DepMiner: Automatic Recommendation of Transformation Rules for Method Deprecation

Oleksandr Zaitsev*†, Stéphane Ducasse†, Nicolas Anquetil†, and Arnaud Thiefaine*
*Arolla, Paris, France
†Inria, Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL, Lille, France
Email: {oleksandr.zaitsev, arnaud.thiefaine}@arolla.fr, {stephane.ducasse, nicolas.anquetil}@inria.fr

***Abstract*—Software applications often depend on external libraries and must be updated when one of those libraries releases a new version. Deprecation is a common practice for informing client developers about the breaking changes that will be introduced into the API of a library in future releases. However, knowing that certain functionality is deprecated is not enough. Developers of the client applications must be informed about what are the correct replacements that would fix their code. Library developers can transfer such in the form of deprecation messages, comments, documentation, or even the transformation rules that can be applied to client code. However, in practice, it is often the case that (1) functionality is removed without being deprecated; (2) deprecations are added before the release at which point even library developers find it hard to identify the correct replacements. We propose to help library developers by automatically generating deprecations and transformation rules based on the analysis of the two versions of library's API and mining the commit history. The generated deprecations can be inserted into the source code of a library before the release thus making it easier for the client developers to update their dependencies to the new version. We implemented our approach in a tool called *DepMiner* that recommends methods to deprecate and generates transformation rules for them. We have applied our tool to five open-source projects and proposed the generated deprecations to project developers. They marked 138 recommended deprecations as acceptable. 63 generated deprecations were integrated into the Pharo project.**

***Index Terms*—software evolution, software migration, library update, deprecations, data mining**

I. Introduction

Most modern software depends on multiple external libraries [1]. 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: *e.g.,* classes, methods, or fields get renamed, deleted, or moved around, new functionalities are introduced, etc. [2]. As a result, developers depending on those libraries must either update their code or continue having outdated and no longer maintained dependencies.

Deprecation is a common practice for supporting the 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 following release *n+1* or sometimes even later, in release *n+k*. The 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 messages that suggest a correct replacement for an obsolete item. Modern programming languages and IDEs provide various tools that make deprecations more informative and powerful. For example, in Java, the `@Deprecated` annotation as well as the `@deprecated` Javadoc tag can mark a method or class as deprecated while the `@link` or `@see` tags can reference the correct replacement in the source code [3]. The JetBrains MPS (MetaProgrammingSystem) platform adds an extension to IDE that allows programmers to find all usages of deprecated methods or classes in their code [4]. While the above approaches only inform client developers about the correct replacement, deprecations in Pharo[1] programming language can be supplied with transformation rules that will automatically fix client code. When deprecated method is invoked, the call-site is dynamically identified and automatically fixed at runtime without interrupting the execution [5]–[7].

However, developers of real projects do not always follow the 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 [8]–[10]. 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# [11], and 67% for JavaScript [12]. Brito *et al.,* have also demonstrated that large systems (both in terms of source code and community sizes) have smaller percentage of deprecations with replacement messages, arguably because small systems are easier to maintain.

We propose to target this problem by helping library developers introduce well-documented deprecations into their systems before every release. To that end, we designed an approach that recommends (1) methods[2] should be deprecated; (2) correct replacements for deprecated methods expressed in the form of transformation rules that can be used to automatically fix client code. Our approach is based on the

This work was financed by the Arolla software company

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

[2]In our study, we focus only on method deprecations. However, similar approach can be developed for classes and fields

idea that when a method of a library is renamed, replaced by one or more other methods, or removed without replacement, other locations in the code of the library that used this method (unit tests for example) must often be updated accordingly. We propose to mine the commit history of the library to learn how it has reacted to the changes in its own API, and then use this information to generate transformation rules for method deprecations. By mining small repetitive changes from library's commit history and combining this information with a list of methods that were added/removed between the two versions of the library, we can recommend methods that should be deprecated and generate transformation rules for those deprecations. We may also retroactively mine past changes to propose transformation rules for methods that were manually deprecated but were not supplied with a transformation rule or a replacement message.

We implemented this approach in a tool called *DepMiner* and used it to recommend deprecations for 5 open source projects. The tool generated 138 recommendations that were accepted by the developers of those projects, including 134 new deprecations and 4 transformation rules for existing deprecations. We have submitted the deprecations that were generated for Pharo project as pull requests. 61 new deprecations were integrated into the project and 2 transformation rules were added to existing deprecations.

Mining the change history to generate code transformation rules is not a new idea. Schaffer et al. [13], Teyton et al. [14], and Hora et al. [15] mined the history of commits to recommend the rules for updating client system to the new version of an external library. Pandita *et al.,* [16] and Alrubaye *et al.,* [17] used a similar technique to help client developers replace dependencies to one library with dependencies to another one. Brito *et al.,* [11] have designed a recommendation tool that suggests replacement messages for deprecations by learning from client systems that have already identified the correct replacements and updated their code.

