DepMiner: Automatic Recommendation of Transformation Rules for Method Deprecation

Oleksandr Zaitsev, Stéphane Ducasse, Nicolas Anquetil, Arnaud Thiefaine

DepMiner: Automatic Recommendation of Transformation Rules for Method Deprecation*

Oleksandr Zaitsev[1,2], Stéphane Ducasse[2], Nicolas Anquetil[2], and Arnaud Thiefaine[1]

[1] Arolla, Paris, France
{oleksandr.zaitsev,arnaud.thiefaine}@arolla.fr
https://www.arolla.fr
[2] Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL
{stephane.ducasse,nicolas.anquetil}@inria.fr
https://www.inria.fr

Abstract. Software applications often depend on external libraries and must be updated when one of those libraries releases a new version. To make this process easier, library developers try to reduce the negative effect of breaking changes by deprecating the API elements before removing them and suggesting replacements to the clients. Modern programming languages and IDEs provide powerful tools for deprecations that can reference the replacement or incorporate the rules written by library developers and use them to automatically update the client code. However, in practice library developers often miss the deprecation opportunities and fail to document the deprecations. In this work, we propose to help library developers support their clients with better deprecations. We rely on the transforming deprecations offered by Pharo and use data mining to detect the missing deprecation opportunities and generate the transformation rules. We implemented our approach for Pharo in a prototype tool called *DepMiner*. We have applied our tool to five open-source projects and proposed the generated deprecations to core developers of those projects. 63 recommended deprecations were accepted as pull requests.

1 Introduction

Most modern software depends on multiple external libraries [3]. Each one of those libraries is a separate project that is managed by its own team of developers. Like any other software, libraries evolve from one version to another, parts of their Application Programming Interfaces (API) are changed (classes, methods, or fields get renamed, deleted, or moved around, new functionalities are introduced, etc. [12]). As a result, developers depending on those libraries must either update their code or continue having outdated and no longer maintained dependencies.

* This work was financed by the Arolla software company.

Deprecation is a common practice for supporting library evolution by notifying client systems about the changed or removed features and helping them adapt to the new API. Instead of removing a feature in release n, it is marked as *deprecated* (*"to be removed"*) and only actually removed in a later release $n+k$. Client systems that call a deprecated feature receive a deprecation warning which gives developers time to update their code.

It is a good practice for library developers to supply deprecations with code comments or warning messages that suggest a replacement for an obsolete item. For example, *"Method a() is deprecated, use b() instead"*. To support this practice, Java provides the @Deprecated annotation as well as the @deprecated Javadoc tag that can mark a method or class as deprecated while the @link or @see tags can reference the correct replacement in the source code [18]. Pharo[3] has a powerful deprecation engine called *Deprewriter* [13]. It allows library developers to add transformation rules to their method deprecations specifying the replacements. When a deprecated method is invoked, Deprewriter identifies the call-site at run-time and uses the rule to update the client code without interrupting its execution [28, 31, 32].

However, developers of real projects do not always follow good deprecation practices. They tend to introduce breaking changes to the APIs by renaming or removing certain classes, methods, or fields without deprecating them first [5, 40, 41]. Also, several large-scale studies of popular software projects have revealed that the proportion of deprecations that contain a helpful replacement message (in a form of comment, string, annotation, etc.) is only 66.7% for Java, 77.8% for C# [7], and 67% for JavaScript [23].

Multiple approaches have been proposed to support client developers by automatically inferring missing messages. Dig *et al.,* [11] proposed to detect refactorings between the two versions of the library based of the textual similarity of source code and the similarity of references. Schaffer et al. [34], Dagenais *et al.,* [9], and Hora et al. [15] mined frequent method call replacements in the commit history of a library to learn how it adapted to its own changes. Pandita *et al.,* [24] and Alrubaye *et al.,* [1] used a similar technique to help client developers replace dependencies to one library with dependencies to another one. Teyton et al. [38] and Brito *et al.,* [7] recommend replacements by learning from client systems that have already updated their code.

In this work, we look at the problem from the perspective of library developers. We propose an approach and a tool called *DepMiner* to help them identify breaking changes before the release, understand when and by whom they were introduced, and find the potential replacements that could be suggested to the clients. We generate the recommendations in the form of transformation rules that can be used by Pharo's Deprewriter. Inspired by the existing approaches that were proposed to support the client developers [9, 15, 34], our approach is based on the frequent method call analysis. The main differences are: (a) we recommend replacements *before* the release which makes it impossible to rely

[3] Pharo is a dynamically-typed object oriented programming language and IDE: https://pharo.org/

on the clients that were already updated; (b) Pharo is a dynamically-typed language, which means that we can not rely of type information when analyzing method call replacements; (c) Pharo has no explicit method visibility (*i.e.,* public or private specifiers), which makes it hard to define the API. DepMiner can be extended to work with other increasingly popular dynamically-typed languages such as JavaScript, Python, Ruby, etc.

