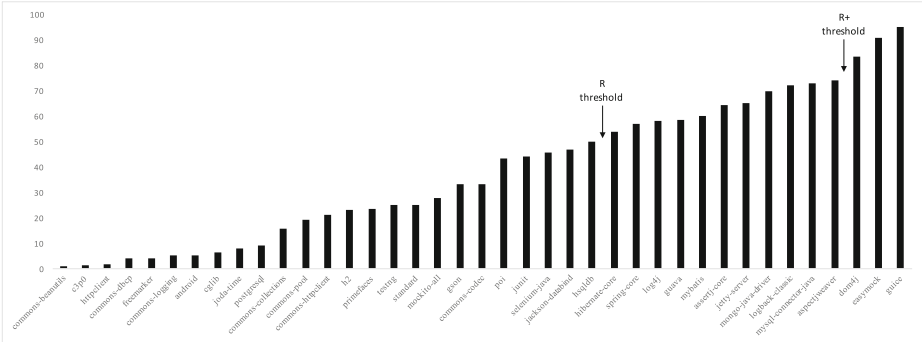


Fig. 9 Removal of deprecated features, excluding APIs that never remove a deprecated feature

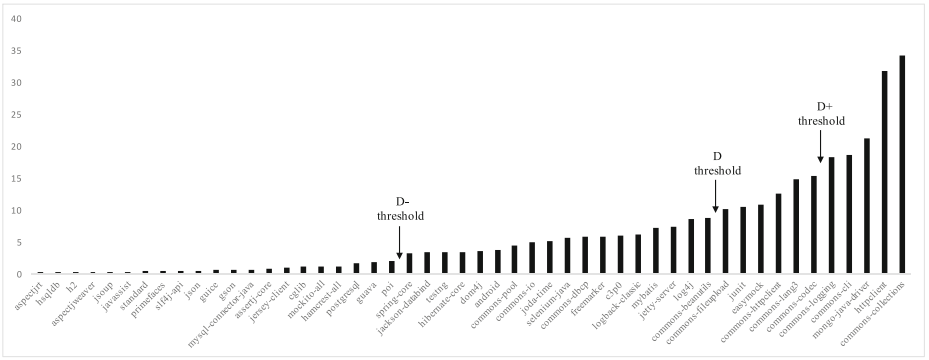


Fig. 10 Frequency of deprecating features, excluding APIs that do not deprecate at all

DS1: **Inactive**

Description: The API is not actively releasing new versions, thus deprecated features are not being removed.

APIs: log4j $(\not{A}, R+, D, B)$, commons-dbcp $(\not{A}, R-, D, B)$, hamcrest-all $(\not{A}, \not{R}, D-, B)$, standard $(\not{A}, R, D-, B)$, dom4j $(\not{A}, R++, D, B)$, c3p0 $(\not{A}, R-, D, B)$, commons-httpclient $(\not{A}, R, D+, B)$, android $(\not{A}, R-, D, B)$, commons-pool $(\not{A}, R, D, \not{B})$.

Expected reaction: APIs in this category are not at all active, *i.e.*, they have not released a new version in the last 5 years, hence there is no danger in using a deprecated feature as there is no immediate danger of it being removed in the new release of the API. Hence, here we expect no reactions to take place.

Analysis of consumers: This is the most common strategy with 9 out of 50 APIs belonging to this category. Two of the APIs in this set never affect their consumers, while for 5 APIs over 80% of the consumers never react, as expected. For

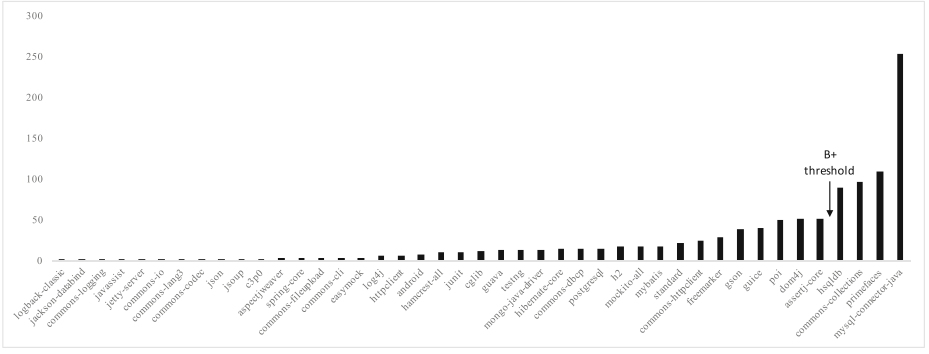


Fig. 11 Frequency of breaking features, excluding APIs that never directly introducing breaking changes

the last two APIs, some reactions do take place. Out of these APIs, 'commons-pool' is the only outlier where 100% consumers affected by deprecation react and do so by rolling back the version of the dependency being used.

DS2: **Very little deprecation**

Description: The API has no history of removing features, deprecated or otherwise.

APIs: slf4j-api $(A+, \not{R}, D-, \not{B})$, aspect-jrt $(A+, \not{R}, D-, \not{B})$, json $(A, \not{R}, D-, B)$, javassist $(A, \not{R}, D-, B)$, jsoup $(A+, \not{R}, D-, B)$, jersey-client $(A+, \not{R}, D-, \not{B})$.

Expected reaction: Here APIs deprecate features, however, they never remove deprecated features from their APIs. Thus they always remain backward compatible. In this case, we expect no reactions.

Analysis of consumers: Six APIs appear to be backward compatible in all cases, this implies that deprecated features are never removed. For 5 of these APIs, no clients are affected (this is expected, since a defining trait of this strategy is that rare deprecations take place). However, for 'Javassist', in over 35% of the cases that a reaction does take place by replacing a deprecated call with one to a third-party API (RP6). Which implies that consumers of this API are not aware that there is no danger in not reacting as no deprecated features are ever removed.

DS3: **Little deprecation**

Description: The API does not deprecate a lot, but when it does, it does not remove the deprecated features.

APIs: joda-time $(A, R-, D, \not{B})$, postgresql ($A+$, $R-$, $D-$, B), cglib (A, $R-$, $D-$, B), commons-beanutils $(A, R-, D, \not{B})$, freemarker (A, $R-$, D, B).

Expected reaction: These APIs remove very few deprecated features, thus most deprecated features remain in the API and there is very little danger that they will be removed later. Here we expect no reactions.

Analysis of consumers: Five APIs that fall under this category. These APIs are mostly backward compatible, thus consumers do not need to react much as there is very little inherent danger in a deprecated feature being removed. Consumers of two APIs do appear to exhibit this behavior (RP1). However, for 'joda-time', we see rollbacks (RP7, 40% of the cases) and for 'postgresql' we see migrations to another API (RP6, 60% of the cases).

DS4: **Never cleans up API**

Description: The API deprecates a lot but never really removes any deprecated feature.

APIs: commons-io $(A, \not{R}, D, B)$, commons-lang3 $(A, \not{R}, D+, B)$, httpclient ($A+$, $R-$, $D++$, B), commons-logging (A, $R-$, $D++$, B), commons-fileupload $(A, \not{R}, D+, B)$, commons-cli $(A-, \not{R}, D++, B)$.

Expected reaction: The APIs threaten a lot of breaking changes by deprecating a lot of features, but none of these deprecated features are removed. Thus, no reaction has to take place due to the lack of danger of using a deprecated feature. No reactions are expected.