We propose to take a different approach and recommend transformation rules for method deprecations by mining the commit history of the library instead of the client systems that were already updated. The main benefit of our approach is that it does not rely on client systems can be used by library developers before they release the new version. We also generate the transformation rules in a matching language supported by Pharo. Those rules can automatically fix client code at runtime without raising deprecation warnings or forcing client developers to search for the correct replacement. Finally, because Pharo is a dynamically-typed language, our approach does not rely on the information about the type of receiver or arguments of a method call. It can be extended to work with other increasingly popular dynamically-typed languages such as JavaScript, Python, Ruby, etc.

The contributions of the paper are (1) the automatic recommendation of methods that should be deprecated based on the API analysis and the commit history; (2) the generation of their accompanying transformation rules based on the frequent method call changes; (3) a first validation with developers.

The rest of this paper is structured as follows. In Section II, we start with a motivating example and discuss the challenges of mining method call changes from the commit history. In Section III, we describe our proposed approach and explain the underlying data mining algorithm. In Section IV, 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 V, we explain the limitations of our approach and discuss the future work.

II. Motivating Example and Challenges

A. Motivating Example

The latest stable version of Pharo (v8.0.0) contained a method RGInstanceVariableSlot.isSpecial(). In the next version of Pharo, this method is renamed to needsFullDefinition(). As a result, client developers will have to find the correct replacement for isSpecial(), which will not be an easy task, especially considering that the new method name is very different from the old one. The good solution for the developers of Pharo would be to annotate the method isSpecial() as deprecated, add a descriptive message to it, and a transformation rule that will automatically fix client code. However this was not immediately done and remembering all deprecations that were introduced during the last year would now be a difficult task.

On the other hand, when isSpecial() was renamed, 8 other methods that invoked it had to be updated. This means that 8 times throughout the commit history, a method call to isSpecial() was replaced with needsFullDefinition(). Also, whenever isSpecial() was removed, it was always replaced with needsFullDefinition() and never with another method call. Mining this kind of patterns from the commit history would allow to identify the correct replacement rule and reintroduce isSpecial() into Pharo 9 with a transforming deprecation to ensure backward compatibility.