To evaluate our approach, we applied *DepMiner* to 5 diverse open-source projects that were implemented in Pharo and suggested its recommendations to the developers of those projects. 138 recommendations generated by our tool were confirmed by developers. 63 generated deprecations were accepted as pull requests into the projects.

The rest of this paper is structured as follows. In Section 2, we briefly describe the Deprewriter tool in Pharo. In Section 3, we discuss the problem of supporting library developers and the challenges that arise when dealing with this problem in Pharo. In Section 4, we describe our proposed approach and explain the underlying data mining algorithm. In Section 5, we evaluate our approach by comparing the generated transformation rules to the ones that are already present in the source code and by performing a developer study. Finally, in Section 6, we explain the limitations of our approach.

2 Deprewriter: Transforming Deprecations in Pharo

Pharo allows developers to enrich deprecations with code transformation rules [13]. If a client system invokes the deprecated method, its source code is automatically fixed during execution to call the replacement:

```
1 isSpecial
2   self
3     deprecated: 'Renamed to #needsFullDefinition'
4     transformWith: '`@rec isSpecial'
5                 -> '`@rec needsFullDefinition'.
6   ↑ self needsFullDefinition
```

Lines 2-5 of the code above demonstrate the syntax of transforming deprecations in Pharo: method isSpecial (name in the first line) is deprecated with a message for the user 'Renamed to #needsFullDefinition' and a transformation rule that replaces method calls to isSpecial with calls to needsFullDefinition. The transformation rule consists of two parts: the *antecedent*, matches the method call that should be replaced ; the *consequent*, defines the replacement. '@rec and '@arg are rewriting variables matching respectively the receiver of the invocation and its argument.

Transforming deprecations are a powerful technique that can save time for client developers. Because now, instead of reading the source code of a library and looking for the correct replacement, they only need to run the unit tests of their project to have their code fixed automatically.

3 Why Do We Need to Support Library Developers?

To understand the propagation of transforming deprecations, we have extracted all deprecated methods from v8.0.0 of the Pharo Project[4]. We discovered that out of 470 valid deprecations in Pharo 8, 190 deprecations (40%) do not contain transformation rules. Out of those 190 non-transforming deprecations, 41 (22%) can have a simple transformation rule that can be generated automatically; 85 deprecations(45%) require developers with project expertise to provide extra information (additional argument, default value, etc.) and write a rule manually; the other 64 deprecations (34%) are complex and can not be expressed using the language for transformation rules that is used in Pharo. This indicates that developers don't always write transformation rules for their deprecations. Similar trends can be observed in other programming languages. For example, according to large-scale studies of software systems, the proportion of deprecations that do not contain a helpful replacement message (in a form of comment, string, annotation, etc.) is 33% for Java, 22% for C# [7], and 33% for JavaScript [23].

Those observations demonstrate the need for an automated tool to recommend the replacement messages for method deprecations. There are two main challenges when implementing such a tool for Pharo:

Challenge 1: Absence of method visibility. Languages like Java and C++ have public, private, and protected keywords that can help identify methods that are meant to be used by clients and can be considered as part of API. However, in languages like Python or Pharo all methods are public [35]. Sometimes Python developers use underscores at the beginning of method names to mark them as "private" but it is more of a good practice than a strict requirement and this practice is not always followed. Although Pharo developers often adopt different practices to mark methods as private, none of those practices are universally adopted by the Pharo community.

Challenge 2: Absence of static type information. Pharo is a dynamically-typed programming language [21, 37]. The absence of static type information complicates the task of identifying correct method mappings between the old and the new version because it is not easy to map method calls in the source code to the actual method implementations. We also do not know the argument types. This has an important implication that we can get a combinatorial explosion when analysing a sequence of messages. The research community has proposed type inference for dynamically-typed languages [14, 25–27, 36, 37] or use dynamic type information collected by the Virtual Machine to get concrete types [22]. But such type inferencers often do not cover the full language [37] or are not applicable

[4] Pharo is a programming language and an IDE written entirely in itself. This can be a source of confusion. In this section, we discuss the analysis of how transforming deprecations, introduced into Pharo (an open-source project with over 150 contributors), were used by its developers to deprecate methods in other parts of the same project. In other words, we study how Pharo developers use the rewriting functionality of Pharo to deprecate methods in Pharo.

to large code bases [36]. In this paper, we do not perform type inference and consider that the type information is missing, which constitutes a challenge for the data mining algorithm.

