Why are features deprecated? An investigation into the motivation behind deprecation

Sawant, Anand; Huang, Guangzhe ; Vilen, Gabriel; Stojkovski, Stefan ; Bacchelli, Alberto

Publication date
2018

Published in
34th IEEE International Conference on Software Maintenance and Evolution (ICSME)

Citation (APA)
Sawant, A., Huang, G., Vilen, G., Stojkovski, S., & Bacchelli, A. (Accepted/In press). Why are features deprecated? An investigation into the motivation behind deprecation. In 34th IEEE International Conference on Software Maintenance and Evolution (ICSME) IEEE.

Why are features deprecated? An investigation into the motivation behind deprecation

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

Guangzhe Huang, Gabriel Vilen, Stefan Stojkovski
Delft University of Technology
Delft, The Netherlands
(G.Huang-1, H.G.Vilen, S.Stojkovski)@student.tudelft.nl

Alberto Bacchelli
University of Zurich
Zurich, Switzerland
bacchelli@uzh.ifi.ch

***Abstract*—In this study, we investigate why API producers deprecate features. Previous work has shown us that knowing the rationale behind deprecation of an API aids a consumer in deciding to react, thus hinting at a diversity of deprecation reasons. We manually analyze the Javadoc of 374 deprecated methods pertaining four mainstream Java APIs to see whether the reason behind deprecation is mentioned. We find that understanding the rationale from just the Javadoc is insufficient; hence we add other data sources such as the source code, issue tracker data and commit history. We observe 12 reasons that trigger API producers to deprecate a feature. We evaluate an automated approach to classify these motivations.**

I. Introduction

An Application Programming Interface (API) is a set of defined functionalities provided by a programming library or framework.[1] APIs promote the reuse of existing software components [2]. By integrating a third-party API in a code base, a developer can save development time and effort and use a well-tested system.

To remain useful in a mutating environment [3], most APIs evolve by introducing new features, removing older ones, and changing existing features. Some of the changes due to API's evolution can be breaking in nature and can have an adverse impact on the API consumers [4]. One way for API producers to avoid directly introducing a breaking change in their API first to *deprecate* the feature, thus communicating a warning to the consumers. Deprecation is available in most mainstream languages such as C#, PHP, and Java. Regarding a definition of deprecation, the official Java documentation states: "A program element annotated @Deprecated is one that programmers are discouraged from using, typically because it is dangerous, or because a better alternative exists" [5].

API deprecations are commonplace [6], however, to what extent is their motivation clear? Recent research has reported that API consumers decide on whether to react to API deprecation, based on the reason behind the deprecation [7], thus hinting at the several possible purposes for deprecating a feature. Indeed, deprecation is a convenient way for API producers to make sure that both popular IDEs and compiler inform API consumers that something is not right—deprecation is a *unique communication mechanism* and, as such, can be used for conveying different messages.

[1]In this paper, with the term *API* we refer only to local APIs (*e.g.*, frameworks and libraries), as opposed to web-APIs [1].

The goal of this study is investigating the reasons behind feature deprecation. The critical motivations for pursuing this goal include: (1) gaining a deeper understanding of a popular language feature, from a new angle, (2) discovering unmet developers' communication needs, by uncovering unorthodox usages of deprecation,[2] which may signal those needs, (3) investigating what deprecation says regarding APIs' evolution. The results can inform and guide practitioners' practices as well as future academic studies on this and similar mechanisms, and on developers' communication.

To this aim, we conduct an in-depth analysis of 374 deprecated features in four popular Java APIs: Spring [9] (15,086 users), Hibernate [10] (8,143 users), Guava [11] (9,542 users), and Easymock [12] (1,484 users). Our study is exploratory and we answer three research questions: why are API features deprecated, what is the frequency of deprecation rationales, and how well can we automatically classify the reason for a deprecation from software repositories.

To uncover the various rationales for a deprecation we manually analyze over 1,100 documents relating to 374 deprecated features. Three authors conducted this analysis and a fourth validated it. This effort results in the creation of a taxonomy of 12 rationales behind deprecation. We then investigate what rationales have been used most frequently across the considered APIs. Finally, we employ a supervised machine learning approach [13] to create an automated approach to infer the rationale behind a deprecation. We evaluate the performance by using different cross validation [14] techniques.

We found that determining the reason for deprecating a feature is far from trivial: The motivation is rarely mentioned in the accompanying Javadoc. Nevertheless, through the analysis of software repositories (mainly, code versioning and issue tracking systems) we could define a taxonomy of 12 high-level reasons. Of these, two are unorthodox uses of deprecation (for temporary features and for incomplete implementations), thus indicating unmet developers' communication needs. Finally, we found that an automated approach to classify the deprecation reasons based on machine learning reaches promising results, but only if trained on project-specific instances.

[2]In the opinion of the Java language designers, developers often misuse the deprecation mechanism [8].

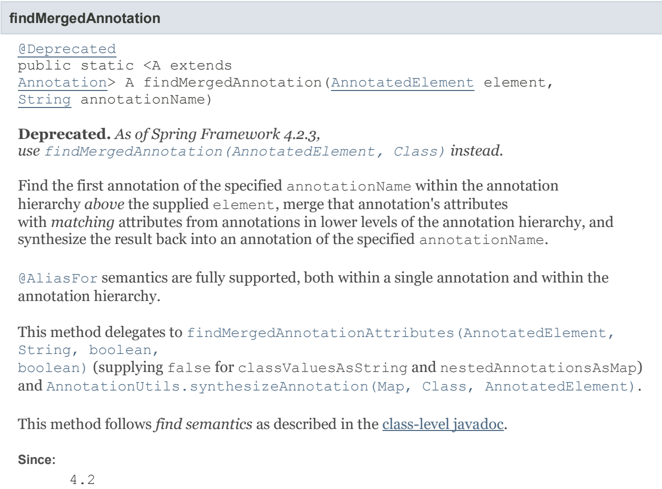


Figure 1. Javadoc of a deprecated API feature from the Spring API [15].

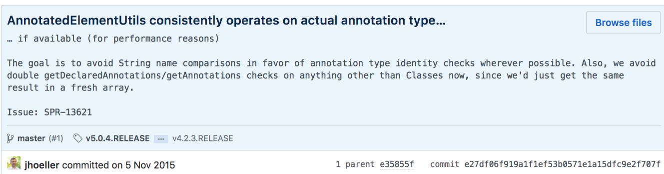


Figure 2. The commit deprecating the feature, pointing to the JIRA issue [16].

II. Motivation

Figure 1 shows an example deprecation message from the Spring framework. We see that the API producers have deprecated the feature in version 4.2.3 of the API and they recommend that the consumer use an alternative feature. The Javadoc does not explain the rationale behind this change. By recovering the commit (seen in Figure 2) in which this feature was deprecated, we find that it contains no rationale behind the deprecation; hence, we have to refer to the JIRA issue ID mentioned in the commit. From the Spring issue tracker post (seen in Figure 3), we see that there is a performance slowdown when using specific methods from the Annotation class. To rectify this, Spring introduced a replacement feature to fix the issue and deprecated the original element.