B. Transforming Deprecation

Modern programming languages and IDEs like Pharo allow developers to enrich deprecations with code transformation rules. If a client system invokes the deprecated method, its source code is automatically fixed at run time to call the replacement:

```
1  isSpecial
2      self
3          deprecated: 'Renamed to #needsFullDefinition'
4          transformWith: '`@rec isSpecial'
5                  -> '`@rec needsFullDefinition'.
6
7      ↑ 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 can replace method calls to isSpectial 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. However, as we analysed 470 valid deprecations in Pharo 8, we discovered that 190 of them (40%) do not contain transformation rules. Out of those 190 non-transforming deprecations, 41 (22%) can have a simple transformation rule that can be written by a developer with no project expertise or even generated automatically. 85 (45%) deprecations require developers with project expertise to provide extra information (additional argument, default value, etc.) and write a rule manually. The other 64 (34%) deprecations 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. We performed a survey including 46 developers out of which 13 people answered a question about the difficulty of writing transformation rules: 3 developers found it easy or very easy; 9 developers found it of medium difficulty; and 1 developer marked it as difficult.

Similar trends can be observed in other programming languages. For example, according to the large-scale studies of software systems, deprecations that contain a helpful replacement message (in a form of comment, string, annotation, etc.) constitute only only 66.7% of all deprecations in Java projects, 77.8% in C# [11], and 67% in JavaScript projects [12].

C. Challenges

Before discussing the proposed approach, it is important to understand several challenges that arise when we mine method call changes from the commit history of a dynamically-typed programming language:

a) Not all changes are related to deprecations: Often programmers simply change the code to introduce the new behaviour, simplify or optimise the implementation, etc. For example, many frequent method call replacements describe good coding practices such as $\{\mathsf{ifTrue:}, \mathsf{not}\} \rightarrow \{\mathsf{ifFalse:}\}$. Even though this is a valid rule that can be mined from the commit history, it does not mean that ifTrue: or not should be deprecated.

b) Absence of method visibility: Deprecations are usually reserved for public methods only. Private methods can not be used outside of the class that defined them. Hence, those methods can be removed without being deprecated because there are no external users who should be informed about the removal of a private method. Recommending to deprecate a private method can be considered a false positive. So before making the recommendations, we need to know which methods of the project are public (part of the API) and which ones are private (intended only for the internal usage). Languages like Java and C++ have a public keyword that can help identify methods that are part of API. However, in languages like Python or Pharo all methods are public. 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. The absence of method visibility in modern programming languages adds a layer of complexity to our task.

c) Dynamically-typed languages: Many modern programming languages such as Python, Ruby, Javascript, or Pharo [18], [19] are dynamically-typed. Those languages do not have a static type information which complicates the task of identifying correct method mappings between the old and the new version.

Consider the following example. The log: method can be implemented by two different classes: Logger and Number. In the first case, log: is a method that writes a log message onto the stream. In second case, log: represents the mathematical logarithm. When in the commit history of a dynamically-typed programming language we see that the method call to log: was deleted, we do not know what was the class of its receiver and therefore which of the two methods was being called.

Additionally, inheritance and polymorphism across different hierarchies produce code where a single user ends up using multiple implementations at run-time. For instance, in Pharo 8 the method name() is called 3109 times and implemented in 346 classes, isEmpty() has 1595 callers and 103 implementors.

When working with dynamic languages, we do not know the types of arguments. This has an important implication that we can get a combinatorial explosion when wanting to use sequence of messages in the analysis. Megamorphic calls can be removed from the analysis but still highly polymorphic methods render the analysis unusable. The research community did propose in the past to use type inference for dynamically-typed languages [18], [20]–[24] or use Dynamic Type information collected by the Virtual Machine to get concrete types [25], [26]. But such type inferencers often do not cover the full language [18] or are not applicable to large code bases [22]. In this project, however, we do not perform type inference and consider that the type information is simply missing.

III. Approach

We propose to assist library developers in the task of finding proper replacements for the deprecated methods by mining frequent method call replacements from the commit history [13]–[15], [17]. The idea is the following: when certain method signature from the external library gets modified (*e.g.,* method is renamed, split into multiple methods, argument is removed, etc.), the library's usage of that method has to be updated. This means that all calls to the old method in the source code of the library will be replaced with the calls to the one or many new methods. This can affect other methods of the library that use the method that has changed, unit tests that cover the changed method, or examples that demonstrate how it can be used (in Pharo, it is a common practice to add specially annotated example methods to the class side).

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 [27]–[29].
4) Use frequent method call replacements to recommend transformation rules. Combine this information with the list of deleted methods to propose new deprecations.

In the following sections, we will discuss each step in details.

A. Identifying Methods of the Old and the New API

First we need to define two sets of methods: the ones that were part of the old API and those that belong to the new API. Every rule that we will generate must replace a method from the old API with one or more methods from the new API.

As we have discussed in Section II, many programming languages do not have method visibility which means that we do not know for sure which methods are public and which ones are private. For those languages, in case the project API is not explicitly defined by developers, we define that the API consists of all methods except the test methods or example methods. We will denote two sets that contain methods of the old and new API as API_{old} and API_{new} respectively.

For dynamically-typed languages such as Pharo, where we can not know which implementation of a method will be invoked by a method call, the sets API_{old} and API_{new} only need to contain the method names[3].

We will also define two other sets, API^*_{old} and API^*_{new}, with an asterisk, equivalent to the first ones but excluding explicitly deprecated methods. Explicitly deprecated methods can be considered part of the API (sets API_{old} and API_{new}) because they can be accessed by clients of the library. However, for this study, we remove them from the API because we are only interested in methods that are "supposed to be used" in a given version of the project. To generate transformation rules for method deprecations, we will need to find the mapping from API_{old} into API^*_{new}.

B. Collecting Method Call Changes

Given the history of commits between the old version and the new version of the project, we extract method changes from every commit. A *method change* describes how one specific method was changed by a given commit. We define method change as a combination of five values: *commit SHA*, *class name*, *method name*, *old source code*, and *new source code*.

For each method change, we parse its source code (old and new one) and extract method calls from it[4]. By comparing the method calls that appeared in the source code of a given method before and after the commit, we get the sets of deleted and added method calls for every method change. From every set of deleted method calls, we remove all methods that are not included in API_{old}. Similarly, we remove added method calls that are not part of API^*_{new}. Notice that the first set may contain methods that were deprecated in the old version because those deprecations may still remain in the new version of the library. However, the second set can not contain deprecated methods because we do not want to replace anything with a method that will be removed. As a result, we get a collection of method changes where each method change (m, c) describes how method m was changed by commit c using two sets: (1) a set of calls to the methods from the old API that were removed from the source code of method m by commit c and (2) a set of calls to non-deprecated methods from the new API that were added to the source code of method m by commit c.

We remove all method changes for which either of those sets is empty. Method changes that contain too many deleted or added method calls are probably the ones that were changed to introduce a new feature and not to update the usage of a deleted or renamed method. Those method changes are not useful and only create noise in our dataset, so to filter them out 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 $K = 3$).

C. Using A-Priori to Find Repetitive Changes

After collecting the dataset of method call replacements from the commit history, we apply a data mining algorithm to find all frequent subsets of those replacements. To understand what are frequent method call replacements and how they can be extracted from method changes, consider the following example. We have a collection of two method changes with the following sets of added and deleted method calls:

```
Method Change 1:
  deleted: { isEmpty, not, add: }
  added: { new, isNotEmpty }.