4 DepMiner: Recommending Transforming Deprecations by Mining the Commit History

We propose to assist library developers in the task of detecting the missed deprecation opportunities and finding proper replacements for the deprecated methods by mining frequent method call replacements from the commit history. Our approach consists of four steps:

1. Identifying the methods that belong to the old API and the new API of the project.
2. Collecting the database of method call replacements from the commit history.
3. Mining frequent method call replacements using the A-Priori algorithm for frequent itemsets mining.
4. Generating deprecations with transformation rules.

Identifying Methods of the Old and the New API. As we have discussed in Section 3 (Challenge 1), all methods in Pharo are public in the sense that clients can access them, however not all of those methods are meant to be used. To deal with this challenge, we define several categories of methods in Pharo that can be considered private: (a) *initialize methods* — they act like constructors in Pharo; (b) *unit test methods*, including setUp, tearDown, and methods of mock classes; (c) *example methods*; (d) *baseline methods* — define project structure and dependencies; (e) *help methods* — a form of documentation; (f) *methods in "private" protocols* — any protocol that includes the word "private". We implemented heuristics to infer the visibility of methods in Pharo and released them in a public repository[5]. With that information, we define two sets of methods: API_{old} — "public" methods in the old version, and API_{new} — "public" methods in the new version.

Collecting Method Changes from the Commit History. Given the history of commits between the old version and the new version, we extract method changes from every commit. A *method change* describes how one specific method was changed by a given commit. For each method change, we parse the source code of a method before and after it was changed and extract a set of method calls from each version. As a difference between those two sets, we get the sets of deleted and added method calls for every method change. We remove all deleted method calls that were not part of API_{old} and all added method calls that are not part of API_{new}. Because Pharo is a dynamically-typed language, we do not know which implementation of a method will be executed (see Section 3,

[5] https://anonymous.4open.science/r/VisibilityDeductor-EF86

Challenge 2). As a result, many method calls in our dataset are false positives, because they call the method with the same name as the one in API_{old} and API_{new}, but in reality that method is called from a different library (*e.g.,* methods such as add() or asString() can be implemented by different classes). To deal with this problem and reduce the noise in our data, we choose a threshold K and remove all method changes that have more than K added or more than K deleted method calls (by experimenting with different values of K, for this project we selected $K = 3$). We also removed all calls to highly polymorphic methods such as = and printOn:. Finally, we removed all method changes for which either the set of deleted or the the set of added calls was empty.

Mining Frequent Method Call Replacements. After collecting the dataset of method changes from the commit history, we apply a data mining algorithm to find all frequent subsets of method call replacements. This technique was inspired by the work of Schäfer *et al.,* [34], Hora *et al.,* [15], and Dagenais *et al.,* [10] who proposed similar history-based approaches to support the clients of Java libraries. In terms of market basket analysis, each method change can be represented as a transaction or an itemset. To do that, we merge the sets of added and deleted calls in a method into a single set. For example, {deleted(isEmpty), deleted(not), deleted(add), added(new), added(isNotEmpty) }. By selecting a minimum support threshold min_{sup}, we use a data mining algorithm such as A-Priori, Eclat, or FP-Growth to find all combinations of method calls that appear in different method changes at least min_{sup} times (frequent itemsets). Then we construct association rules by putting all deleted method calls into the *antecedent* (left hand side) and all added method calls into the *consequent* (right hand side). We remove the rules with empty antecedent or empty consequent. For each association rule $I \rightarrow J$, we calculate its confidence — the probability that a set of deleted calls I appear jointly with added calls J and not with something else:

$$confidence(I \rightarrow J) = \frac{support(I \cup J)}{support(I)}$$

We select a confidence threshold min_{conf} and filter out all association rules that do not reach this threshold. The current implementation of Deprewriter supports only one-to-one (one antecedent, one consequent) and one-to-many rules (one antecedent, several consequents) — the ones that define the replacement of *one* method call (the method from the old API that is being deprecated) with one or more method calls. Therefore, we remove all many-to-one and many-to-many rules from the collection of association rules.

Generating Recommendations. Based on two sets of methods, API_{old} and API_{new}, and the collection of association rules $Assoc$, mined from the method changes, we can now provide recommendations to library developers:

1. **Proposed deprecations** — we find all methods of the old API that were deleted without being deprecated (every method m such that $m \in API_{old}$ and $m \notin API_{new}$). If we can find at least one association rule in $Assoc$

that defines the replacement for a given method m, then we recommend to reintroduce m into the new version of a project with deprecation and a transformation rule if it can be generated.
2. **Transformation rules for existing deprecations** — first we identify all manually deprecated methods from API_{new} that do not contain a transformation rule. For every such method m, if we can find at least one association rule $a \in Assoc$ that defines the replacement for m, we recommend to insert a transformation rule into the deprecation of m either automatically (in case the transformation rule can be inferred from a, as we will discuss below) or semi-automatically (in case we can only show the association rule a to developers and ask them to write a transformation rule manually).