This example shows that the deprecation of a feature itself does not necessarily carry its reason, but uncovering (although complicated, as in this case) and understanding the possible types of deprecation reasons is relevant from many perspectives, including those we describe in the following.

(1) To guide practitioners and research tools. API consumers have indicated that knowing the motivation behind deprecation is critical to decide whether to react [7]. Empirically uncovering the possible reasons for deprecation can suggest to practitioners whether they should respond to a deprecation in principle, as well as whether the motivations are project-specific or can be mostly generalized. Knowing deprecation reasons can inform the design of tools to better support the replacement of a deprecated feature, by exploiting the deprecation reason.

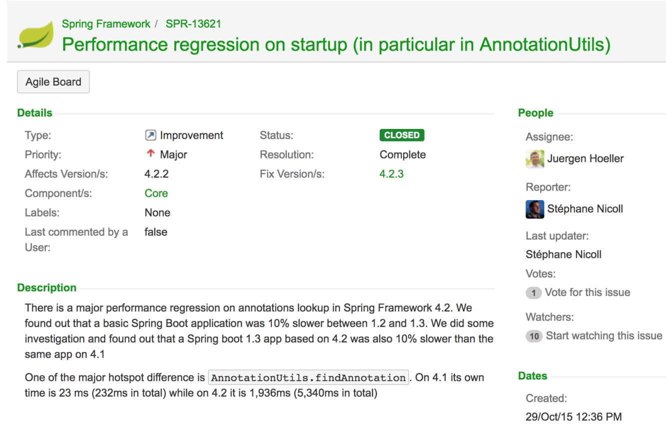


Figure 3. Issue detailing the need for changing the deprecated method [17].

(2) To uncover unmet communication needs. When Java was in the process of changing its deprecation mechanism for Java 9, a motivation was that API producers were misusing the deprecation mechanism. This misusage may signal that deprecation is used to fulfill a communication need that is unmet by any other tool. Knowing the various reasons behind deprecation allows us to understand how many different cases of misuse of the deprecation mechanism have taken place and may let us discover what needs future research should address to devise a more appropriate communication tool.

(3) Understand how an API evolves. APIs evolve and replace old features with new ones. The older features are deprecated and not directly removed from the API to minimize the number of breaking changes introduced. Knowing what reasons an API uses most popularly can aid researchers in gaining a deeper understanding as to how and why APIs evolve [18].

(4) Understanding to what extent API documentation is lacking. API documentation is an essential tool that aids API consumers in effectively and accurately using an API's features [19]. API producers must invest in the documentation for their API so that they ease the burden of adoption of API features [20]. In the case of deprecated API features, giving the consumers an indication as to what new feature should be used and how, is essential [21], [22]. In addition to that, explaining the rationale behind the deprecation and providing a timeline for the removal of the deprecated feature have been found to be essential to an API consumer. We see in Figure 1 that the rationale is impossible to infer, a consumer has to read the JIRA issue on the subject (seen in Figure 3). In this study, we get an indication to what extent API documentation indicates the reason behind deprecation and conveys the same to the consumer, or where it can be found, thus informing practitioners, as well as researchers investigating tools to support API documentation.

III. Methodology

The *goal* of the study is to empirically investigate and classify the reasons that triggered the deprecation of features in popular APIs. The *perspective* is of researchers and practitioners, interested in an empirical understanding of the reasons behind deprecation, to guide practice and future research.

Our study revolves around three research questions:

RQ_1: How can reasons for deprecating features be categorized? With the first research question, we seek to investigate and classify the diversity of reasons that triggered API producers to deprecate a feature in their systems. We do this by manually analyzing the information about these features and their deprecation as they are available in software repositories.

RQ_2: How often does every reason for deprecation occur? After having categorized the reasons triggering deprecation, we analyze their frequency to quantify the different purposes of API producers.

RQ_3: How effective is an automated approach in classifying the reason behind a deprecation? Finally, we exploit the set of manually categorized reasons to investigate how effectively we can automatically classify the rationale via standard machine learning techniques, using the relevant data from the software repositories. Should the results of this automatic classification be promising, future research could investigate tools to automatically augment existing API documentation with the rationale behind the deprecation, thus providing useful information [7] to the API consumers.

A. Subjects: Systems and Deprecated Features

In this study, we focus primarily on the Java ecosystem, because (1) Java is the most popular programming language [23], (2) this ecosystem has a large number of popular and mature APIs for study, and (3) the deprecation mechanism in Java is very prominent and widely used by API producers [6].

Systems. From the Java ecosystem, we select four third-party open-source software APIs.[3] Our goal and research methods dictate the choice of limiting ourselves to four APIs. On the one hand, we strive to collect as many diverse reasons as possible to increase our empirical understanding of this phenomenon; on the other hand, we can realistically investigate no more than a few hundred deprecated features, because understanding the reason of deprecation requires perusing a possibly large number of documents per feature (as seen in the example in Figures 1–2). Given these requirements, investigating a large number of systems is suboptimal: Keeping the number of features we can analyze equal, it is reasonable to think that we are more likely to find a smaller diversity of reasons in more systems (*i.e.*, we find only the most occurring reasons per system), than in fewer systems but studied more in-depth. Hence we limit ourselves to four systems.

As criteria for the choice of the four systems, we consider popularity (as defined by the number of Java projects on GitHub that use the system; we use the dataset by Sawant and Bacchelli to benchmark the popularity [24], [25]), size, length of history, number of deprecated features, availability of software repositories, and diversity in producers (*e.g.*, we would not consider two APIs from Google) as well as domain. Table I describes the APIs we eventually selected (*i.e.*, Guava, Spring, Easymock, and Hibernate).

Table I
OVERVIEW OF SELECTED APIS

API	Description	Considered Release	Number of Consumers
Easymock	Java object mocking framework	3.5.1	1,484
Guava	Google's collections library	23.0	9,542
Hibernate	Object/relational mapping framework	5.2.12	8,143
Spring	Dependency injection framework	5.0.0	15,086

Deprecated features. We focus on the latest available version of each API. Since these are all Java-based projects, we use the Eclipse JDT AST parser [26] to identify all the deprecated features. The resulting dataset contains almost 2,300 deprecated methods from the four APIs. We randomly select the methods from each of the APIs to create our sample investigation set. Considering that we want to estimate proportions of reasons for RQ2, we choose a sample set size that leads to a 95% confidence level and a margin error of no more than 5% on the computed proportions [27]; this resulted in a sample of 374 deprecated features to manually investigate, together with information from other relevant data sources.

B. RQ1. Manually determining the reasons for a deprecation

To answer RQ1, we follow a three-step method. Three authors of this paper conducted the first (S1) and second (S2) steps, while the fourth conducted the third step (S3). S1 regards the determination of the rationale behind the deprecation of an individual feature, S2 regards the grouping of individual deprecation reasons into high-level categories to create a taxonomy of reasons, S3 validates the results of the first two steps. In the following, we detail each step.