Method Change 2:
  deleted: { remove:, isEmpty, not }
  added: { isNotEmpty, sort }.
```

In both method changes, the calls to isEmpty() and not() were deleted and a call to isNotEmpty() was added. This repetitive change allows us to infer the rule

$$\{\text{isEmpty}, \text{not}\} \rightarrow \{\text{isNotEmpty}\}$$

We say that the *support* of this rule is 2 because in the database of method changes it appeared twice.

The problem of finding such repetitive replacements can be expressed in terms of frequent itemset mining (market basket problem). It goes like this: given the set of transactions, find all sets of items that often appear in transactions together. For example, six customers have bought products in the supermarket on a given day. This gives us six transactions:

[3]Method names in Pharo incorporate the information about the number of arguments. For example, a method assert:equals: has two argument placeholders. It can be called as self assert: 1+2 equals: 3

[4]In our study, we used the Iceberg tool for this purpose: https://github.com/pharo-vcs/iceberg

```
T1: {eggs, milk, butter}
T2: {milk, cereal}
T3: {eggs, bacon}
T4: {bread, butter}
T5: {bread, bacon, eggs}
T6: {bread, avocado, butter, bananas}
```

Using an algorithm for frequent itemset mining, such as *A-Priori*, we find that the following two subsets of items frequently appear in transactions together with support[5] equal to 2:

```
{eggs, bacon}
{bread, butter}
```

To represent method changes as transactions that can be used for frequent itemset mining, we join the sets of deleted and added method calls into a single set. This way, two method changes from the example above become two transactions:

```
T1: {deleted(isEmpty), deleted(not),
     deleted(add), added(new),
     added(isNotEmpty}
T2: {deleted(remove), deleted(isEmpty),
     deleted(not), added(isNotEmpty),
     added(sort)}
```

To find frequent itemsets using A-Priori, we need to specify the minimum support threshold N. In other words, we consider a set of items frequent if they appear together in at least N transactions. So in the example above, if we run A-Priori with minimum support $N = 2$, we will get the following itemsets:

```
{deleted(isEmpty), deleted(not)}
{deleted(isEmpty), added(isNotEmpty)}
{deleted(not), added(isNotEmpty)}
{deleted(isEmpty), deleted(not),
 added(isNotEmpty)}
```

Now we construct association rules by putting all deleted method calls into the left hand side (*antecedent*) and all added method calls into the right hand side (*consequent*). We remove the rules with empty antecedent or empty consequent. This gives us the following association rules:

$$\{\text{isEmpty}\} \rightarrow \{\text{isNotEmpty}\}$$
$$\{\text{not}\} \rightarrow \{\text{isNotEmpty}\}$$
$$\{\text{isEmpty, not}\} \rightarrow \{\text{isNotEmpty}\}$$

Note that those association rules are different from the ones that are generated by the A-Priori algorithm when it is used for association rules mining. A-Priori generates association rules by going through all possible partitions of the elements taken from itemset into the antecedent and consequent sets. But we already know which method calls were deleted and which ones were added and therefore, in our case, one itemset is directly mapped into one association rule.

[5]We use the word *"support"* to identify the absolute frequency — number of transactions that include a given itemset. However, in the terminology of market basket analysis, *"support"* can sometimes refer to the relative frequency — the number of transactions that include a given itemset divided by the total number of transactions.

To reduce the number of association rules, we calculate their confidence and use it to filter out the rules for which the confidence value is low. To calculate the confidence of a rule $I \rightarrow J$, we divide the number of method changes that deleted a set of method calls I and added the set of method calls J by the number of all method changes that deleted the set of method calls I:

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

Confidence can be treated as a probability that a set of method calls I is replaced with method calls J and not with something else. In our little example, all three association rules will have the confidence value 1. However, in a larger dataset of method changes, associations $\{\text{isEmpty}\} \rightarrow \{\text{isNotEmpty}\}$ and $\{\text{not}\} \rightarrow \{\text{isNotEmpty}\}$ will have relatively small confidence because not every time that the method call not() is deleted is it replaced with the method call isNotEmpty(). Similarly, the calls to isEmpty() are not always replaced with isNotEmpty(). However, the last rule $\{\text{isEmpty, not}\} \rightarrow \{\text{isNotEmpty}\}$ will have a higher confidence because when both method calls isEmpty() and not() are removed, they are often replaced with isNotEmpty(). By choosing a high enough confidence value, we will filter out the first two rules and keep only the last one.

D. Generating Recommendations

The data mining process described above, parametrised with minimum support and minimum confidence values, takes the collection of method changes as input and returns the collection of association rules. Each rule represents a method call replacement that frequently appeared in the collection of method changes. Now we can use those association rules to propose method deprecations or generate transformation rules for the existing deprecations. For this task, we are only interested in one-to-one or one-to-many rules — 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. We will denote this filtered collection as $Assoc$.

Based on two sets of methods, API_{old} and API_{new} that were discussed in Section III-A, 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 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 in the form ‘@rec selector1: ‘@arg → ‘@rec selector2: ‘@arg can be 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 at least one of those conditions is not satisfied, then the transformation rule can not be generated and must be written manually by a developer who has some additional knowledge about the code base (*e.g.,* if there are two added method calls, which one should be called first? if the added method call has more arguments than the deleted one, what should be the values of those extra arguments?). In those cases, we only show 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.

E. Implementation

We have implemented our approach for Pharo IDE in a tool called *DepMiner*. Our tool provides a user interface for browsing two versions of a given project, mining frequent method call replacements from the commit history between those two versions, and generating deprecations with transformation rules that can be automatically inserted into the source code of the project. In Figure 1, you can see *DepMiner* proposing library developer to reintroduce method isSelf with a generated deprecation and a transformation rule ‘@rec isSelf → ‘@rec isSelfVariable.

IV. Evaluation