Transformation rules of the form ‘@rec selector1: ‘@arg → ‘@rec selector2: ‘@arg are generated automatically from the association rule such as {selector1:} → {selector2:} only if:

- association rule is one-to-one (one deleted method call replaced with one added method call),
- deleted and added method calls have the same number of arguments,
- deleted and added method calls are defined in the same class of the new version of the project (and therefore can have the same receiver).

If one of those conditions is not satisfied, the transformation rule can not be generated and must be written manually by a developer. In those cases, we only show to developers the association rule together with the examples of method changes in which those rules appeared and ask them to write a transformation rule manually.

5 Evaluation

We have implemented our approach in a prototype tool for Pharo called *DepMiner*.[6] Our implementation is based on the A-Priori algorithm for mining frequent itemsets. We have applied *DepMiner* to several open-source projects and asked core developers of those projects to review the recommendations produced by DepMiner.

5.1 Evaluation Setup

Selected projects. For this study, we have selected five open-source projects:

- **Pharo**[7] — a large and mature system with more than 150 contributors, containing the language core, the IDE, and various libraries.

[6] https://anonymous.4open.science/r/DepMiner-0D5B

[7] Pharo is an open-source project written in Pharo programming language (see footnote in Section 3), https://github.com/pharo-project/pharo

– **Moose Core**[8] — Moose is a large platform for data and source code analysis. It consists of multiple repositories, we focus only on the core repository of Moose.
– **Famix**[9] — generic library that provides an abstract representation of source code in multiple programming languages. Famix is part of the Moose project.
– **Pillar**[10] — a markup syntax and tool-suite to generate documentation, books, websites and slides.
– **DataFrame**[11] — a specialized collection for data analysis that implements a rich API for querying and transforming datasets.

We selected such projects because: (1) we were able to interview and ask maintainers to validate the proposed deprecations, (2) the projects evolved over several versions and are still under active development, (3) we wanted to compare the performance of *DepMiner* on the projects with different maturity and complexity levels.

For this study, we define three types of projects:

– **Tool** — a project that is designed for the end users (in the experiment: Moose, Pillar). For example, a text editor, a website, or a smartphone app. In many cases, APIs of those projects do not change that much (e.g. poorly named method that is not called by external projects might not be renamed) and when they do change, deprecations are rarely introduced.
– **Library** — a project that is supposed to be used as dependency by other projects (in the experiment: Famix, DataFrame). For example, a data structure, a networking library, or a library for numeric computations. Projects of this type must have a stable API and good versioning. They are most likely to introduce deprecations.
– **SDK** — a special type of project that describes Pharo. It is a combination of multiple different projects. Pharo has many users and even small changes to API can break software that is built with Pharo. This means that deprecations are very important for this type of projects.

Table 1. Selected software projects

Project	Type	Old version	New version	Commits
Pharo	SDK	v8.0.0	af41f85	3,465
Moose Core	Tool	v7.0.0	v8.0.0	1,519
Famix	Library	a5c90ff	v1.0.1	948
Pillar	Tool	v8.0.0	v8.0.12	508
DataFrame	Library	v1.0	v2.0	225

[8] https://github.com/moosetechnology/Moose
[9] https://github.com/moosetechnology/Famix
[10] https://github.com/pillar-markup/pillar
[11] https://github.com/PolyMathOrg/DataFrame

Two versions of each project. To mine the repetitive changes and propose deprecations, we must first select two versions of each project: the *new version* for which we will propose the deprecations and the *old version* to which we compare the new version of the project. All patterns will then be mined from the slice of the commit history between those two versions. Table 1 lists the two versions of each project that we have loaded as well as the number of commits between those two versions.

Mining frequent method call replacements. We used DepMiner to mine frequent method call replacements from the histories of those projects and recommend deprecations with transformation rules. In Table 2, we report the minimum support and minimum confidence thresholds that were used to initialize the A-Priori algorithm. The minimum support threshold for each project was selected experimentally. We started with a large support threshold = 15 (meaning that we are only interested in replacements that happened at least 15 times) and decreased it until the number of generated recommendation seemed sufficiently large. The confidence threshold was selected based on the number of method changes and the number of rules that *DepMiner* generated for a selected support value. For Pharo and Famix we can expect rules with confidence of at least 0.4. For other projects, we limit confidence to 0.1. In the last two columns of Table 2, we present the number of association rules (frequent method call replacements) that were found by *DepMiner* given the settings discussed above, and the number of rules that can automatically generate the transformation rules of the form '@rec deletedSelector: '@arg → '@rec addedSelector: '@arg (only one-to-one rules where deleted and added selectors have the same number of arguments).