S1. Determining the reason of an individual deprecation. This step is conducted by three authors of this paper together. For each feature, they start by inspecting the documentation that is supposed to contain the rationale and replacement for the deprecation [5]: The Javadoc associated with the feature.

They found that (1) most Javadoc messages include the annotation $@link$ that links to the alternative feature that should be used instead of the deprecated feature, but (2) the message seldom includes the rationale behind the deprecation. This lack of rationale made it unfeasible to understand the reason from only the Javadoc. Thus, the investigation is expanded to include data from other software repositories, which are then inspected by the three authors:

1) **Commit history.** The commit message for a change can contain the rationale behind it and the nature of the change. Thus, we use the JGit project to traverse the history of each file in the master branch of the API. We then isolate the commit wherein one of the deprecated entities was first

[3]We do not consider the Java JDK API for our analysis because uncovering the rationale behind deprecation is not always possible as tracing alternative sources of information (*e.g.*, issue trackers and developers' communication) is hindered by the closed nature of the Oracle JDK.

deprecated. We, thus, inspect the accompanying commit message.

2) Source code. Source code comments (not Javadoc) can often contain the rationale behind changes made to a method. These comments are usually for the benefit of the subsequent contributor to this method or file. For each of the deprecated methods, we isolate the entire source code of the method.

3) Issue tracker. Issue trackers contain discussions among developers and information on issues in the API. The rationale behind a change can be understood from the discussions and issues posted in the issue tracker if they pertain to the method under investigation. We manually isolate the issues (from JIRA or the GitHub issue tracker) mentioned in the commit messages that deprecated a feature.

4) Other sources. We perform a cursory investigation of sources such as StackOverflow, the Google search engine, developer blogs and mailing lists specific to each API. Each of these sources, for example email [28], [29], contains API consumer-/producer-driven content that can contain information on the rationale behind the deprecation of a feature. However, we found that sources are not always consistent and do not contain the information that we require.

Through the analysis of the aforementioned sources, the three authors determine a precise reason for the deprecation of each feature.

S2. A taxonomy of deprecation reasons. In the second step, the same three authors conduct three iterative content analysis sessions [30] to group the individual rationales used into higher level reasons. Iteratively, for each rationale found in the previous step, the involved authors verify whether they have previously identified a reason of this nature to which this rationale can be assigned or whether they need to create a new reason. This iterative process resulted in a taxonomy of 12 reasons for the deprecation of features.

S3. Validation. As the third and last step, another author independently repeats the analysis to verify both (i) the understandability of the category descriptions in the taxonomy from the second step and (ii) the assignment of deprecated features to these categories. The resulting inter-rater agreement between the two classifications was 93%; the authors discussed the 7% that was not agreed upon until they reached a consensus. In Section IV we present the final taxonomy.

C. RQ2. Frequency of the deprecation reasons

In this research question, we aim at analyzing how frequently each category of our taxonomy appears. To this aim, we compute the frequency with which each high-level category of deprecation reason is assigned to an individual deprecated feature during the iterative content analysis. In Section V we present and discuss the results, overall and by API.

D. RQ3. Automatic classification of deprecation reasons

In our third research question, we investigate standard machine learning techniques to automatically classify the reasons of a deprecation into the taxonomy identified in RQ1. While employing a sophisticated method such as deep learning goes beyond the scope of the current work we aim to create an automatic classification technique with a fair level of accuracy.

Machine learning approaches. We employ a supervised machine learning approach [13] to create our automated inference approach. With this approach, a set of features are used to predict the value of a variable (in our case, the *classification* of the reason) using a machine learning classifier (*e.g.*, Naive Bayes [13]). The role of the classifier is to determine the importance and role of each feature in predicting the classification by learning from already classified examples. In particular, we consider two different kinds of supervised classifiers: (1) probabilistic classifier (specifically, naive Bayes multinomial) and (2) decision tree algorithm (specifically, random forest).These classifiers make different assumptions on the underlying data, as well as have distinct advantages/drawbacks for execution speed and overfitting.

Features. To classify the reason for deprecation, we have the textual data (Javadoc comment, commit message, and issue tracker data) that describes the deprecated feature at our disposal. For this reason, we reduce our task to a text classification problem [13], which we tackle adapting the widespread *Vector Space Model* (VSM) [31]. VSM considers each document (*i.e.*, the deprecated feature and all the relevant text) as a vector of identifiers (*i.e.*, in our case, all the terms that appear in the whole set of available texts in our dataset) whose value is determined by the normalized number of occurrences of each identifier in the document (*e.g.*, 0 if the term never occurs). The identifiers given as output from VSM represent the features for the machine learner and the normalized word counts are the corresponding values.

To determine the terms to consider for VSM, we create a vocabulary by tokenizing each textual resource. We split tokens on whitespace, special characters, and punctuation; moreover, we split variable names that are CamelCased into individual entities. Finally, we do not alter Javadoc tags such as '@deprecated' and '@link'.

Dataset and Evaluation. To train and test the performance of the proposed machine learning approach, we use the dataset produced in RQ1 and RQ2; then we mainly adopt *n-Fold Cross Validation* [14]. This strategy randomly partitions (using stratified sampling to maintain the proportion of classes) the data into *n* folds of equal size, then *n-1* folds are used as training and the last as testing. The process is repeated *n* times, using each time a different fold as a test set. The performance of the experimented models is computed using widespread classification metrics such as precision and recall; in our paper, for space reasons, we report the *percentage of correctly classified instances*, while the full results are available in the accompanying replication package [32].

E. Threats to validity

Construct validity. In the manual analysis of the rationale behind deprecating specific features, we may have misclassified or missed out on certain motivations behind deprecation.

We ensure the accuracy of our classification by having three authors simultaneously manually analyze all the samples in our dataset and create an initial categorization of the rationale, followed by another author repeating the manual classification process to ensure accuracy. To ensure that we uncover most motivations behind deprecation, we limit ourselves to 4 mainstream Java APIs that pertain to different domains and have different developers and characteristics.

Generalizability. Having focused only on the Java ecosystem, the rationales that we have uncovered may apply only to the Java-based APIs and not to APIs in other languages. We mitigate this by trying to ensure that the rationale we discover is not Java specific, rather as abstract as possible. Furthermore, Java is the most popular language with a deprecation mechanism and other object-oriented languages share similar development practices.

IV. RQ1 Results: Diversity of Reasons

We describe each category in the taxonomy that resulted from our analysis, reporting examples from our dataset.

BC. Avoid bad coding practices

Example Javadoc

```
/**
 * Allow injection of the dialect to use.
 * @deprecated The intention is that Dialect should be required to be
 * specified up-front and it would then get ctor injected.
 * @param dialect The dialect
 */
 @Deprecated public void setDialect(Dialect dialect)
```

One of the principal goals of APIs is to provide a set of features to a consumer that can be integrated without introducing issues or bad coding practices in the consumers' code base. There are certain cases where the API does not always achieve this goal. In the example above, it is preferred that the `Dialect` should be specified up front as that would allow the `Dialect` object to be injected directly in the constructor. Using a setter method of a class implicitly means that the dependency is optional, constructor injection instead is used when the class cannot function without the dependency.