Analysis of consumers: Six APIs fall in this category. For three of these APIs, no consumer is affected by deprecation. On the other hand for the consumers of the other 3 APIs, we do see quite some reactions. For commons-cli (>50% of the cases), commons-lang3 (>50% of the cases) and commons-io (20% of the cases) we see that deprecated calls are replaced by another third-party API (RP6).

DS5: **Rarely cleans up API**

Description: The API deprecates several features, yet only removes a few of these features.

APIs: junit (A, R, $D+$, B), commons-codec (A, R, $D+$, B), commons-collections (A, R, $D++$, $B+$).

Expected reaction: These APIs deprecate a lot of features, however, also remove a few of these deprecated features. Thus there is moderate danger in using a deprecated feature from this API. We expect a few reactions, but not too many.

Analysis of consumers: Three APIs belong to this category. Just like the previous category we expect to see reactions as a lot of deprecations take place but few removals. Consumers of two APIs out of this set do not react to deprecation in over 80% of the cases. However, consumers of commons-collections do follow the more expected pattern of reacting, by replacing deprecated invocations with those being made to other third-party APIs (RP6, >60% of cases).

DS6: **Directly breaks few methods**

Description: The API removes the features sometimes, but not frequently.

APIs: mockito-all ($A+$, R, $D-$, B), gson (A, R, $D-$, B), h2 ($A+$, R, $D-$, B), poi ($A+$, R, $D-$, B), primefaces (A, R, $D-$, $B+$).

Expected reaction: Not a lot of features are deprecated, instead, some breaking changes are introduced directly. There is quite some danger to using a (deprecated or otherwise) feature from an API in this set as the API can remove a feature at any time. We expect to see reactions.

Analysis of consumers: Five APIs exhibit this strategy. We see that in the case of 3 APIs over 98% of the consumers do not react, and one does not affect consumers at all. Only for mockito-all do consumers react by replacing deprecated calls with new ones to a third-party API (RP6, 25% of the cases).

DS7: **Directly breaks a lot of methods**

Description: The API removes deprecated and non-deprecated features frequently.

APIs: mysql-connector-java ($A+$, $R+$, $D-$, $B+$), guava ($A+$, $R+$, $D-$, B), aspectjweaver ($A+$, $R+$, $D-$, B), hsqldb (A, $R+$, $D-$, $B+$), guice (A, $R++$, $D-$, B), assertj-core ($A+$, $R+$, $D-$, B).

Expected reaction: Here APIs introduce breaking changes with regularity instead of first deprecating a feature. We expect to see many consumers being affected by deprecation.

Analysis of consumers: Surprisingly, this strategy is exhibited by 6 APIs. This implies that 6 APIs choose to break their client code as opposed to evolving in a clean manner and first deprecating a feature and only after that removing the

feature. Thus, these APIs are not too bothered by the thought of breaking their consumers' code. This is apparent, given that consumers of 5 out of 6 APIs are unaffected by deprecation at all. Only for guava do we see that consumers are affected and they react. These reactions are either by migrating to another API or rolling back the version of the API being used.

DS8: **Removes deprecated features**

Description: The deprecated features are sometimes removed in a future version.
APIs: jackson-databind ($A+$, R, D, B), testng ($A+$, R, D, B), selenium-java ($A+$, R, D, $\not{B}$).
Expected reaction: These APIs are quite active and they deprecate and remove deprecated features regularly. Given the danger of using these features, we expect reactions.
Analysis of consumers: 3 APIs exhibit this strategy. For testng, in over 60% of the cases, there is no reaction seen. Whereas for selenium-java no API consumers are affected by deprecation. However, only for jackson-databind do we see that consumers react in just under 40% of the cases by replacing with another API (RP6).

DS9: **Actively cleans up deprecated features**

Description: The API removes most deprecated features in future versions.
APIs: spring-core ($A+$, $R+$, D, B), hibernate-core ($A+$, $R+$, D, B), logback-classic ($A+$, $R+$, D, B), mybatis ($A+$, $R+$, D, B), mongo-java-driver ($A+$, $R+$, $D++$, B), easymock (A, $R++$, $D+$, B), jetty-server ($A+$, $R+$, D, B).
Expected reaction: Here APIs remove deprecated features with regularity, thus ensuring that consumers will be confronted with breaking changes when upgrading the version of an API. We expect to see reactions.
Analysis of consumers: This strategy is exhibited by 7 APIs. We see in the case of spring-core, hibernate-core, mybatis, and easymock that no reactions actually take place in over 80% of the cases. Only in the case of mongo-java-driver do we see that in 35% of the cases do reactions take place, where consumers replace deprecated invocations with those to another API (RP6). This counter-intuitive behavior of the API consumers can be explained by the fact that majority of the consumers do not upgrade the version of the API being used, thus minimizing the chance that they will be affected by a breaking change in the API.

4.4.2 API Consumer Perspective on Non-reaction

We asked API consumers to indicate whether one or more reasons for not reacting to deprecation has applied to them in the past. Results from this survey can be seen in Fig. 12.

The most common reason (53.4% of respondents) for not reacting is that the specified replacement by the API is either too complicated to use or is not a good enough replacement. This shows that API producers might not be making their replacement features developer friendly. Furthermore, it also calls for API producers to invest in making detailed upgrade guides or improving documentation. 49% of respondents also indicated that they found reacting to deprecation time-consuming and not worthwhile. This might be explained by the

fact that our survey respondents work in industry thus not having sufficient time to react to deprecation.

42% of respondents indicate that they do not react to a deprecated feature as they would rather use another API that provides similar functionality. This is in line with the earlier results where consumers found it hard to react to deprecation due to the convoluted and time-consuming nature of the replacement. Consumers also indicate that they are lax when it comes to reacting to deprecation as they do not feel particularly threatened by the deprecation as the immediate danger of a new release of the API that removes the deprecated feature being used is non-existent. This is reflected in the results seen in Section 4.4.

Approximately the same percentage of consumers indicate that the fact that deprecation is a non-breaking change has an impact on their decision to react (38%) as those that indicate that this fact has no impact on their decision to react (37%). Thus, consumers have diverging opinions over the effectiveness of the deprecation mechanism. However, the fact that the deprecated feature might never be removed thereby never becoming a breaking change, does not act as motivation for non-reaction either.

The reason behind deprecation or lack thereof does not have a major impact on the non-reaction pattern observed according to 47% of consumers. This indicates that consumers feel that the reason behind deprecation is a driving factor to react to deprecation, which is in line with previous results (Sawant et al. 2018b).

Reasons to not react such as the ability to rollback the version being used or developing an in-house alternative to the deprecated API receive very little support from API consumers. Most indicate that such reasons do not motivate non-reactions.