Table 2. Association rules mined from the commit history

Project	Min sup.	Min conf.	Assoc. rules	Transforming
Pharo	5	0.4	377	152
Moose Core	2	0.1	88	40
Famix	4	0.4	149	60
Pillar	2	0.1	49	16
DataFrame	5	0.1	22	7

5.2 Evaluation by Project Developers

We have performed a first developer study of our tool involving the core developers from each project listed in Section 5.1. We asked 4 developers with different areas of expertise to validate the recommendations generated for Pharo and one developer for each of the other 4 projects (two developers had expertise in two projects each so in total, our study involved 6 developers).

To each developer, we showed the pretrained *DepMiner* tool with recommended methods to deprecate and recommended transformation rules to insert

into the existing deprecations. The developers had to select the changes which, in their opinion, should be merged into the project. *DepMiner* allows its users to browse multiple version of the project as well as the commits history. Each recommendation is supported by the list of commits in which the given method call replacement has appeared. This allowed developers who participated in our study to make an informed decision. For the Pharo project we considered recommendation accepted if it was accepted by at least one developer (because different developers might know different parts of the whole system).

Proposed deprecations. Table 3 reports the numbers of deprecations that were recommended to developers for each project (column *Recommended*), the number of those recommendations that were accepted (column *Accepted*), and the number of those accepted recommendations that contain an automatically generated transformation rule (column *Transforming*). Each recommended deprecation is a method that was deleted from the project without being deprecated first and which we propose to re-introduce with the recommended replacement.

Table 3. Number of recommended deprecations accepted by developers

Project	**Recommended**	**Accepted**	**Transforming**
Pharo	113	61	56
Moose Core	33	1	1
Famix	87	68	28
Pillar	1	0	0
DataFrame	11	4	4

One can see that *DepMiner* was very effective in generating recommendations for Pharo (113 recommendations, 61 accepted), Famix (87 recommendations, 68 accepted), and DataFrame library (11 recommendations, 4 accepted) but rather ineffective on Moose Core (33 recommendations, 1 accepted) and Pillar (1 recommendation, 0 accepted).

The different performance on those projects can not be explained by their size. For example, the DataFrame project is the smallest one in our list, but out of 11 deprecations generated by *DepMiner*, 4 deprecations were accepted. On the other hand, for the Pillar project, which is 10 times larger in terms of the number of methods, only 1 deprecation was generated and it was not accepted. Further study is required to explain the differences between DataFrame and Pillar, but we can speculate that bad performance on Pillar is caused by the low variability of API. Methods of DataFrame were often renamed, removed, or reorganised, which was reflected in test cases and picked up by *DepMiner*. On the other hand, the API of Pillar remained stable even though new functionality was added to it and many bugs were fixed.

Missing rules. The other type of recommendations that we showed to developers were transformation rules for existing non-transforming deprecations. Table 4 re-

ports the number of existing deprecations that are missing a transformation rule, the number of recommendations that DepMiner managed to generate for those deprecations, and finally the number of recommendations that were accepted by developers.

Table 4. Number of missing rules accepted by developers

Project	Missing	Recommended	Accepted
Pharo	189	6	2
Moose Core	2	0	0
Famix	27	2	2
Pillar	0	0	0
DataFrame	0	0	0

Deprecations that are missing the transformation rule (the non-transforming deprecations) represent either complicated cases for which the transformation rule can not be provided (e.g. method was deleted without replacement) or simple cases for which developers forgot to write a rule. As we mentioned in Section 3, for 22% of non-transforming deprecations the transformation rule could be generated automatically (assuming that we know the correct replacement), the other 78% of non-transforming deprecations require a complex rule that must be written manually. *DepMiner* proposed 6 transformation rules for existing non-transforming deprecations in Pharo (2 of which were accepted) as well as 2 transformation rules for Famix (both were accepted).

Pull Requests. Out of 5 projects that we used in our study, only Pharo Project was preparing an upcoming release. We applied DepMiner to the latest commit of the development version of Pharo and this allowed us to submit the recommendations that were confirmed by developers as pull requests. All 61 confirmed deprecations and 2 confirmed transformation rules for existing deprecations were merged into the v9.0.0 release of of Pharo.