DP. Design pattern

Example Javadoc

```
/**
 * Creates a mock object that extends the given class, order checking is
 * enabled by default.
 * @param < T > the class that the mock object should extend.
 * @param name the name of the mock object.
 * @param toMock the class that the mock object should extend.
 * @param constructorArgs constructor and parameters used to instantiate the
 * mock.
 * @param mockedMethods methods that will be mocked, other methods will
 * behave normally
 * @return the mock object.
 * @deprecated Use {@link #createMockBuilder(Class)} instead
 */
 @Deprecated public <T>T createStrictMock(final String name,final Class<T>
   toMock,final ConstructorArgs constructorArgs,final Method...mockedMethods)
```

Partial mocking is a very nice feature, but having to use the reflection API directly to get the constructor and methods is less than ideal, so we created a MockBuilder which we've been using for this.

Example commit message

We find cases in which API producers deprecate the old feature and slowly phase them out as consumers are encouraged to use the new version of the functionality that makes use of a design pattern. In the example above, the project moved from accessing a feature through reflection to using the design pattern named Builder.

DU. Dissuade usage

Example Javadoc

```
/**
 * Not supported. Use {@link
 * ImmutableSortedMultiset#toImmutableSortedMultiset} instead.
 * This method exists only to hide {@link
 * ImmutableMultiset#toImmutableMultiset} from consumers of {@code
 * ImmutableSortedMultiset}.
 * @throws UnsupportedOperationException always
 * @deprecated Use {@link ImmutableSortedMultiset#toImmutableSortedMultiset}.
 * @since 21.0
 */
@Deprecated public static <E>Collector<E,?,ImmutableMultiset<E>>
  toImmutableMultiset(){
 throw new UnsupportedOperationException();
}
```

We find cases where API producers implement an interface in a class, without implementing all of its methods. The un-implemented methods are marked as deprecated so that the consumer is given an indication (as a compiler warning) that this feature should not be used. In the example above, the Javadoc also recommends a replacement.

FD. Functional defects

Example Javadoc

```
/**
 * Compare 2 arrays only at the first level
 * @deprecated Use {@link java.util.Arrays#equals(char[],char[])} instead
 */
@Deprecated public static boolean isEquals(char[] o1, char[] o2)
```

org.hibernate.internal.util.compare.EqualsHelper doesn't consider arrays when comparing objects for equality. Since EqualsHelper is used in many places, such as dirty checking, this problem results in unexpected behavior, such as array type fields always being considered dirty.

Issues related to this problem include:
HHH-4110 CLOSED
HHH-2482 CLOSED
HHH-7810 OPEN
HHH-3009 CLOSED
HHH-7496 CLOSED (probably)

Example issue tracker message

The introduction of flaws in API features is inevitable. We find that, at times, API producers deprecate features with defects instead of removing them. We see an example in the deprecated method above where the implementation of the equals method does not consider arrays when comparing objects for equality, which in turn causes issues in other parts of the API, as seen in the extract from the related issue report.

ME. Merged to existing method

Example Javadoc

```
/**
 * Returns an equivalence that delegates to {@link Object#equals} and {@link
 * Object#hashCode}. {@link Equivalence#equivalent} returns {@code true} if
 * both values are null, or if neither value is null and
 * {@link Object#equals} returns {@code true}. {@link Equivalence#hash}
 * returns{@code 0} if passed a null value.
 * @deprecated use {@link Equivalences#equals}, which now has the null-aware
 * behavior
 */
 @Deprecated public static Equivalence<Object> nullAwareEquals()
```

We found instances in which an API provides two features achieving the same end goal, but one has more nuance associated with it and performs extra checks. Over time, the API producer decides to combine these different checks into the same feature, thus resulting in the other being deprecated. In the above example, the `equals` method in the `Equivalences` class now performs a null check, thus rendering the `nullAwareEquals` obsolete.

NF. New feature introduced

Example Javadoc

```
/**
 * @deprecated Use {@link ValueGraph#equals(Object)} instead. This method
 * will be removed in late 2017.
 */
@Deprecated public static boolean equivalent(@Nullable ValueGraph<?,?>
  graphA,@Nullable ValueGraph<?,?> graphB)
```

Now that ValueGraph no longer extends Graph, change all the common.graph interfaces to handle equals()/hashCode() "normally". Deprecate Graphs.equivalent().

Example commit message

Sometimes, when API producers introduce a new feature, the design of the project is also changed. In these cases, the producers deprecate the older, superseded features.

ND. No dependency support

Example Javadoc

```
/**
 * Expect any boolean but captures it for later use.
 * @param captured Where the parameter is captured
 * @return 0
 * @deprecated Because of harder erasure enforcement, doesn't compile in
 * Java 7
 */
@Deprecated public static boolean capture(final Capture<Boolean> captured)
```

Over time APIs upgrade the dependencies on which they depend. With these upgrades, specific features in the API can no longer be supported and need to be removed and replaced with modern functionality. In the cases we analyzed, upgrades in the Java version often cause the incompatibilities. In the example above we see that the `capture` method is not supported in Java 7 and becomes deprecated.

RD. Redundant methods

Example Javadoc

```
/**
 * @deprecated since 5.2, to be removed in 6.0 with no replacement.
 */
@Deprecated public ModificationStore getStore()
```

From a discussion with Adam, there was supposed to be another value, DIFF that would store the diff of String-based value fields. It was never implemented and isn't … needed, so [it's] safe to deprecate and remove.

Example commit message

Redundant is code that is neither required nor essential and need not be executed. The negative consequence of redundant code is that it results in bloated source code and reduced maintainability. We found cases in which the API producers deprecated a feature when it is useless or no longer necessary.

RN. Renaming of feature

Example Javadoc

```
/**
 * Old name of {@link #getHost}.
 * @deprecated Use {@link #getHost()} instead. This method is scheduled for
 * removal in Guava 22.0.
 */
@Deprecated public String getHostText()
```

The considered APIs have been developed over a long time by multiple developers, thus contain inconsistencies in the naming convention. These inconsistencies might have been introduced due to a lack of foresight or a change in the API's nomenclature convention. Just renaming a feature to adhere to new norms would break consumer code and hence would not be backward compatible. The existing name is kept in place. In fact, we notice in the cases that we manually analyze that the original name is intended to be left in the API indefinitely, *i.e.*, there are no plans to remove such features. However, API producers do deprecate the feature with the incorrect name and encourage consumers to adopt the new feature that adheres to the naming convention of the project.

SF. Security flaws

Example Javadoc

```
/**
 * Returns a hash function implementing the MD5 hash algorithm (128 hash
 * bits).
 * @deprecated If you must interoperate with a system that requires MD5, then
 * use this method, despite its deprecation. But if you can choose your hash
 * function, avoid MD5, which is neither fast nor secure. As of January 2017,
 * we suggest: For security: {@link Hashing#sha256} or a higher-level
 * API. For speed: {@link Hashing#goodFastHash}, though see its docs for
 * caveats.
 */
@Deprecated public static HashFunction md5()
```

A security vulnerability might have been inadvertently introduced in a feature of an API at its inception or over time, thus requiring the immediate action of the API producers to address the issue. The producer deprecates the flawed feature and replaced it with one which does not suffer from the same flaw. In this way, the producer warned the consumer in the documentation that usage of such a feature is unsafe.