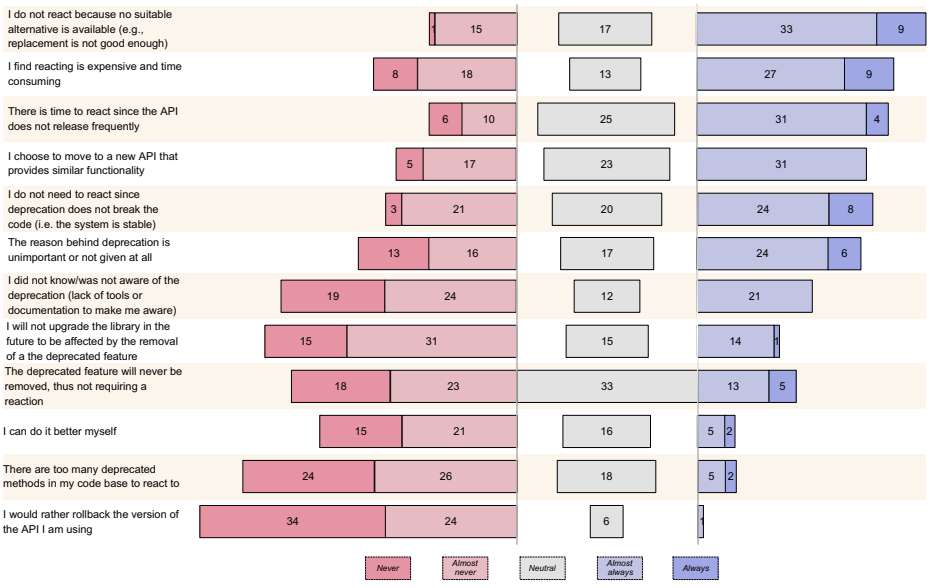


Fig. 12 API consumers reasons for not reacting to deprecation

Consumers also mentioned other reasons that have motivated them to not react to deprecation. One consumer mentioned that the fact that the feature had been deprecated for a long time and never been removed, made it apparent that no reaction was needed. In some cases in industry, the management does not want to invest time or money in upgrading a dependency and reacting to deprecated/breaking changes in the API. Another consumer indicated that upgrading the API binary sometimes leads to incompatibilities with other binaries thus preventing reactions to deprecation.

We wanted to further understand the reasons behind reacting to deprecation, to see what motivated consumers to react. Some of these reasons can be seen in Fig. 13.

We see that the reason behind deprecation, the low cost of reaction, the seriousness of the deprecation and the need to upgrade the library are all considered to be very important (over 60% of consumers in each case) motivations behind reacting to deprecation. This is in line with the responses that we obtained for non-reactions.

> $\boldsymbol{RQ_4}$**.** Deprecation strategies adopted by APIs are not strongly linked with the reaction patterns seen. Further diving into the non-reactions, we observe that consumers do not react to deprecation due to the cost and complexity of the required change.

5 Discussion

5.1 Deprecation Not Considered

The deprecation mechanism is used to indicate that a certain API feature is now obsolete and will be removed in the future. The aim of using such a dedicated mechanism is that it allows API consumers a period of time to react to the deprecation and take the course of action they deem most suitable for their use case.

In our study, we observe that consumers do not heed this deprecation warning. Over 95% of the cases in which a deprecated feature is used by a consumer, the depreaction is never reacted to. In the other 5% of the cases, the nature of the reaction varies and we have observed six different reaction patterns.

In 55% of the cases where no reaction has taken place, the file containing the deprecated invocation is never modified. This could indicate that API consumers may not notice that an API feature being used in their source code is deprecated. If confirmed, this situation would strengthen the case that simple compiler warnings for deprecation might not be sufficient in calling the consumers attention to this issue; this would call for another manner in which the visibility of a deprecated feature can be promoted. Java language developers have attempted to address this by introducing a dedicated tool called jdeprscan[5], which scans Java class files to detect the usage of deprecated features and warns the consumer.

[5] https://docs.oracle.com/javase/9/tools/jdeprscan.htm #JSWOR-GUID-2B7588B0-92DB-4A88-88D4-24D183660A62

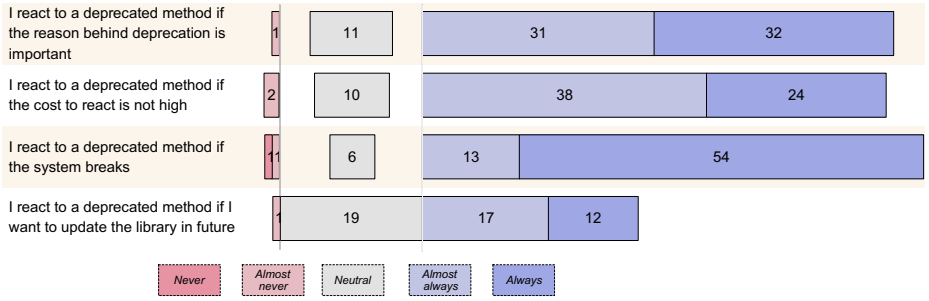


Fig. 13 API consumers reasons for reacting to deprecation

The scale of the non-reactions is surprising and shows us that the deprecation mechanism is not achieving its stated goal. In fact, consumers claim that the reason behind deprecation is what drives their decision to react. This is similar to what Sawant et al. (2018b) found in their qualitative study. They recommend that the deprecation mechanism implementation in Java should make it explicit as to what the reason behind the deprecation is. Seeing the scale of non-reactions, we can speculate that this enhancement in deprecation will entice more consumers to react to deprecation.

Overall, by increasing the overall awareness surrounding the deprecation of a feature, there is a chance that more consumers may react. This would require providing the consumer with more visibility of the deprecated feature and the motivation of the deprecation, in addition to sufficient documentation that acts as a guide to reaction. Further studies can be devised to verify whether and to what extent these three factors combined may reduce the frequency of non-reactions (from the 88% it is at right now).

5.2 Lack of Affectedness by Deprecation

Deprecation is how API producers can indicate obsolete functionality and warn consumers to not use certain features. The principal stated use case for deprecation is when an old/obsolete feature is replaced by a new implementation, but the API producers do not want to directly replace the existing feature as doing so would directly break consumer code, thus making deprecation a compromise solution.

We would expect for the APIs in our dataset that several features would have gone through the deprecation process and deprecation would have affected consumers significantly. In fact, the APIs under consideration are mature, popular, large, and have evolved significantly over time. However, we see that only 9,317 out of 297,254 (3%) Java projects on GitHub are affected by deprecation. This is surprising as we would have expected API evolution to have affected more consumers. Indeed, investigating the API deprecation behavior, we see that it is rare that an API deprecates large portions of the API. Furthermore, we see that for some popular APIs such as Guava, breaking changes are introduced in conjunction with deprecating a feature. Both these facts might explain the lack of affected consumers that we observe.

On the other hand, our results may also be the result of consumers rarely updating the APIs they use. We see that for most APIs only 5% of the consumers upgrade the dependency that they use with none of the APIs having more than 13% of their consumers that upgrade versions. This is in line with previous work by Sawant and Bacchelli (2017) who reported

that a small number of consumers upgrade versions. The rare upgrading may contribute to explain why few consumers are affected by deprecation.

5.3 API Producers' Policies are Not Associated to Consumers' Reactions

API producers use deprecation to communicate with their consumers about the obsolete nature of an API feature. They can use different evolution strategies when it comes to deprecating features: They might deprecate a lot of features, and never remove these deprecated features, or they could directly introduce breaking changes without first deprecating a method.

We observe that the various deprecation strategies adopted by producers are minimally associated with the consumers' decision to react to deprecation. Even when it would be imperative that a consumer reacted to deprecation due to the danger of that deprecated method being removed by the API, we observe consumers not reacting.

The Java language designers in JEP277 (Marks 2017) estimate that the variance in deprecation strategies and having no singular convention has led to the confusion surrounding deprecation. Consumers are unsure of whether they have to react or not. They are unaware of the future of a deprecated feature. In the opinion of the Java language designers, this confusion has led to consumers doing nothing with a deprecated feature.

5.4 Consumers do not Keep up with API Evolution

Most of the APIs that we study regularly release new versions, which contain a combination of improvements to existing features, addition of new features, and/or removal/deprecation of older features.