6 Limitations of Our Approach

Unused/untested methods. Our approach is based on library's usage of its own API. This means that we can not infer anything for methods that are not used by the library itself but only intended for clients. Test cases play the role of clients of the library's API, so for the methods that are well tested, we can have enough input to identify the correct replacement for them. But if a method is not used by the library and not covered by test, then its deletion or renaming will not be reflected anywhere else in the source code.

Reflective operations. Modern programming languages offer reflective operations [8, 29]. They allow developers to invoke methods programmatically and

create generic and powerful tools. However, since some methods can be invoked reflectively for example passing the name of the method to be invoked in a variable, when a different argument is passed to a reflective call, our tool cannot identify such change. Most static analysers ignore such case [4].

Unordered set of method calls. Our tool is based on mining method call replacement by comparing the set of calls that were deleted from the source code of a modified method to the set of calls that were added to it. We do not take into account the order of method calls, the distance between them or how they are composed: `a().b()` or `a(b())`. This is a limitation of our approach because: (1) sometimes deleted and added method calls are located far away in source code and not related to each other; (2) if one method call is being replaced with two or more method calls, we do not know how they should be composed.

7 Related Work

Breaking changes. Breaking changes are the code modifications in all API elements that break backward compatibility [12]. In their large-scale analysis of Java libraries, Xavier *et al.,* [40] discovered that 28% of API changes break backward compatibility however, on the median, only 2.54% of clients are impacted by them. In their follow-up study, Xavier *et al.,* [41] performed a survey of developers to understand why they introduce breaking changes into their projects. They identified five reasons: library simplification, refactoring, bug fixes, dependency changes, project policy. This study was later extended by Brito *et al.,* [5] who reported that 47% of breaking changes are due to refactorings. These results can be complemented by the previous study by Dig and Johnson [12] who analysed breaking changes in five Java systems and discovered that 81-100% of them are caused by refactorings. Those findings are important for our study because changes that are introduced by refactorings (e.g. renaming, replacement, spitting, etc.) ofter require simple repetitive fixes in the client code that can be expressed with transformation rules.

Impact of deprecations. Robbes *et al.,* [30] studied the impact of API changes, and in particular deprecations, on Pharo and Squeak ecosystems. They report that the majority of client systems are updated over a day, but in some cases the update takes longer and is performed only partially. Sawant *et al.,* [33] report similar results for Java. Hora *et al.,* [16, 17] complemented the previous studies by analysing the impact of API evolution on Pharo ecosystem, but focusing only on those changes which are not related to deprecations. They claim that API changes have large impact on the ecosystem and most of the changes that they found can be implemented as rules in static analysis tools. Several authors have also explored the effectiveness of deprecation messages. Large-scale empirical studies of software written in Java and C# [6,7] as well as JavaScript [23] revealed that 22-33% of deprecations in those languages are not supported by replacement messages. In their study of Pharo ecosystem, Robbes *et al.,* [30] also showed

that almost 50% of deprecation messages do not help to identify the correct replacement.

Library migration and update. Mining the commit history to find a mapping between two versions of API is not a new idea. Similar problems were targeted in the context of library update (updating client system to depend on a new version of the external library) and library migration (replacing the dependency on one library with that on a different library). First approaches in this area were based on static code analysis and the textual similarity of method signatures [19, 43]. Wu *et al.,* [39] proposed an approach that combined call dependency and text similarity analyses. Schäfer *et al.,* [34] proposed to mine library update rules from already updated client systems. The following studies analysed the commit history between the two versions of a library [10, 20] and proposed changes that should be applied to client systems. Teyton *et al.,* [38] used data mining and propose rules for library migration by learning from the commit histories of already migrated clients. Hora *et al.,* [15] proposed a similar approach to find method mappings between different releases of the same library. Pandita *et al.,* [24] approached the problem of library migration by analysing textual similarity of documentation from different libraries. Alrubaye *et al.,* [2] proposed a novel machine learning approach that inferred the mapping between the API elements of two different library by extracting various features from library documentation and solving a classification problem. Our approach of mining the commit history was inspired by previous works in the domain of library migration and update. However, instead of proposing migration rules to client developers, we propose transforming deprecations to library developers.

Recommending deprecations and replacements. To the best of our knowledge, our study is the first to propose supporting library developers by recommending deprecations and generating replacement rules. Brito *et al.,* [7] recommend replacement messages for deprecations by learning from client systems that have already identified the correct replacements and updated their code. Our approach takes information from the history of a library and therefore does not depend on migrated clients (which might not be available or may not cover the full API). Xi *et al.,* [42] designed an automatic process for migrating deprecating API to its replacement in client systems. However, their approach is based on the replacement messages in code documentation. One of the goals of our approach is to generate those replacement messages when they are not known.