SC. Separation of concerns

Example Javadoc

```
/**
 * @deprecated Use {@link #setImplicitNamingStrategy} or {@link
 * #setPhysicalNamingStrategy} instead
 */
@Deprecated public void setNamingStrategy(String namingStrategy)
```

In object-oriented programming, each class or module is supposed to have its responsibilities. Sometimes an API feature can do too many things simultaneously, *i.e.*, it has too many responsibilities. To fix this, we found cases in which the API producer decided to split a single feature into multiple ones, also deprecating the original feature and creating a transition guide.

TF. Temporary feature

Example Javadoc

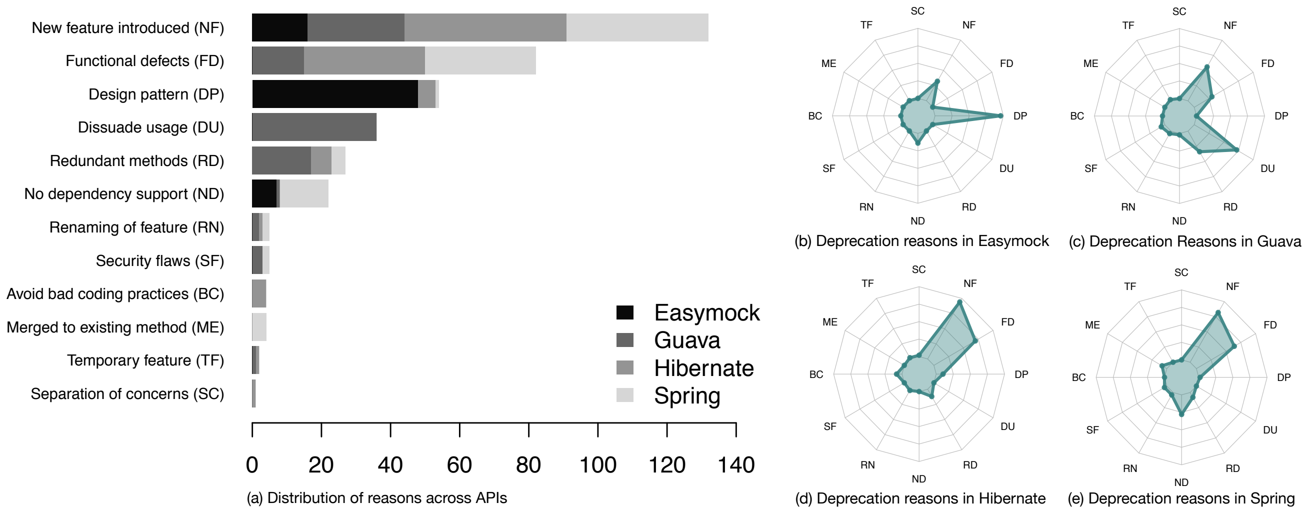
```
/**
 * @deprecated (since 5.2.1), while actually added in 5.2.1, this was added
 * to cleanup the audit strategy interface temporarily.
 */
@Deprecated public EnversService getEnversService()
```

An API producer might introduce a feature only for a temporary purpose to aid consumers in using certain functionality. Once, this is no longer needed, the temporary feature might be deprecated. We see that such temporary features are planned to be removed almost instantly from the API so that no consumer actually has the opportunity to use it in a future version.

Finding 1. The analysis of 374 deprecated features and over 1,100 accompanying documents yielded 12 rationales that API producers have used to deprecate a feature, thus showing a sizeable diversity of purposes.

V. RQ2 Results: Frequency of Reasons

After having categorized and described the *diversity* of reasons that led API producers to deprecate one of their API features, we now focus on determining how each of these reasons is prevalent in our dataset. Figure 4 reports the results by reasons in overall decreasing frequency (left-hand side) and by API (right-hand side). Overall, introducing a new feature (NF), the presence of functional defects (FD), the replacement with a design pattern (DP) account for the majority of reasons (268 cases out of 374 or 72%). The most frequent reason (NF) is the only one that appears in all the systems in our dataset,



(a) Distribution of reasons across APIs
(b) Deprecation reasons in Easymock
(c) Deprecation Reasons in Guava
(d) Deprecation reasons in Hibernate
(e) Deprecation reasons in Spring

Figure 4. Frequencies of reasons for deprecating features, by API

while the others have a more project-specific prevalence. We now describe the results by API.

Easymock. The most frequent reason for deprecation in Easymock is the implementation of a new design pattern that required a change in the interface. There are also 7 cases where the feature in their API was not supported by a newer version of Java. In 16 cases a better implementation of the same feature was superseding the existing one. Overall, we see that in Easymock most deprecations are not of a grave nature: A consumer could safely continue using a deprecated feature from this API.

Guava. Developers in Guava predominantly introduce new features to replace existing ones, hence use the deprecation mechanism. There are also some cases where there were functional defects in the feature and some instances of having security flaws. There is one feature that has been deprecated due to it being introduced as a temporary feature. In 36 cases, Guava deprecates a feature to dissuade usage of it due to an incomplete implementation of an interface; this is a case of misuse of the deprecation mechanism. Here the feature was not deprecated due to it being superseded by a new feature or because the feature had become obsolete. In most cases, it may be safe to continue using a deprecated feature from Guava, but the reasons for deprecation are diverse and one needs to verify them first.

Hibernate. In the case of Hibernate, developers have deprecated almost 50% of the deprecated functions due to the presence of functional issues in the features. This means that when Hibernate deprecates a feature it is usually to fix major issues in the API. In the other cases, the features are deprecated due to new functionality being introduced, because of redundancy, or because it encourages bad coding practices.

Spring. In 41 cases Spring has replaced an existing feature with a better implementation. Although Spring is an old and well-tested API, there have been 32 functional flaws to fix and 2 security issues as well. Spring also has deprecated features due to incompatibilities with newer versions of Java. Overall, there seems to be some danger to using deprecated features from Spring and, in many cases, a consumer needs to replace a deprecated feature with its successor.

> **Finding 2**. Introducing a new feature, the presence of a functional defect, and change of interfaces are the most frequent reasons for deprecating an API. However, only the first is shared across all projects.

VI. RQ3 RESULTS: AUTOMATIC REASON CLASSIFICATION

Our RQ3 investigates to what extent a machine learning approach can automatically classify the reason for deprecation.

A. *Methodological details*

Although we always use VSM (Section III-D), we progressively add more information sources to evaluate their effect on the classification. In the first stage, we only use tokens from Javadoc comments to classify the rationale, in the second we add commit messages, in the third, we add issue tracker data.

We evaluate three training/testing conditions: (1) 10-fold cross validation within the same API (*i.e.*, we evaluate each system separately), (2) overall 10-fold cross validation (*i.e.*, we merge all the instances in a single dataset), (3) cross-project validation (*i.e.*, we use the instances from three systems for training and test the resulting model on the last system; we rotate the test system each time).