We observe that, for any of the studied APIs, only a maximum of 13% of consumers move to the latest version of the API. In the survey, only 31% of developers indicate that they always upgrade the version of the API being used. These numbers are aligned, albeit less extreme, with previous studies, one of which reported that 81.5% of consumers do not change the version (Kula et al. 2018) and another that less than 4% of library dependencies are upgraded (Teyton et al. 2014).

Considering the consumers that do upgrade, we observe that they tend to not react to deprecation. They choose not to use the new features that replace obsolete features. In fact, replacing with the recommended replacement is one of the least popular ways in which we found consumers reacting to deprecation.

This evidence seems to indicate that consumers are not concerned with keeping up with API evolution. Consumers pick the version that works best for them and stick to it (a fact echoed by 41% of developers in the survey).

This leads us to question whether APIs should be evolved with great regularity or if API producers should invest time in making minor improvements and releasing them. Concerning upgrading behavior, Bavota et al. (2015a) found that consumers are more likely to adopt a new version if it includes major improvements.

5.5 Need for an Automated Tool to Keep with API Evolution

One of the principal reasons behind consumers not keeping up with API evolution, as emerged from the survey, is the cost involved in upgrading dependencies and reacting to deprecated or broken features. In the case of industry, this cost is too high a price to pay to change something that is already working, as explicitly stated by one of our survey respondents.

This hints at the usefulness of tools that can enable developers to deal with API evolution in a cost-effective manner. Some attempts at automatically dealing with deprecation have already been made. Henkel and Diwan (2005) proposed to capture refactoring operations made by API producers to their codebase when adapting to their own deprecated features and then replaying these operations on the API consumers. Similarly, Xing and Stroulia (2007) developed an approach that recommends alternative features from an API to replace an obsolete feature by looking at how the API's own codebase has adapted to change. These approaches rely on support from the API developers to aid the API consumers in the transition.

Attempts have been made to create automated tools that aid developers in dealing with API evolution and not just specifically deprecation. Dagenais and Robillard present a tool called SemDiff (Dagenais and Robillard 2009), which aids developers in dealing with framework changes where a method they use is suddenly no longer provided. Schäfer et al. (Schäfer et al. 2008) mine framework usage rule changes from already ported usages of the framework and propose the same to a developer dealing with a breaking change. Kapur et al. (2010) created a tool call Trident that allowed developers to directly refactor obsolete API calls. Savga and Rudolf created Comeback! (Ṡavga and Rudolf 2007), a tool that records framework evolution and for each change creates an adapter so that a developer does not have to change his code.

The only tool that has been created specifically to support API consumers in transitioning away from deprecated API features is that created by Perkins (2005). This tool replaces deprecated method invocations with the source code of the deprecated method from the API itself. This has been shown to be effective in 75% of cases, although this approach is not universal and introduces verbose code to the API consumers codebase.

One assumption that all these aforementioned approaches make is that consumers upgrade the dependency version being used. As we see in this study, developers mostly do not change the version of the API they use. This calls for the development of techniques that actually incentivize the adoption of new versions by reducing the cost and time involved in upgrading. Holmes and Walker (2010) have made a start by creating a tool that filters relevant change information from a library so that consumers are made aware of the major changes made to the API. Furthermore, as our study has shown us, consumers are very likely to take deprecation lightly as it is not a breaking change and does not require immediate attention. This calls for a tool that effectively aids an API consumer in the transition away from deprecated API features. Both these tools are still an open research challenge.

While in this context we only talk about the needs of Java developers, such kind of tooling support in other languages would also go a long way in aiding API consumers in dealing with API evolution.

5.6 Comparison with Other Languages

In this paper, we focus on Java-based APIs and their consumers. We identify seven different ways in which API consumers could react to deprecated API features in their source code. However, we see that there is very little reaction to deprecated API features, in fact, not reacting is the most popular way in which consumers choose to deal with deprecated API features.

Java's deprecation mechanism and deprecation policy have been blamed as the main reason behind the lack of reaction to deprecated features. JEP 277 (Marks 2017) mentions that more information needs to be conveyed to the consumer, information that can prove pivotal in the decision-making process behind reacting to deprecation. Sawant et al. (2018b)

confirm this and suggest that Java's deprecation mechanism be extended to inform the consumer about the severity of the deprecation and the version in which the deprecated feature is to be removed.

Languages such as C#, Kotlin and Visual Basic provide a way in which the severity of the deprecation can be conveyed. Ruby and Dart allow API consumers to indicate the version in which a deprecated feature is going to be removed. We postulate that given the difference in the information that is conveyed by the API producer to the consumer in such languages, the variety and scale of the reaction to deprecation is probably very different to that what we have observed in Java. This hypothesis is strengthened when we observe the results from a previous study by Robbes et al. (2012) on the Smalltalk (whose deprecation mechanism is similar to Java's in terms of characteristics) ecosystem where the number of reactions is not that high either.

A study to compare reaction patterns to deprecation across the different programming languages would allow us to confirm whether the implementation of a deprecation mechanism has an impact on consumer behavior. Such a study is out of scope for this paper, however, we propose it as future work.

5.7 Semantic Versioning Impacting Deprecation Reaction Behavior

The practice of deprecation is related to the practice of semantic versioning. In semantic versioning, package version numbers follow a specific scheme consisting of 3 numbers: MAJOR.MINOR.PATCH. When releasing a new version, one of the numbers will be incremented according to the following rules (from https://semver.org):

- The MAJOR number is incremented when a new version makes incompatible API changes
- The MINOR number is incremented when new functionality is added in a backward-compatible manner
- The PATCH number is incremented when a backward-compatible bug fix is issued

On the other hand, the goal of deprecation is to provide an incentive for developers to change their software, but without breaking backward compatibility. Thus, an API deprecating even a large number of API elements does not need to increase its major number to comply with semantic versioning. The actual breaking change would happen when the deprecated API element is removed; only then would an API following semantic versioning need to increase its major version number.

Both mechanisms can be seen as complementary. One could see deprecation as an "advance warning" that an API element is likely to be removed in the future, such as a future major version of the API. Indeed some APIs, such as Guava, explicitly note in their deprecation message when a method is scheduled to be removed (*e.g.*, "This method is scheduled to be removed in Guava 16.0"). This strategy seems to be "the best of both worlds".

While such a strategy seems to be the best way to approach the problem, based on our study and the studies of Raemaekers et al. (2014, 2017) on semantic versioning, our outlook on the chances of such a mechanism leading to a desired behavior in practice (*i.e.*, rapid adaptation to deprecation and API changes) is pessimistic. The work of Raemaekers et al. shows that many packages on Maven central do not follow semantic versioning, incurring rework for their consumers. Our work shows that few API consumers actually react to deprecation. Thus, it is unclear whether combining both approaches would be more successful, although a specific study of this would be the best way to obtain concrete evidence.

6 Related Work

6.1 Studies on API Deprecation

Several studies investigate the deprecation of API features, its impact and its need to help developers deal with deprecation.

Robbes et al. (2012) and Hora et al. (2015) have studied the impact of deprecation of APIs in the Pharo ecosystem. Robbes et al. focused on the deprecation of certain API features and the effect that this deprecation has on the entire Pharo ecosystem. They found that the deprecation of a single feature can have a large impact on the ecosystem, despite this only a small proportion of consumers bother reacting to deprecation. Hora et al. looked at changes to the API changes not marked as deprecated beforehand. They find that a larger number of consumers react to API changes marked as deprecated as opposed to those not marked as deprecated, thus showing that in the Pharo ecosystem a larger proportion of consumers consider deprecation to be of importance.