8 Conclusion

Method deprecation is a powerful technique for supporting the evolution of software libraries and informing client developers about the upcoming breaking changes to the API. We proposed to mine the frequent method call replacements from the commit history of a library and use them to recommend method deprecations and transformation rules. We implemented our approach for Pharo IDE

in a tool called *DepMiner*. We applied our tool to five open-source projects and asked 6 core developers from those projects to accept or reject the recommended changes. In total, 134 proposed deprecations were accepted by developers as well as 4 transformation rules for the existing deprecations. 61 new deprecations and 2 transformations rules for existing deprecations were integrated into the Pharo project.

9 Acknowledgements

We want to thank all developers who participated in our experiment and helped us evaluate the recommendations, particularly Guillermo Polito, Pablo Tesone, Marcus Denker, and Benoît Verhaeghe. We are also grateful to the Arolla software company for financing this research.

References

1. Alrubaye, H., Mkaouer, Mohamed Wiemand Ouni, A.: On the use of information retrieval to automate the detection of third-party java library migration at the method level. In: ICPC'19 (2019)
2. Alrubaye, H., Mkaouer, M.W., Khokhlov, I., Reznik, L., Ouni, A., Mcgoff, J.: Learning to recommend third-party library migration opportunities at the API level. Journal of Applied Software Computing pp. 106–140 (2020)
3. Baldassarre, M.T., Bianchi, A., Caivano, D., Visaggio, G.: An industrial case study on reuse oriented development. In: 21st IEEE International Conference on Software Maintenance (ICSM'05). pp. 283–292. IEEE (2005)
4. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 241–250. ICSE '11, ACM, New York, NY, USA (2011)
5. Brito, A., Valente, M.T., Xavier, L., Hora, A.: You broke my code: understanding the motivations for breaking changes in APIs. Empirical Software Engineering **25**(2), 1458–1492 (2020)
6. Brito, G., Hora, A., Valente, M.T., Robbes, R.: Do developers deprecate APIs with replacement messages? a large-scale analysis on java systems. In: International Conference on Software Analysis, Evolution, and Reengineering (SANER). pp. 360–369. IEEE (2016)
7. Brito, G., Hora, A., Valente, M.T., Robbes, R.: On the use of replacement messages in API deprecation: An empirical study. Journal of Systems and Software **137**, 306–321 (2018)
8. Callau, O., Robbes, R., Rothlisberger, D., Tanter, E.: How developers use the dynamic features of programming languages: the case of smalltalk. In: Mining Software Repositories International Conference (MSR'11) (2011)
9. Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. In: International Conference on Software Engineering (ICSE'08). pp. 481–490. ACM, New York, NY, USA (2008)
10. Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. ACM Transactions on Software Engineering and Methodology (TOSEM) **20**(4), 1–35 (2011)

11. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: ECOOP. pp. 404–428 (2006)
12. Dig, D., Johnson, R.: How do APIs evolve? a story of refactoring. Journal of Software Maintenance and Evolution: Research and Practice (JSME) **18**(2), 83–107 (Apr 2006)
13. Ducasse, S., Polito, G., Zaitsev, O., Denker, M., Tesone, P.: Deprewriter: On the fly rewriting method deprecations. JOT (2022)
14. Furr, M., hoon (David) An, J., Foster, J.S., Hicks, M.: Static type inference for Ruby. In: Symposium on Applied Computing (SAC'09) (2009)
15. Hora, A., Etien, A., Anquetil, N., Ducasse, S., Valente, M.T.: Apievolutionminer: Keeping api evolution under control. In: Proceedings of the Software Evolution Week (CSMR-WCRE'14) (2014)
16. Hora, A., Robbes, R., Anquetil, N., Etien, A., Ducasse, S., Valente, M.T.: How do developers react to api evolution? the Pharo ecosystem case. In: International Conference on Software Maintenance (ICSM'15). pp. 251–260 (2015)
17. Hora, A., Robbes, R., Tulio Valente, M., Anquetil, N., Etien, A., Ducasse, S.: How do developers react to api evolution? a large-scale empirical study. Software Quality Journal **26**, 161–191 (Mar 2018)
18. Oracle. how and when to deprecate APIs. java se documentation, https://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/deprecation/deprecation.html
19. Kim, M., Notkin, D., Grossman, D.: Automatic inference of structural changes for matching across program versions. In: International Conference on Software Engineering (ICSE'07). pp. 333–343. IEEE (2007)
20. Meng, S., Wang, X., Zhang, L., Mei, H.: A history-based matching approach to identification of framework evolution. In: International Conference on Software Engineering (ICSE). pp. 353–363. IEEE (2012)
21. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17**, 348–375 (1978)
22. Milojković, N., Béra, C., Ghafari, M., Nierstrasz, O.: Inferring Types by Mining Class Usage Frequency from Inline Caches. In: International Workshop on Smalltalk Technologies IWST'16. Prague, Czech Republic (Aug 2016)
23. Nascimento, R., Brito, A., Hora, A., Figueiredo, E.: JavaScript API deprecation in the wild: A first assessment. In: International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 567–571. IEEE (2020)
24. Pandita, R., Jetley, R.P., Sudarsan, S.D., Williams, L.: Discovering likely mappings between APIs using text mining. In: International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 231–240. IEEE (2015)
25. Passerini, N., Tesone, P., Ducasse, S.: An extensible constraint-based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types. In: International Workshop on Smalltalk Technologies (IWST'14) (Aug 2014)
26. Pluquet, F., Marot, A., Wuyts, R.: Fast type reconstruction for dynamically typed programming languages. In: Dynamic Languages Symposium (DLS). pp. 69–78. ACM, New York, NY, USA (2009)
27. Ren, B.M., Foster, J.S.: Just-in-time static type checking for dynamic languages. In: Conference on Programming Language Design and Implementation (PLDI) (2016)
28. Renggli, L., Gîrba, T., Nierstrasz, O.: Embedding languages without breaking tools. In: D'Hondt, T. (ed.) Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10). LNCS, vol. 6183, pp. 380–404.