We have applied *DepMiner* to several open source projects and used two strategies to evaluate the generated recommendations:

1) Comparing the generated transformation rules to the ones that are already present in the project.
2) Asking project developers to use DepMiner and analysing the recommendations which they accepted.

A. Evaluation Setup

a) Selected projects: For this study, we have selected five open source projects:

- **Pharo**[6] — a large and mature system with more than 20 active and regular contributors, containing the language core, the IDE, and various libraries.
- **Moose Core**[7] — Moose is a large platform for data and source code analysis. It consists of many different repositories but because our tool works on the level of single repository, we focus only on the core repository of Moose.
- **Famix**[8] — generic library that provides an abstract representation of source code in multiple programming languages. Famix is part of the Moose project.
- **Pillar**[9] — a markup syntax and tool-suite to generate documentation, books, websites and slides.
- **DataFrame**[10] — 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 or are still under active development, (3) we wanted to compare the performance of *DepMiner* on the projects with different maturity and complexity levels.

In Table I we describe each project with the following metrics: type of project, number of packages, number of methods, number of deprecated methods, total number of contributors on GitHub, number of commits per day (calculated as the total number of commits between March 25, 2020 and March 25, 2021, divided by 356), and number of stars on GitHub. For this study, we define three types of projects:

- **Tool** — a project that is designed for the end users. 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. 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.
- **IDE** — 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 various software that is built with Pharo. This means that deprecations are very important for this type of projects.

b) 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 will compare the new version of the project. All patterns will

[6]https://github.com/pharo-project/pharo
[7]https://github.com/moosetechnology/Moose
[8]https://github.com/moosetechnology/Famix
[9]https://github.com/pillar-markup/pillar
[10]https://github.com/PolyMathOrg/DataFrame

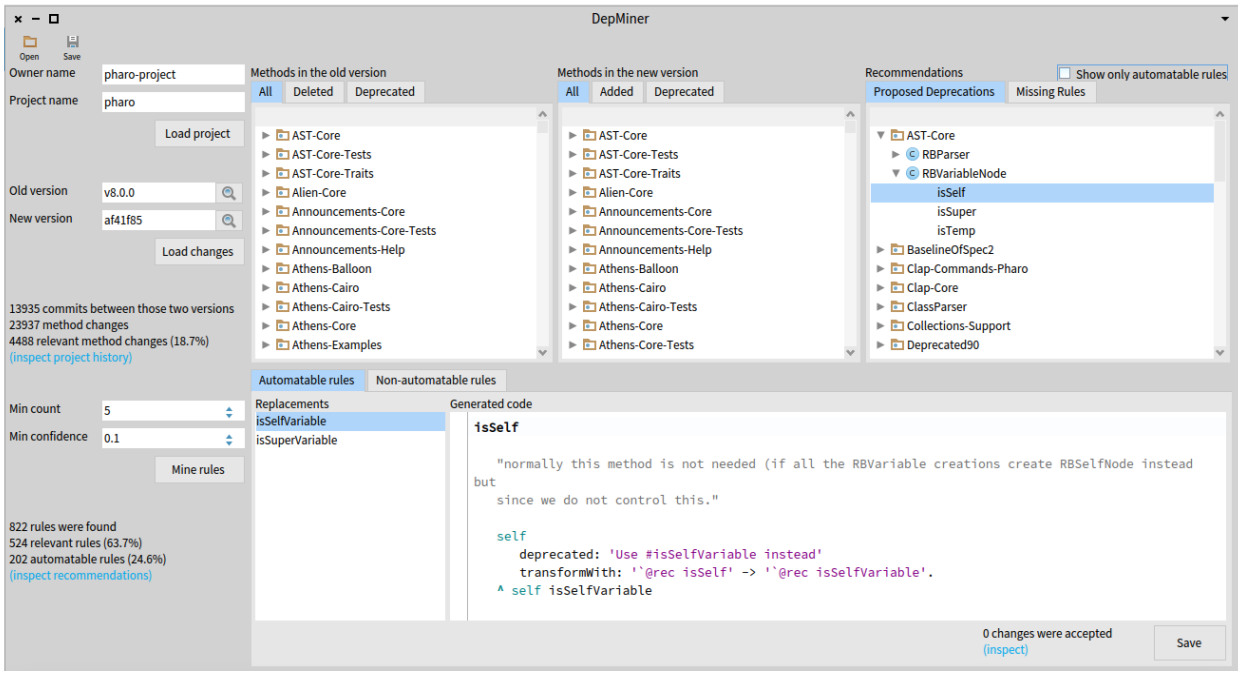

Fig. 1. DepMiner tool in Pharo 9 showing the proposed deprecation of isSelf to the developer

TABLE I
SOFTWARE PROJECTS THAT WE HAVE SELECTED FOR OUR STUDY

Project	Type	Packages	Methods	Depr. Methods	Contributors	Commits per day	Stars
Pharo	IDE	736	116,212	515 (0.4%)	130	11.1	610
Moose Core	Tool	16	1,670	2 (0.1%)	19	1.3	88
Famix	Library	56	6,538	113 (1.7%)	19	2.2	2
Pillar	Tool	57	5,848	2 (0.03%)	19	1.5	40
DataFrame	Library	8	661	0	6	0.05	52

then be mined from the slice of the commit history between those two versions. In Table II, we list the two versions of each project that we have loaded as well as the number of commits between those two versions, the total number of method changes extracted from those commits, and the number of method changes that are relevant for our analysis (see Section III-B).

c) 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 III, 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 number of support = 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 relevant 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 by 0.1.

In Table IV, we present the number of association rules (frequent method call replacements) that were found by *DepMiner* given the settings discussed above, as well as the number of relevant association rules that can be used to propose deprecations (one-to-one or one-to-many rules), and the number of rules that can automatically generate the transformation rules in the form '@rec deletedSelector: '@arg → '@rec addedSelector: '@arg (only one-to-one rules where deleted and added selectors have the same number of arguments).

B. Comparing Generated Deprecations to the Existing Ones

Before comparing transformation rules that were generated by *DepMiner* to the ones that were already present in the projects, it is important to understand the different nature of those two sets. First set contains the rules that were mined from the commit history based on the analysis of how the library was using its own API. This set can not contain transformation rules for methods that are not called by other methods of the same library or not covered by multiple tests. The other set contains the transformation rules that were manually written by developers. This set does not share the same restriction.

TABLE II
COMMIT HISTORIES BETWEEN THE OLD AND THE NEW VERSIONS OF SELECTED PROJECTS

Project	Old version	New version	Commits	Method changes	Relevant
Pharo	v8.0.0	af41f85	3,465	23,937	4,488 (19%)
Moose Core	v7.0.0	v8.0.0	1,519	22,587	466 (2.1%)