B. *Results*

Since random forest always outperformed the naive Bayes multinomial classifier, we only report results for the former.

Within system validation. The first four groups in Table II report the results of the classifier when tested within the same

Table II
CLASSIFICATION RESULTS, USING 10-FOLD CROSS VALIDATION WITHIN THE SAME SYSTEM AND ACROSS ALL SYSTEMS.

		% correct instances	K	weighted avg.	
				recall	ROC
Guava	JD	0.883	0.843	0.883	0.963
	JD+CM	0.893	0.855	0.893	0.956
	JD+CM+IT	0.903	0.868	0.903	0.963
Easymock	JD	1.000	1.000	1.000	1.000
	JD+CM	0.986	0.970	0.986	1.000
	JD+CM+IT	NA	NA	NA	NA
Hibernate	JD	0.610	0.334	0.610	0.777
	JD+CM	0.740	0.559	0.740	0.862
	JD+CM+IT	0.720	0.526	0.720	0.866
Spring	JD	0.640	0.477	0.640	0.794
	JD+CM	0.850	0.780	0.850	0.956
	JD+CM+IT	0.880	0.824	0.880	0.962
All	JD	0.759	0.686	0.759	0.915
	JD+CM	0.853	0.808	0.853	0.959
	JD+CM+IT	0.866	0.825	0.866	0.962

system using 10-fold cross-validation. For Guava, the classifier performs well on just Javadoc data. For Easymock, the classifier achieves 100% correct instances with just the Javadoc. This result is probably due to most deprecations being caused by the refactoring of a feature to use design patterns: Since Easymock always uses the same pattern (builder pattern), the terminology is the same. With more data, the accuracy of the classification decreases, probably due to added noise. For both Spring and Hibernate the classification accuracy is below 65% with only Javadoc data. With the addition of commit message data, the accuracy increases. In the case of Spring, when we add issue tracker data, the accuracy increases to 88%. However, in the case of Hibernate, the accuracy suffers slightly with issue tracker data, again probably due to added noise.

Mixed-system validation. We combine the data for all the APIs and treat this combined dataset as our singular vocabulary; Table II in the group named 'All' shows the results. With only Javadoc data, the classifier correctly classifies almost 76% of the cases. This might be due to all the Easymock instances, which the algorithm can easily classify with only Javadoc data. Adding commit message data improves the classification by almost 10%. Issue tracker data yields a minor improvement.

Cross-project validation. Given that the results are promising at project level, we tried to perform cross-project validation. However, results were consistently lower than 30% in the number of correctly classified instances. For example, in the case of Easymock, the method only reaches 24%. This result seems to indicate that there is project specific terminology that helps the machine learner to discern the different reasons. We also conducted cross-project classification for only one category (binary classification). We choose the addition of a new feature (NF) as our test category since we expect project-specific terms to be minimal. Although the number of correctly classified instances is 54%, this is still poor in comparison to project level classification. This result leads us to conclude that automated classification techniques work best at a project level due to project-specific terminology.

> **Finding 3**. An automatic classification approach can correctly classify more than 85% of deprecation reasons in three systems and 74% in the fourth. However, to achieve these results, data from commit messages and issue reports is often necessary and the classifier must be trained with project-specific instances.

VII. DISCUSSION

We discuss how our results lead to implications for future research and recommendation for practitioners.

A. Unmet developers' communication needs

Programming languages provide API producers with deprecation mechanisms to allow them to communicate with the API consumers, in a way that is recognized and rendered distinctively by popular IDEs and compilers. In Java, by marking a feature as deprecated, a compiler warning is thrown and the IDEs render the element as struck-through.

Recently, the Java language designers stated that the deprecation mechanism has been used not only for communicating about obsolete features but also for alternative purposes, which they labeled as "misuses" [8].[4] Previous work indeed found one case where the API producer has used the deprecation mechanism for an unorthodox purpose [7]. In this case, the JUnit API marked beta features as deprecated to warn consumers about these features' beta nature, which may lead to unanticipated future changes.

In our study, we uncover two additional cases in which API producers use deprecation for unorthodox purposes: (1) to indicate a temporary feature that is in place until a permanent solution can be found (as done by Guava and Hibernate), and (2) to indicate that a class only implements part of an interface and the unimplemented methods are essentially just stubs (as done in a widespread manner by Guava).

This finding raises the broader question of why API producers use the deprecation mechanism for purposes besides marking out obsolete features. It is reasonable to think that these are cases of communication needs for an API producer that are not being met by the Java language specification. For example, while the Guava API producers do try also to throw an exception to prevent the usage of specific features, they use the deprecation mechanism to issue compiler warnings. As an additional example, XWiki has introduced a workaround [33] where the usage of an '@Unstable' annotation in combination with an Eclipse plugin issues a warning in the IDE about the nature of the used feature.

[4]We consider these "misuses" from a more constructive perspective, that is, as evidence of developers' communication needs that are unmet by the current mechanisms and can be basis for future research and improvements.

This finding leads us to question if Java should invest in introducing a generic warning mechanism as a more flexible communication mechanism that would not lead to misinterpretations. This mechanism would allow API producers to throw a compiler warning for purposes other than deprecation, thus possibly addressing the producers' unmet communication needs. In this vein, languages such as PHP and Ruby have already set a precedent, where deprecation mechanisms and warning mechanisms are simultaneously present. By performing a similar study to the one we present in this paper in the API ecosystems of those two languages, researchers and the Java language designers can better understand whether there are benefits to having a generic warning mechanism and if indeed the cases of misuse would be minimized while fulfilling developers' communication needs.

B. Different evolution strategies

API evolution has been studied by researchers to understand *how* APIs evolve [34]. Researchers have also investigated the decision process behind evolving the API and introducing breaking changes in the API [18]. API evolution strategies are generally based on what features they change and how it affects the API consumer [6], [35], [36], [37], [4]. Significant work has also gone into alleviating the burden of dealing with API evolution [38], [39], [40], [41], [42].

In our study, we investigate the rationale used by API producers to evolve and render specific features as obsolete and introduce new features to replace them. We see that there are 12 reasons behind deprecating a feature. The frequency of usage of these rationales differs per API. For instance, we see in the case of Easymock most deprecations are due to the usage of design patterns, while for Spring, most changes are due to functional defects or newly introduced features.

We found initial evidence that by understanding the rationale behind the deprecation, we also better understand the evolution strategy adopted by an API and how it might affect a consumer. With Spring and Hibernate we see that a large number of deprecations are due to functional defect being present in the API. However, with Easymock and Guava other less important reasons such as redundancy of a method or refactoring to use a design pattern. Based on this, we can deduce that developers in Spring and Hibernate discover issues in features on a regular basis. On the other hand with Easymock and Guava, most of the changes are due to maintaining the API on a regular basis, without introducing several new features.

Further work can be conducted expanding on this line to see whether knowing the rationale behind evolution—thanks to the analysis of deprecation reasons—gives a better approximation of the evolution strategy of an API. This work would help informing tools to support practitioners keeping up with API evolution.