Springer-Verlag (2010), http://scg.unibe.ch/archive/papers/Reng10aEmbeddingLanguages.pdf
29. Richards, G., Hammer, C., Burg, B., Vitek, J.: The eval that men do: A large-scale study of the use of eval in javascript applications. In: Proceedings of Ecoop 2011 (2011)
30. Robbes, R., Röthlisberger, D., Tanter, E.: Extensions during software evolution: do objects meet their promise? In: European Conference on Object-Oriented Programming (ECOOP). pp. 28–52. Springer-Verlag, Berlin, Heidelberg (2012)
31. Roberts, D., Brant, J., Johnson, R.E.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems (TAPOS) **3**(4), 253–263 (1997)
32. Roberts, D., Brant, J., Johnson, R.E., Opdyke, B.: An automated refactoring tool. In: Proceedings of ICAST '96 (Apr 1996)
33. Sawant, A.A., Robbes, R., Bacchelli, A.: On the reaction to deprecation of 25,357 clients of 4+1 popular java APIs. In: International Conference on Software Maintenance and Evolution (ICSME). pp. 400–410. IEEE (2016)
34. Schäfer, T., Jonas, J., Mezini, M.: Mining framework usage changes from instantiation code. In: International Conference on Software Engineering (ICSE). pp. 471–480. ACM, New York, NY, USA (2008)
35. Schärli, N., Black, A.P., Ducasse, S.: Object-oriented encapsulation for dynamically typed languages. In: Proceedings of 18th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'04). pp. 130–149 (Oct 2004)
36. Spoon, S.A., Shivers, O.: Demand-driven type inference with subgoal pruning: Trading precision for scalability. In: Proceedings of ECOOP'04. pp. 51–74 (2004)
37. Suzuki, N.: Inferring types in smalltalk. In: Symposium on Principles of Programming Languages (POPL'81). pp. 187–199. ACM Press, New York, NY, USA (1981)
38. Teyton, C., Falleri, J.R., Blanc, X.: Automatic discovery of function mappings between similar libraries. In: Working Conference on Reverse Engineering (WCRE). pp. 192–201. IEEE (2013)
39. Wu, W., Guéhéneuc, Y.G., Antoniol, G., Kim, M.: Aura: a hybrid approach to identify framework evolution. In: International Conference on Software Engineering (ICSE). vol. 1, pp. 325–334. IEEE (2010)
40. Xavier, L., Brito, A., Hora, A., Valente, M.T.: Historical and impact analysis of API breaking changes: A large-scale study. In: International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 138–147. IEEE (2017)
41. Xavier, L., Hora, A., Valente, M.T.: Why do we break APIs? first answers from developers. In: International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 392–396. IEEE (2017)
42. Xi, Y., Shen, L., Gui, Y., Zhao, W.: Migrating deprecated API to documented replacement: Patterns and tool. In: Proceedings of the 11th Asia-Pacific Symposium on Internetware. pp. 1–10 (2019)
43. Xing, ZhenchangandStroulia, E.: API-evolution support with diff-catchup. IEEE Transactions on Software Engineering **33**, 818 – 836 (2007)