Famix	a5c90ff	v1.0.1	948	9,391	1,295 (14%)
Pillar	v8.0.0	v8.0.12	508	2,294	132 (6%)
DataFrame	v1.0	v2.0	225	649	206 (32%)

TABLE III
VALUES OF THE MINIMUM SUPPORT AND MINIMUM CONFIDENCE THRESHOLDS THAT WERE USED TO MINE FREQUENT METHOD CALL REPLACEMENTS

Project	Min support	Min confidence
Pharo	5	0.4
Moose Core	2	0.1
Famix	4	0.4
Pillar	2	0.1
DataFrame	5	0.1

TABLE IV
ASSOCIATION RULES (FREQUENT METHOD CALL REPLACEMENTS) MINED FROM COMMIT HISTORY

Project	Assoc. rules	Relevant	Transforming
Pharo	670	377	152
Moose Core	183	88	40
Famix	233	149	60
Pillar	62	49	16
DataFrame	39	22	7

Although two sets are different in nature, it is still interesting to see how they intersect. In Table V, we report the numbers of unique transformation rules that were already present in each project, the number of transformation rules generated by *DepMiner*, and the number of rules that belong to both sets. For Pharo project, *DepMiner* managed to reproduce 24 out of 264 existing rules based on the commit history (9%). For Famix, it reproduced 15 out of 74 existing rules (20%). Other three projects do not have enough transformation rules.

TABLE V
NUMBER OF EXISTING UNIQUE TRANSFORMATION RULES, GENERATED TRANSFORMATION RULES, AND THE INTERSECTION OF THOSE TWO SETS

Project	Existing	Generated	Intersection
Pharo	264	152	24
Moose Core	0	40	0
Famix	74	60	15
Pillar	2	16	0
DataFrame	0	7	0

C. Evaluation by Project Developers

We have performed a first developer study of our tool involving the core developers from each project listed in Section IV-A. We asked 4 developers with different areas of expertise to validate the recommendations generated for Pharo and 1 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 accept the changes which, in their opinion, should be merged into the project. Then they sent us the list of accepted changes. *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 a given method call replacement has appeared. This allowed developers who participated in our study to make an informed decision. Some developers approached this task very diligently and spent several hours discussing the recommended deprecations. For Pharo project we considered the recommendation accepted if it was accepted by at least one developer. Then we have submitted the accepted recommendations as pull requests, 63 of which were merged into the project.

a) Proposed deprecations: In Table VI, we report the numbers of deprecations that were recommended to developers for each project, the number of those recommendations that were accepted, and the number of accepted recommendations that contain an automatically generated transformation rule. Each recommended deprecation, in this case, 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 VI
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.

In Table VII, we provide several examples of deprecations that were proposed by *DepMiner* for Pharo project. First column contains the source code of a method that existed in Pharo 8 but was removed in Pharo 9. By analysing the history of commits between Pharo 8 and 9 (see Table II), out tool has inferred the association rules that define replacements for deleted methods. Those rules are presented in the second column of Table VII together with support (number of times the replacement took place in the commit history) and confidence (probability that deleted method was replaced with the one that is being suggested and not with something else). Using the mined association rules, *DepMiner* generated deprecations with transformation rules that are presented in the third column of the table. Those recommendations were validated by four developers from Pharo project and submitted as pull requests (PR). The results are presented in the last column.

First example in Table VII presents recommendation that was accepted by all four developers and integrated into Pharo as pull request. It is an interesting case of renaming because the old method name isSpecial is very different from the new one needsFullDefinition and therefore hard to identify by developers who are reading the source code. Second example presents the case when the calls to atName: must be replaced with at:. This recommendation was validated only by one out of four developers (perhaps, because at: is a very generic method in Pharo). However, after further discussion, it was accepted as PR into Pharo 9. The final example presents the recommendation that was rejected by project developers. In the commit history of Pharo 8 $\rightarrow$ 9, the method call to assoc: was replaced with variable:. Also, as indicated by confidence, 75% of times when the call to assoc: was removed, a call to variable: was added. However, a closer inspection of method changes from the commit history shows that those two methods are called with different types of arguments. In the code listing below, we present one of those method changes.