Sawant et al. (2016) performed a large-scale study on GitHub based consumers of 5 popular Java APIs to see how they are affected by deprecation and whether or not they make a move to react to deprecated features. They found that only a small proportion of consumers update dependency versions. Furthermore, the proportion of consumers affected by deprecation varies per API, however, irrespective of the scale of affected consumers, the number of reactions is minimal. Sawant et al. extended this study to look at consumers of the Java JDK API (Sawant et al. 2018). They found that for the Java API consumers are rarely affected by deprecation, dispelling the notion put forth by the Java developers themselves. Furthermore, they theorized that an APIs deprecation policy would have an impact on the consumers' decision to react to deprecation, but did not show any conclusive evidence in either direction. In this study, we show that the deprecation practices have a minimal impact on the consumers' reaction patterns.

Brito et al. (2016a) analyzed the documentation accompanying deprecated features in 661 Java systems. They found that in 64% of the cases, API producers had taken the effort to recommend a replacement in the documentation. Brito et al.'s study does not look at whether these replacement messages also mention the rationale behind the deprecation, a facet of the documentation that the consumers find important as confirmed by our study.

Hou and Yao (2011) studied release notes of the JDK, AWT and Swing APIs, looking for rationales for the evolution of the APIs. They found that in the case of deprecated API features, several reasons were evoked: Conformance to API naming conventions, naming improvements (increasing precision, conciseness, fixing typos), simplification of the API, reduction of coupling, improving encapsulation, or replacement of functionality. Many of these rationales mirror those mentioned in the Java documentation on deprecation. They also found that only a small portion of API features were deleted without a replacement specified. Sawant et al. (2018b) asked API producers why they used the deprecation mechanism. They also asked producers if they preferred that consumers would react to deprecation and what they did to support any kind of reaction. Based on their findings, they propose some more changes that have to be made to the deprecation mechanism so that it fulfills the needs of both API producers and consumers.

6.2 Studies on API Evolution

API producers have to decide what part of an API they have to exclude from public access, these are the so-called internal APIs. These parts of the API are reserved for use by the API

itself and not intended for public consumption. Businge et al. (2013a) found that out of 512 Eclipse plugins, 44% use internal Eclipse APIs. This finding is confirmed by Hora et al. (2016) who found that 23.5% of 9,702 GitHub based Eclipse client projects use an internal API. Businge et al. (2013b) dived deeper into the reasons behind developers using internal APIs and they found that developers preferred to use an internal API as it provided them with functionality that other public APIs did not, thus sparing them time and development effort. Hora et al. (2016) found that internal APIs are sometimes promoted to public APIs. To aid API producers in the selection of what API should be promoted, Hora et al. presented an approach for such promotion. The consumers' propensity to use features that they are not supposed to use, is reflected in our study as well, where consumers do not want to transition away from deprecated features. This shows that maintainability takes a back seat to functionality, which leads to technical debt.

Raemaekers et al. investigated the relationship between breaking changes, deprecation, and the semantic versioning policy adopted by an API (Raemaekers et al. 2012). They analyzed a dataset based on 100,000 JAR files on Maven central. They found that API producers introduce deprecated artifacts and breaking changes in equal measure across both minor and major API versions, thus not allowing consumers to predict API stability from semantic versioning. In a follow-up to this study, Raemaekers et al. (2017) found that these breaking changes induce a lot of rework in consumers. Furthermore, the deprecation tags used by these Maven based projects are often used incorrectly.

APIs often introduce breaking changes that directly affect a consumer. Dig and Johnson studied and classified the API breaking changes in 4 APIs (Dig and Johnson 2006), however, they did not investigate the impact of these breaking changes on consumers. They found that 80% of breaking changes were due to refactorings performed by the API producers and released without deprecating the original implementation. Wu et al. (2016) analyzed the Eclipse ecosystem to see how an API change would affect an API consumer. They found that missing API classes affect consumers more frequently than breaking changes. They also find that 11% of API changes can cause a ripple effect among API consumers. Wu et al. (2014) propose a tool called ACUA which would give API producers an overview of the impact of the change that they would make to an API.

In a large-scale study of 317 APIs, Xavier et al. (2017) found that for the median library, 14.78% of API changes break compatibility with its previous versions and that the frequency of these changes increases over time. However, not many clients are impacted by these breaking changes (median of 2.54%). Bogart et al. (2016b) conducted interviews with API developers in 3 software ecosystems: Eclipse, npm, and R/CRAN. They found that each ecosystem had a different set of values that influenced their policies and their decisions of whether to break the API or not. In the case of R/CRAN both Decan et al. (2016) and Bogart et al. found that there is a policy of forcing packages to work with one another, which is perceived to be a problem. Decan et al. (2017) also investigated the evolution of the package dependencies in the npm, CRAN, and RubyGems ecosystems. They found that there is an increasing tendency in the npm and (to a lesser extent) RubyGems packages to specify maximal version constraints, thus allowing certain package maintainers to protect themselves from package updates.

Bavota et al. (2015a) qualitatively investigated a Java subset of the Apache ecosystem to see how their dependencies change over time. They observed that when an API adds a lot of new features, consumers are more likely to adopt this new version, thus triggering a change in the dependency. However, when the changes are small and insignificant or if a removal of a feature takes place, then consumers prefer not to change versions of the dependency.

The policy of evolving the Android APIs has been studied as well. McDonnell et al. (2013) investigate stability and adoption of the Android API on 10 systems. They found that the Android API's policy of evolving frequently leads to the consumers being adversely affected by breaking changes, thereby creating many issues when it comes to dealing with API evolution. Linares-Vásquez et al. (2014) also focus on the Android ecosystem, however, they look to analyze StackOverflow posts to see how consumers deal with API evolution. They found that there are more StackOverflow conversations when there is a change in the Android API, thus further showcasing issues with dealing with API evolution. Bavota et al. (2015b) analyzed how changes in APIs being used by apps on the Google Play Store affects app ratings. They show that when more breaking changes are introduced, the rating of the app is lowered.

Web-based API evolution policies have also been studied. Wang et al. (2014) study the case of evolution of 11 APIs by analyzing questions and answers on StackOverflow concerning the evolution of these APIs. study the specific case of the evolution of 11 REST APIs. They identify 21 change types to an API that affects consumers and spark a discussion on StackOverflow. Espinha et al. (2014) analyze how major web API providers evolve their APIs, and they find that there is no unified way in which web APIs are evolved thus leading to confusion among consumers. Espinha et al. (2015) also studied 43 mobile consumer applications depending on web APIs and how they respond to mutations in the web API. They show that in over 30% of the cases the mobile app fails when the web API response is changed.

6.3 Supporting API Evolution

Researchers have proposed many approaches to aid consumers in dealing with the evolution of an API. One of the first approaches was by Chow and Notkin (1996), where they require API producers to annotate changed methods with replacement rules that will be used to update consumer code. Henkel and Diwan (2005) propose CatchUp!, a tool that captures refactorings in the IDE and replays the, on other unrefactored code in the consumer's code base. Xing and Stroulia (2007) propose an approach that analyzes how an API has itself handled changes to its features and then recommends these changes to the API consumer. Dig et al. (2007) propose MolhadoRef a refactoring-aware version control system that works in a similar manner as CatchUp!.