C. API documentation completeness

API documentation is vital in teaching consumers to adopt the API in a correct manner [19]. Incomplete documentation is a considerable obstacle to API consumers [43]. This is the primary reasons that API producers invest a lot of time in documenting their API in a correct and detailed manner [20].

Much work has gone into augmenting current API documentation to aid an API consumer and reduce the documentation burden on the API producer. Stylos *et al.* [44] have looked at augmenting API documentation by including API usage examples mined from open source repositories. Treude and Robillard [45] seek to improve API documentation with examples from community-driven documentation sources such as StackOverflow.

In the context of documentation of deprecated features, researchers have shown that recommending a replacement feature in the deprecation message is helpful to API consumers [21], [22]. In addition to that, informing the consumer about the rationale behind deprecation and the version in which the deprecated feature will be removed performs a vital role in the consumer's decision to react to deprecation [7].

We found overwhelming evidence that the Javadoc for deprecated features seldom mentions the reasons behind deprecating features. In fact, to uncover the rationale behind deprecation it is necessary to refer to the commit messages and to the issue tracker data. Conducting such a thorough search is error-prone and time consuming, thus impractical in a real-world scenario.

We show the different sources needed to infer the rationale behind deprecation. Research effort can be invested to be able to effectively retrieve the traceability links across all the different sources together, such that the justification for deprecation is evident and existing documentation enhanced.

D. Automating the classification of rationale

We investigate how accurately the rationale behind deprecation of a feature can be classified based on its Javadoc, the commit message that deprecates it and the issue tracker post that discusses its deprecation (if present). At a project level, we see that having this information allows us to classify the rationale behind deprecation accurately.

The automated classification relies heavily on project-specific terminology as is evidenced by the fact that cross-project classification yields poor results. Not only does project-specific terminology play a role, but also the specificity of technical terms for each rationale play a role too. For example, in the case of Easymock several deprecations took place due to the refactoring of a feature to use the builder pattern. In this case, the word "builder" is a specific case of refactoring to use a design pattern. If the automated classifier learns on other instances of refactoring to design pattern, such as the one from the Spring API, we see that it decreases the accuracy for the cases in Easymock.

Despite the specific circumstances under which an automated approach can work, the classifier performs promisingly at a project level. API producers can run such an approach on their documentation to automatically categorize the rationale of a deprecated feature and use this categorization to augment their existing documentation.

We see that there is a need for more than just the Javadoc to classify the rationale of a deprecated feature. This result puts into focus the need for the creation of a complete information pipeline that stitches together the Javadoc, commit message, and issues regarding a deprecated feature. This approach would go a long way in aiding automating the classification of the rationale of a deprecated feature.

Further research needs to be conducted in the area of automating the classification of the rationale behind the deprecated feature. We show that a machine learning approach can work and provide an initial baseline for future comparison. More research is needed to investigate whether and how the poor performances in cross-project classification can be tackled, for example by considering further features and other classification techniques such as deep learning.

VIII. Related work

We describe work in the related areas of API documentation needs and improving documentation.

Studies on API evolution. Robbes *et al.* [46] analyze the impact of deprecation of an API feature on the SmallTalk ecosystem. They find that while the number of API consumers affected is high, minimal reaction to deprecation takes place. Sawant *et al.* [47], [6] mine 25,357 Java-based API consumers from GitHub and a further 150,326 Maven central based JAR files to see how many consumers are affected by deprecation and their reactions. They observed that over 10% of deprecated methods affect consumers, but consumers do not react. In contrast to this, we look at the reasons behind deprecation of the API from the API producer perspective.

Hou and Yao [48] investigate the intent behind API evolution by studying release notes. They found that API features were deprecated due to conformance to API naming conventions, naming improvements, simplification of the API and replacement of functionality. Sawant *et al.* [7] interview 17 API producers as to why they deprecate features and catalog seven reasons behind deprecation. In this study, we analyze documentation at a fine-grain (Javadoc, issue tracker and commit messages) level to understand the reason behind deprecation. This analysis leads us to uncover 12 rationales behind deprecation. Moreover, we evaluate how well an automated technique can classify the reason for a deprecation.

Studies on API documentation needs. Robillard and Deline show that API documentation is a vital resource for developers who want to adopt a new API [19]. Myers and Stylos concur with this view and provide evidence that API documentation plays a significant role in making it usable [49]. Maalej and Robillard show that API reference documentation should complement the API by providing information that is not obvious from the API syntax [50].

Uddin and Robillard uncover that consumers find it much harder to understand the API producer's intentions due to inadequate documentation [43]. Monperrus *et al.* analyzed API Javadoc to see what was being talked about and in what cases there was an information shortfall [51]. They state that a deprecation tag in Javadoc with no rationale or conditions is an anti-pattern. Brito *et al.* find that in over 60% of the cases the replacement for a deprecated feature is specified [21], [22]. During Sawant *et al.*'s investigation into the deprecation mechanism, they found that consumers miss is the rationale behind the deprecation itself [7].

In this study, we find that deprecated API documentation often does not document the reason behind the deprecation, despite this being important for consumers.

Studies on improving API documentation. One of the primary challenges with producing high-quality API documentation is the large amount of time and effort that goes into creating the documentation [20].

Dekel and Herbsleb improve API documentation by highlighting specific directives that are present in the documentation so that the consumer is made explicitly aware of particular conditions that he has to be aware of [52]. Researchers have investigated ways to improve existing documentation by augmenting it with examples mined, for instance, from source code repositories [44], [53], [54], [55] and StackOverflow [45].

Subramanian *et al.* create a tool to called Baker that links source code examples to API documentation [56]. Baker can do this in a real-time manner thus always keeping the usage examples in the API documentation fresh. Dagenais and Robillard look to recover the traceability links between APIs and their learning resources [57]. This study aims to produce a comprehensive set of documentation that is mined from a variety of sources such as developer blogs, StackOverflow and mailing lists, all in one place.

We found that the rationale behind deprecation can be automatically classified but, more than one source of information is required. This understanding can aid in providing consumers with the rationale behind deprecation.

IX. Conclusion

We have presented an explorative study we conducted to uncover the rationale behind the deprecation of an API feature. We manually analyzed over 1,100 document artifacts relating to 374 from 4 mainstream Java APIs deprecated features. This analysis led to the creation of a taxonomy comprising 12 reasons for deprecation. We observe that there are several cases of deprecation being used in an unexpected, unorthodox manner, thus hinting at currently unmet communication needs. Finally, we found that an automated approach to classifying deprecation reasons can reach promising accuracy, but only when it is trained on instances from the same project. We discussed the results and their implications concerning unmet communication needs, API evolution strategies, API documentation completeness, and future work.

Acknowledgements

The authors would like to thank the anonymous reviewers for their thorough feedback. A. Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

REFERENCES

[1] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*, vol. 7. University of California, Irvine Doctoral dissertation, 2000.

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

[3] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[4] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "API change and fault proneness: a threat to the success of android apps," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pp. 477–487, ACM, 2013.