Dagenais and Robillard present SemDiff (Dagenais and Robillard 2008, 2009) a tool that observes the framework's evolution to make API change recommendations. Schäfer et al. mine the consumer's reaction to API evolution (Schäfer et al. 2008), and then propose these mined changes to other consumers dealing with the same evolution issues. Wu et al. present a hybrid approach (Wu et al. 2010) that uses call dependency and textual similarity to recommend adaptations to the API changes.

Kapur et al. (2010) created a tool call Trident that allows consumers to directly refactor obsolete API calls in the IDE. Savga and Rudolf created Comeback! (Ṡavga and Rudolf 2007) which records framework changes and recommends ways to adapt to these changes to the consumer.

Nguyen et al. (2010) propose a tool called LibSync that uses graph- based techniques to help consumers migrate from one framework version to another. Holmes and Walker notify developers of essential changes made to external dependencies to draw their attention to these events (Holmes and Walker 2010).

Cossette and Walker (2012) studied five Java APIs to evaluate how API evolution recommenders would perform in the cases of API breaking changes. They found that all recommenders handle only a subset of the cases, but that none of them could handle all the cases.

Finally, Perkins (2005) created a tool that was specifically targeted to consumers dealing with deprecated features in their codebase. This tool replaces the invocation made to deprecated features with the code from the deprecated method itself, thereby aiding in the removal of the deprecation warning message on the code.

7 Threats to Validity

The heuristics that we use to automatically detect and measure the frequency of each of the reaction patterns might be flawed. These heuristics might result in incorrect identification of a reaction pattern or not detect a pattern at all. To mitigate this threat we test the heuristics on the manually analyzed dataset of reaction patterns to ensure that they accurately detect the reaction patterns.

This study focuses on Java based API consumer projects hosted on GitHub, which comes with its own set of threats as documented by Kalliamvakou et al. (2014). We try to mitigate these threats by focusing only on GitHub projects that are active and not forks of any other project. However, these projects are all open source projects which might be hobbyist projects, personal projects or educational projects. In all these cases regular maintenance of code might not be of the highest priority. The effect this might have on our data might be minimized as we focus primarily only on Maven based projects, which shows that these projects adhere to a minimum of software engineering practices.

During the mining of the API usages, we focus only on the master branch of a project. This might result us in missing out on reactions that might have taken place in a branch that has as yet not been merged with the master branch. However, we expect that these project adhere to the git norm of having the latest working copy of the code in the master branch (Chacon and Straub 2014).

In this study, we focus on the most popular Java APIs. We do this so that there is sufficient API usage data that can be mined for our results. This might bias our study towards the more main stream Java development, thus leading us to possibly missing out on reaction patterns that might occur for the less popular Java APIs. We strive for a large breadth of API consumers in this study, and overall have 297,254 GitHub based projects that use 50 popular APIs. Given such a large number of API consumers, we expect some of these projects to use less popular APIs as well. We would expect that the way in which these consumers react to deprecations in one API should be similar to how they react to deprecation in other APIs. Thus, we expect reaction patterns to not vary due to the popularity of the APIs.

Most of our survey respondents are from industry and their answers deviate from what is observed on GitHub. We hypothesize that this can be the result of three different causes. (1) it could be that industrial code adheres to other (higher) standards; on the other hand [(2)]survey respondents may have reported what would be *expected* when dealing with deprecation (this would be in the line of socially desirability bias Furnham 1986) or (3) by not randomizing the survey questions, the responses might have inadvertently been influenced. Further studies can be setup and conducted to investigate these hypotheses further.

8 Implications

The key implications that we distill from this study are:

- The majority of Java-based projects on GitHub do not show substantial reaction to deprecation, this is reflected by the fact that there are very few reactions to deprecated features and API consumers choose to keep references to deprecated features in their source code. This may suggest that a deprecation *per se* is not perceived as a good enough reason to change one's code. This result is in line with previous research (Sawant et al. 2018b), which provided evidence that consumers would like to have more information (*e.g.*, severity of the deprecation or version in which the feature will be removed) to take a more informed decision when it comes to reacting to deprecation.
- API consumers need to be made more aware of the API's evolution policy. We notice that in cases where no reaction need take place a reaction does take place and other cases where the need to react is more pressing (due to the APIs propensity to remove deprecated features) we observe no reaction. This seems to indicate that consumers do not know the exact evolution behavior of the API that they are using. This information can either be composed in a new online platform or on the Maven central website where the dependency is hosted, such that consumers have an early warning mechanism at their disposal.
- The scale of non-reactions may also indicate that the effort to manually make every single transition from a deprecated feature to a non-deprecated one may be not trivial. This puts into focus the increasing need for an automated tool that aids in API transition that would reduce developer burden on this front.

9 Conclusion

We have presented a large empirical scale study that analyzes how frequently an API consumer reacts to deprecation in an API. This is the first work of its kind that identifies the various possible reaction patterns that can take place. We identify seven reaction patterns by way of manual analysis of API consumer code. We then quantify the frequency of these reaction patterns by mining and analyzing API usages of 50 popular Java based and their consumers. The overall size of the dataset under consideration encompasses 297,254 projects and over 1.3 billion API usages.

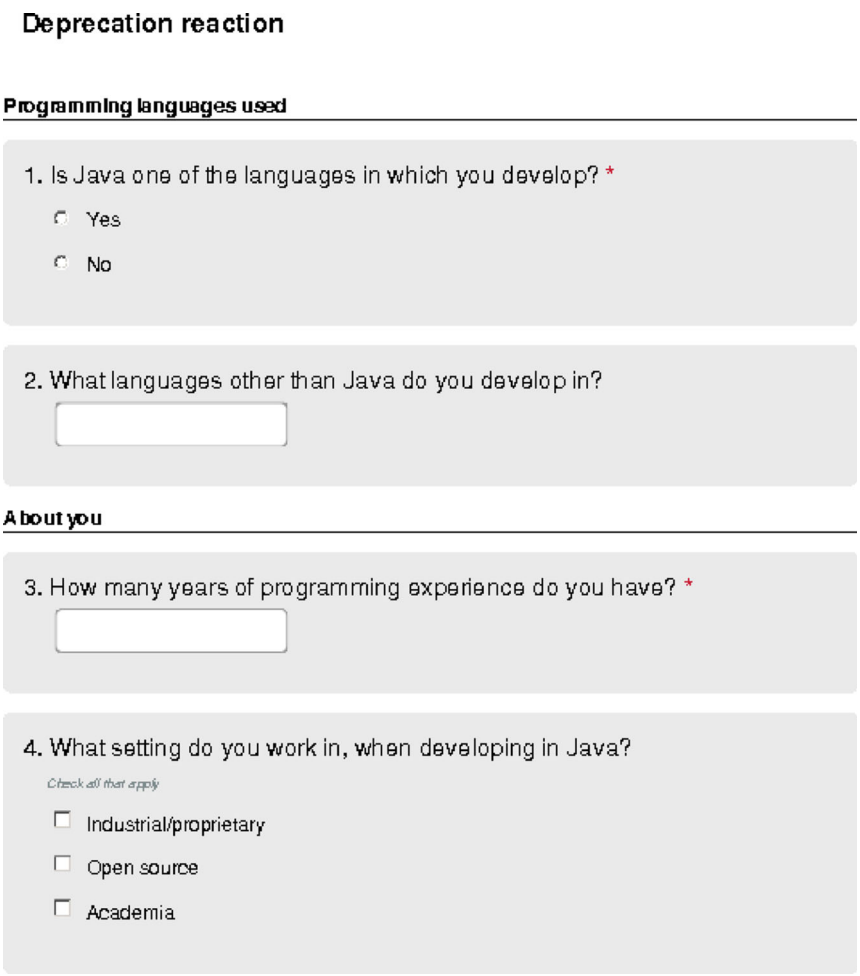
In our manual analysis we saw that the bulk of the consumers never react to deprecated features. This fact is reflected in our large scale analysis as well. Surprisingly, replacing with the recommended replacement only happens in 0.02% of the cases, while non reactions happen in 88% of the cases. This shows that API consumers are either unconcerned with the fact that their code uses deprecated API features or have not noticed this fact. Furthermore, when diving into they upgrade behavior of API consumers, we see that very few consumers even change the version of the API that they use.