[5] J. R. Rose, "How and when to deprecate API." http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase11-419415.html#7122-jdk-1.1-doc-oth-JPR, 1996. last accessed May 2017.

[6] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of clients of 4+1 popular Java APIs and the JDK," *Empirical Software Engineering*, pp. 1–40, 2017.

[7] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli, "Understanding developers' needs on deprecation as a language feature," in *Proceedings of the 40th International Conference On Software Engineering*, ICSE 2018, pp. 181–190, IEEE/ACM, 2018.

[8] S. Marks, "JEP 277: Enhanced Deprecation." http://openjdk.java.net/jeps/277, 2014–2017. last accessed Aug 2017.

[9] "Spring API repository." https://github.com/spring-projects/spring-framework. accessed on April 2018.

[10] "Hibernate API repository." https://github.com/hibernate/hibernate-orm. accessed on April 2018.

[11] "Guava API repository." https://github.com/google/guava. accessed on April 2018.

[12] "Easymock API repository." https://github.com/easymock/easymock. accessed on April 2018.

[13] N. M. Nasrabadi, "Pattern recognition and machine learning," *Journal of electronic imaging*, vol. 16, no. 4, p. 049901, 2007.

[14] M. Stone, "Cross-validatory choice and assessment of statistical predictions," *Journal of the royal statistical society. Series B (Methodological)*, pp. 111–147, 1974.

[15] "Javadoc for spring deprecation." https://www.javadoc.io/doc/org.springframework/spring-core/4.2.3.RELEASE, 2018. last accessed April 2018.

[16] "Commit that deprecates feature in spring." https://github.com/spring-projects/spring-framework/commit/e27df06f919a1f1ef53b0571e1a15dfc9e2f707f, 2018. last accessed April 2018.

[17] "Spring issue tracker post." https://jira.spring.io/browse/SPR-13621, 2018. last accessed April 2018.

[18] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "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*, pp. 109–120, ACM, 2016.

[19] M. P. Robillard, "What makes APIs hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, 2009.

[20] B. Dagenais and M. P. Robillard, "Creating and evolving developer documentation: understanding the decisions of open source contributors," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 127–136, ACM, 2010.

[21] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "Do developers deprecate APIs with replacement messages? a large-scale analysis on Java systems," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1, pp. 360–369, IEEE, 2016.

[22] G. Brito, A. Hora, M. T. Valente, and R. Robbes, "On the use of replacement messages in API deprecation: An empirical study," *Journal of Systems and Software*, vol. 137, pp. 306–321, 2018.

[23] "Tiobe index." http://www.tiobe.com/tiobe_index. last accessed on Apr 2018.

[24] A. A. Sawant and A. Bacchelli, "A dataset for API usage," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR 2015, pp. 506–509, IEEE Press, 2015.

[25] A. A. Sawant and A. Bacchelli, "fine-GRAPE: fine-Grained APi usage Extractor–An approach and dataset to investigate API usage," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1348–1371, 2017.

[26] "Eclipse jdt ast parser." http://help.eclipse.org/kepler/ntopic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTParser.html, 2018. last accessed April 2018.

[27] M. Triola, *Elementary Statistics*. Addison-Wesley, 10th ed., 2006.

[28] A. Bacchelli, M. Lanza, and V. Humpa, "RTFM (Read The Factual Mails) –augmenting program comprehension with REmail," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, CSMR 2011, pp. 15–24, IEEE, 2011.

[29] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. van Deursen, "Communication in open source software development mailing lists," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR 2013, pp. 277–286, IEEE, 2013.

[30] W. Lidwell, K. Holden, and J. Butler, *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub, 2010.

[31] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[32] "Replication package." https://www.dropbox.com/s/veef9j517okmf2d/replication-package.zip?dl=0, 2018. last accessed April 2018.

[33] "Xwiki unstable annotation." http://dev.xwiki.org/xwiki/bin/view/Community/DevelopmentPractices#H40UnstableAnnotation, 2018. last accessed April 2018.

[34] D. Dig and R. Johnson, "How do APIs evolve? a story of refactoring," *Journal of Software: Evolution and Process*, vol. 18, no. 2, pp. 83–107, 2006.

[35] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for api evolution," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 599–602, IEEE Computer Society, 2009.

[36] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, p. 19, 2011.

[37] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," in *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pp. 138–147, IEEE, 2017.

[38] Z. Xing and E. Stroulia, "API-evolution support with Diff-CatchUp," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007.

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

[40] J. Henkel and A. Diwan, "CatchUp!: capturing and replaying refactorings to support API evolution," in *Proceedings of the 27th international conference on Software engineering*, pp. 274–283, ACM, 2005.

[41] P. Kapur, B. Cossette, and R. J. Walker, "Refactoring references for library migration," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA 2010, pp. 726–738, ACM, 2010.

[42] I. Şavga and M. Rudolf, "Refactoring-based support for binary compatibility in evolving frameworks," in *Proceedings of the 6th international conference on Generative programming and component engineering*, pp. 175–184, ACM, 2007.

[43] G. Uddin and M. P. Robillard, "How API documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[44] J. Stylos, B. A. Myers, and Z. Yang, "Jadeite: improving API documentation using usage information," in *CHI'09 Extended Abstracts on Human Factors in Computing Systems*, pp. 4429–4434, ACM, 2009.

[45] C. Treude and M. P. Robillard, "Augmenting API documentation with insights from stack overflow," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 392–403, IEEE, 2016.

[46] R. Robbes, M. Lungu, and D. Röthlisberger, "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*, p. 56, ACM, 2012.

[47] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs," in *Proceedings*

of the 32nd International Conference on Software Maintenance and Evolution, ICSME 2016, pp. 400–410, IEEE, 2016.

[48] D. Hou and X. Yao, "Exploring the intent behind API evolution: A case study," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pp. 131–140, IEEE, 2011.

[49] B. A. Myers and J. Stylos, "Improving API usability," *Communications of the ACM*, vol. 59, no. 6, pp. 62–69, 2016.

[50] W. Maalej and M. P. Robillard, "Patterns of knowledge in API reference documentation," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013.

[51] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? an empirical study on the directives of API documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2012.

[52] U. Dekel and J. D. Herbsleb, "Improving API documentation usability with knowledge pushing," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pp. 320–330, IEEE, 2009.

[53] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente, "Documenting APIs with examples: Lessons learned with the APIMiner platform," in *20th Working Conference on Reverse Engineering*, WCRE 2013, pp. 401–408, IEEE, 2013.

[54] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui, "Mining multi-level API usage patterns," in *22nd International Conference on Software Analysis, Evolution and Reengineering*, SANER 2015, pp. 23–32, IEEE, 2015.

[55] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?," in *37th International Conference on Software Engineering*, vol. 1 of *ICSE 2015*, pp. 880–890, ACM/IEEE, 2015.

[56] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 643–652, ACM, 2014.

[57] B. Dagenais and M. P. Robillard, "Recovering traceability links between an API and its learning resources," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE 2012, pp. 47–57, IEEE, 2012.