We asked developers as to why they would not upgrade the version of the API that they used. And the two major reasons behind this is that the cost involved with the upgrade is often not worth it, and given that the version being used works, there is no pressing need to upgrade. Developers were also asked to rank the reasons behind not reacting to deprecation. Here the developers indicated that the replacement was either too convoluted to use or the cost of reacting was too high. It appears that all these consumers subscribe to the "if it isn't broken then why fix it?" theory.

One of the major contributing factors to this behavior is that deprecation is not viewed as seriously as it should be. A fact that the Java JDK developers accede to as well. Multiple improvements and changes are needed for consumers to take deprecation warning seriously.

However, with this study we are able to confirm that issues with the current implementation of deprecation do indeed exist.

Appendix A



Deprecation reaction

Programming languages used

1. Is Java one of the languages in which you develop? *
   - Yes
   - No

2. What languages other than Java do you develop in?

About you

3. How many years of programming experience do you have? *

4. What setting do you work in, when developing in Java?

   Check all that apply

   - Industrial/proprietary
   - Open source
   - Academia

5. Which of the following best describes your primary work area?

- ☐ Development
- ☐ Test
- ☐ Design
- ☐ Documentation
- ☐ Product support
- ☐ Management and administration
- ☐ Research
- ☐ Other

API upgrade behavior

6. How often do you upgrade the version of the libraries/APIs that you use? *

- ○ Never
- ○ Occasionally/Sometimes
- ○ Always

7. You have indicated that you rarely or never upgrade a dependency. Below is a list of reasons that could potentially prevent you from upgrading the library you are using.
Could you please indicate the frequency with which one of these reasons has prevented you in the past?

	Never	Almost never	Occasionally	Almost always	Always
Another dependency I use prevented me from upgrading an API (co-dependency issues)	○	○	○	○	○
I have a policy of freezing versions	○	○	○	○	○
New versions of the API do not offer any new functionality (i.e. the API only has incremental upgrades)	○	○	○	○	○
I only use what I need and that works well, so why upgrade?	○	○	○	○	○
New versions of the API break functionality that I use	○	○	○	○	○
A new version deprecates a feature that I use	○	○	○	○	○
I do not upgrade due to the cost involved in upgrading (in terms of time and money)	○	○	○	○	○
Enter another option	○	○	○	○	○

8. Below are ways in which developers have been found to respond to deprecated methods in their source code.
In your experience, how frequently have you done one of the following on encountering a deprecated method in your own code?

	Never	Almost never	Occasionally	Almost always	Always
Not react to the deprecation and allow the invocation to remain as is	○	○	○	○	○
Replace the deprecated call with the recommended replacement	○	○	○	○	○
Replace the deprecated call with a replacement from some other API	○	○	○	○	○
Replace the deprecated call with in-house functionality (code written by you or your team)	○	○	○	○	○
Replace the deprecated call with functionality from the Java Development Kit (JDK)	○	○	○	○	○
Delete the deprecated call and not replace with anything	○	○	○	○	○
Roll back the version of the API being used so that the call is no longer deprecated	○	○	○	○	○

9. Below are some reasons behind not reacting to a deprecated feature in an API.
In your experience, which of the following reasons lead to not reacting to a deprecated feature:

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
I find reacting is expensive and time consuming	○	○	○	○	○
I do not need to react since					

I do not need to react since deprecation does not break the code (i.e. the system is stable)					
I do not react because no suitable alternative is available (e.g. replacement is not good enough, is too complicated or unspecified)					
I did not know/was not aware of the deprecation (lack of tools or documentation to make me aware)					
I will not upgrade the library in the future to be affected by removal of the deprecated feature					
The deprecated feature will never be removed, thus not requiring a reaction					
The reason behind deprecation is unimportant or not given at all					
There is time to react since the API does not release frequently					
There are too many deprecated methods in my code base to react to					
I would rather rollback the version of the API I am using					
I choose to move to a new API that provides similar functionality					
I can do it better myself					
Enter another option					
Enter another option					

10. Below are situations that might prompt a reaction to a deprecated feature. Please indicate which one you most agree with:

	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
I react to a deprecated method if the system breaks	○	○	○	○	○
I react to a deprecated method if I want to update the library in the future	○	○	○	○	○
I react to a deprecated method if the cost to react is not high	○	○	○	○	○
I react to a deprecated method if the reason behind deprecation is important	○	○	○	○	○

Enter for lucky draw!

11. Please enter your email for the lucky draw!

Appendix B

APIs
junit
slf4j-api
log4j
mysql-connector-java
spring-core
commons-io
guava
hibernate-core
logback-classic
commons-lang3
mockito-all
jackson-databind
gson
httpclient
commons-dbcp
joda-time
commons-logging
aspectjrt
testng
commons-codec
commons-fileupload
h2
postgresql
aspectjweaver
hsqldb
commons-collections
json
hamcrest-all
cglib
selenium-java
javassist
jsoup
mybatis
standard
commons-beanutils
mongo-java-driver
poi
commons-cli
jersey-client
dom4j
c3p0
commons-httpclient
android
primefaces

commons-pool
guice
freemarker
assertj-core
easymock
jetty-server

References

Bajracharya S, Ngo T, Linstead E, Dou Y, Rigor P, Baldi P, Lopes C (2006) Sourcerer: a search engine for open source code supporting structure-based search. In: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. ACM, pp 681–682

Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S (2015a) How the apache community upgrades dependencies: an evolutionary study. Empir Softw Eng 20(5):1275–1317

Bavota G, Linares-Vasquez M, Bernal-Cardenas CE, Di Penta M, Oliveto R, Poshyvanyk D (2015b) The impact of api change-and fault-proneness on the user ratings of android apps, vol 41

Bholowalia P, Kumar A (2014) Ebk-means: a clustering technique based on elbow method and k-means in wsn. Int J Comput Appl 9:105

Bogart C, Kästner C, Herbsleb J, Thung F (2016a) How to break an api: cost negotiation and community values in three software ecosystems. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp 109–120

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

Brito G, Hora A, Valente MT, Robbes R (2016a) Do developers deprecate apis with replacement messages? a large-scale analysis on java systems. In: 23nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Osaka, p to appear

Brito G, Hora A, Valente MT, Robbes R (2016b) Do developers deprecate apis with replacement messages? a large-scale analysis on java systems. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, vol 1, pp 360–369

Brito G, Hora A, Valente MT, Robbes R (2018) On the use of replacement messages in {API} deprecation: an empirical study. J Syst Softw 137:306–321

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

Brooks Jr, FP (1995) The mythical Man-Month: Essays on software engineering, Anniversary Edition, 2/E Pearson Education India

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

Businge J, Serebrenik A, van den Brand M (2013b) Analyzing the eclipse API usage: Putting the developer in the loop. In: Proceedings of the 17th European Conference on Software Maintenance and Reengineering, CSMR, pp 37–46

Chacon S, Straub B (2014) Pro git, Apress

Chow K, Notkin D (1996) Semi-automatic update of applications in response to library changes. In: Proceedings of International Conference on Software Maintenance (ICSM), pp 359–368

Cossette BE, Walker RJ (2012) Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In: Proceedings of 20th International Symposium on the Foundations of Software Engineering (FSE). ACM, pp 55

Dagenais B, Robillard MP (2008) Recommending adaptive changes for framework evolution. In: Proceedings of 30th International Conference on Software engineering (ICSE)

Dagenais B, Robillard MP (2009) Semdiff: Analysis and recommendation support for api evolution. In: Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society. 599–602

Decan A, Mens T, Claes M, Grosjean P (2016) eLick When github meets CRAN: an analysis of inter-repository package dependency problems. In: Proceedings of the 23rd IEEE International Conference on Software Analysis. Evolution, and Reengineering (SANER,) pp 493–504

Decan A, Mens T, Claes M (2017) eLick An empirical comparison of dependency issues in OSS packaging ecosystems. In: Proceedings of the 24th IEEE International Conference on Software Analysis. Evolution and Reengineering (SANER,) pp 2–12

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

Dig D, Manzoor K, Johnson R, Nguyen TN (2007) Refactoring-aware configuration management for object-oriented programs. In: 29th International Conference on Software Engineering, pp 427–436

Espinha Tiago, Zaidman Andy, Gross Hans-Gerhard (2014) Web api growing pains: Stories from client developers and their code. In: 2014 Software Evolution week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE). IEEE, pp 84–93

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

Fisher RJ (1993) Social desirability bias and the validity of indirect questioning. J Consum Res 20(2):303–315

Eclipse foundation (2018) Eclipse ide, http://www.eclipse.org/

Furnham A (1986) Response bias, social desirability and dissimulation. Person Individ Differ 7(3):385–400

Gousios G, Spinellis D (2012) Ghtorrent: Github's data from a firehose. In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories. IEEE Press, pp 12–21

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

Holmes R, Walker RJ (2010) Customized awareness: recommending relevant external change events. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. ACM, pp 465–474

Hora A, Robbes R, Anquetil N, Etien A, Ducasse S, Valente MT (2015) How do developers react to api evolution? the pharo ecosystem case. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp 251–260

Hora AC, Valente MT, Robbes R, Anquetil N (2016) When should internal interfaces be promoted to public? In: Proceedings of the 24th ACM, SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp 278–289

Hou D, Yao X (2011) Exploring the intent behind api evolution: A case study. In: 2011 18th Working Conference on Reverse Engineering (WCRE). IEEE, pp 131–140

Jansen H (2010) The logic of qualitative survey research and its position in the field of social research methods, In Forum Qualitative Sozialforschung/Forum: Qualitative Social Research, vol 11

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

Kapur P, Cossette B, Walker RJ (2010) Refactoring references for library migration. ACM, vol 45

Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? Empir Softw Eng 23(1):384–417

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

Marks S (2017) JEP 277: Enhanced Deprecation, http://openjdk.java.net/jeps/277, 2014–2017 last accessed

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

Nguyen HA, Nguyen TT, Wilson Jr G, Nguyen AT, Kim M, Nguyen TN (2010) A graph-based approach to api usage adaptation. In: Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications, pp 302–321

Perkins JH (2005) Automatically generating refactorings to support api evolution. In: ACM SIGSOFT Software Engineering Notes. ACM, vol 31, pp 111–114

Raemaekers S, van Deursen A, Visser J (2012) Measuring software library stability through historical version analysis. In: 28th IEEE International Conference onSoftware Maintenance (ICSM). IEEE, pp 378–387

Raemaekers S, Van Deursen A, Visser J (2014) Semantic versioning versus breaking changes: A study of the maven repository. In: Technical Report Series TUD-SERG-2014-016 Accepted for publication by the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014). Delft University of Technology, Software Engineering Research Group, Victoria

Raemaekers S, van Deursen A, Visser J (2017) Semantic versioning and impact of breaking changes in the maven repository. J Syst Softw 129:140–158

Robbes R, Lungu M, Röthlisberger D (2012) How do developers react to api deprecation?: the case of a smalltalk ecosystem. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, pp 56

Rose JR (2017) How and when to deprecate apis, http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase11-419415.html#7122-jdk-1.1-doc-oth-JPR, 1996 last accessed

Ṡavga I, Rudolf M (2007) Refactoring-based support for binary compatibility in evolving frameworks. In: Proceedings of the 6th international conference on Generative programming and component engineering. ACM, pp 175–184

Sawant AA, Robbes R, Bacchelli A (2016) On the reaction to deprecation of 25,357 clients of 4+ 1 popular java apis. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp 400–410

Sawant AA, Bacchelli A (2017) Fine-grape: fine-grained api usage extractor–an approach and dataset to investigate api usage, Empirical Software Engineering, pp 1–24

Sawant AA, Robbes R, Bacchelli A (2018) On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk. Empir Softw Eng 23(4):2158–2197

Sawant AA, Huang G, Vilen G, Stojkovski S, Bacchelli A (2018a) Why are features deprecated? an investigation into the motivation behind deprecation. In: Proceedings of the 34th IEEE International Conference on Software Maintainance and Evolution. IEEE, p in print

Sawant AA, Aniche M, van Deursen A, Bacchelli A (2018b) Understanding developers' needs on deprecation as a language feature. In: Proceedings of the 40th ACM/IEEE International Conference on Software Engineering, ICSE '18, pp561–571

Schäfer T, Jonas J, Mezini M (2008) Mining framework usage changes from instantiation code. In: Proceedings of 30th International Conference on Software Engineering (ICSE), pp 471–480

Spencer D, Warfel T (2004) Card sorting: a definitive guide. Boxes and Arrows:2

Teyton C, Falleri J-R, Palyart M, Blanc X (2014) A study of library migrations in java. J Softw: Evol Process 26(11):1030–1052

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

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

Wu W, Adams B, Guéhéneuc Y-G, Antoniol G (2014) Acua: Api change and usage auditor. In: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, pp 89–94

Wu W, Khomh F, Adams B, Guéhéneuc Y-G, Antoniol G (2016) An exploratory study of api changes and usages based on apache and eclipse ecosystems. Empir Softw Eng 21(6):2366–2412

Xavier L, Brito A, Hora AC, Valente MT (2017) Historical and impact analysis of API breaking changes: A large-scale study. In: Proceedings of the 24th IEEE International Conference on Software Analysis. Evolution and Reengineering, (SANER,) pp 138–147

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

Xing Z, Stroulia E (2007) Api-evolution support with diff-catchup. IEEE Trans Softw Eng 33(12):818–836

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

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

Alberto Bacchelli is an SNSF Professor in Empirical Software Engineering in the Department of Informatics in the Faculty of Business, Economics and Informatics at the University of Zurich, Switzerland. He received his B.Sc. and M.Sc. in Computer Science from the University of Bologna, Italy, and the Ph.D. in Software Engineering from the Università della Svizzera Italiana, Switzerland. Before joining the University of Zurich, he has been assistant professor at Delft University of Technology, The Netherlands where he was also granted tenure. His research interest is empirical software engineering, with a current focus on peer code review, programming language features, and the fundamentals of software analytics.