```
"In Pharo 8"
OCClassScope >> lookupVar: name
  ↑(class innerBindingOf: name)
    ifNotNil: [:assoc | OCLiteralVariable new
      assoc: assoc;
      scope: self;
      yourself]
    ifNil: [ outerScope lookupVar: name]
```

```
"In Pharo 9"
OCClassScope >> lookupVar: name
  ↑(class innerBindingOf: name)
    ifNotNil: [:var | OCLiteralVariable new
      variable: var;
      scope: self;
      yourself]
    ifNil: [ outerScope lookupVar: name]
```

In the old implementation, the method innerBindingOf: returned an association which was then passed to the assoc: method. However, in the new implementation, innerBindingOf: returns a variable and then it is passed to the variable: method. In a dynamically-typed programming language such as Pharo, it is not easy to know the type of arguments or the type of value returned by a method before runtime. That is why *DepMiner* fails to understand that despite high values of support and confidence, variable: is not a new name for assoc: but a different method.

b) Missing rules: The other type of recommendations that we showed to developers were transformation rules for existing non-transforming deprecations. In Table VIII, we report the number of existing deprecations in the project 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.

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 II-B, for 22% of non-transforming deprecations the transformation rule can be generated automatically (given 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). We present the accepted rules in Table IX.

V. Discussion and Future Work

A. Limitations

a) 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. Consider a GUI application that has a method open which opens an application in a new window. No other method of the library is calling the open method and because it is hard to test user interfaces, this method is also not tested. When later the method open is renamed to openInWindow, there are no method calls in the

TABLE VII
EXAMPLES OF PROPOSED DEPRECATIONS

Deleted method	Rule	Generated Deprecation	Accepted

`RGInstanceVariableSlot >> isSpecial`
`  ↑ false`
	{isSpecial} → {needsFullDefinition}
	
	Support: 8
	Confidence: 1
		`RGInstanceVariableSlot >> isSpecial`
		`  self`
		`  deprecated:`
		`  'Use #needsFullDefinition instead'`
		`  transformWith: ''@rec isSpecial'`
		`  -> ''@rec needsFullDefinition'.`
		`  ↑ self needsFullDefinition`
			Yes
			
			Validated by all four developers; accepted as PR

`ClapCommand >> atName: specName` `  ↑ self subcommandNamed: specName ifNone:` `  [ self flagNamed: specName ifNone:` `  [ super atName: specName ] ]`	{atName:} → {at:} Support: 24 Confidence: 0.45	`ClapCommand >> atName: specName` `  self` `  deprecated: 'Use #at: instead'` `  transformWith: ''@rec atName: `@arg'` ` -> ''@rec at: `@arg'.` `  ↑ self at: specName`	**Yes** Validated by one out of four developers; accepted as PR
`OCLiteralVariable >> assoc: anAssociation` `  assoc := anAssociation`	{assoc:} → {variable:} Support: 15 Confidence: 0.75	`OCLiteralVariable >> assoc: anAssociation` `  self` `  deprecated: 'Use #variable: instead'` `  transformWith: ''@rec assoc: `@arg'` ` -> ''@rec variable: `@arg'.` `  ↑ self variable: anAssociation`	**No** Rejected by all four developers

TABLE VIII
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

source code of the library that are affected. And therefore, our approach will not be able to mine the rule for this replacement.

b) Reflective operations: Modern programming languages offer reflective operations [30], [31]. 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 [32].

c) Unordered set of method calls: Our tool is based on mining method call replacement by comparing the set of method calls that were deleted from the source code of a modified method to the set of method calls that were added to it. We do not take into account the order of method calls, the distance between them or the way how they are composed: a().b() or a(b()). This is a limitation of out 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 in which order they should be called or how they should be composed.

d) The importance of domain knowledge: The study of recommendations for Pharo project involved 4 developers with different areas of expertise. Out of 63 recommendations that were accepted as pull requests, only 6 were marked as "good" by all 4 developers, 7 recommendations were accepted by 3 out of 4 developers, 24 recommendations — by 2 developers, and 26 recommendations — by only 1 out of 4 developers. After further discussion, developers agreed that all 63 recommendations should be accepted. This shows that even though our approach simplifies the work of library developers and helps them generate deprecations long after the breaking changes were introduced, certain amount of domain knowledge is still required to validate the recommended changes.

e) Extra work done by A-Priori: Before applying the A-Priori algorithm to find the frequent method call replacements, we have to join deleted and added method calls into a single set. A-Priori will make no distinction between deleted and added method calls and among the itemsets that the algorithm will generate, there will also be the “invalid” ones that contain only deleted or only added method calls. Later, we filter them out and then split each set of frequent items proposed by A-Priori into the left (deleted calls) and right hand sides (added calls) of the rule. This means that by not understanding the nature of items in the itemsets, the A-Priori algorithm is doing some extra work by going through the “invalid” combinations of items. It could be more efficient to propose a modified version of the algorithm that keeps track of the left and right hand side items.

TABLE IX
MISSING TRANSFORMATION RULES THAT WERE ACCEPTED BY DEVELOPERS

Project	Class	Method	Rule
Pharo	Context	namedTempAt:	‘@rec namedTempAt: ‘@arg → ‘@rec tempNamed: ‘@arg
Pharo	Context	namedTempAt:put:	‘@rec namedTempAt: ‘@arg1 put: ‘@arg2 → ‘@rec tempNamed: (‘@rec tempNames at: ‘@arg1) put: ‘@arg2
Famix	FamixTAttribute	hasClassScope:	‘@rec hasClassScope: ‘@arg → ‘@rec isClassSide: ‘@arg
Famix	FamixTMethod	hasClassScope:	‘@rec hasClassScope: ‘@arg → ‘@rec isClassSide: ‘@arg

B. Future Work

a) Good practices: The approach that we propose for mining deprecation rules can also be used to identify good coding practices and show them as suggestions to developers. For example, if assert: and = are commonly replaced with assert:equals: or if ifTrue: and not are commonly replaced with ifFalse:, then we can mine those rules as “good coding practices” and show them as code critiques in the IDE:

$$\{\text{assert:}, =\} \rightarrow \{\text{assert:equals:}\}$$
$$\{\text{isTrue}, \text{not}\} \rightarrow \{\text{isFalse}\}$$

b) Dealing with long methods: As we have mentioned before, one of the limitations of our tool is that it does not consider the distance between method calls. For example, (isEmpty not) and (isEmpty ... 100 lines of code... not) will be the same. Projects in Pharo have relatively short methods (the median is 3 lines of code [33]), but in other languages where methods are longer it might be more appropriate to compare the line diffs instead of the whole methods. This would provide more granularity by extracting the sets of method calls that appear on the same or neighbouring lines and not in the same method.

c) More information about method calls: In this work, we mine method call replacements based only on method names. However, it may be more effective to supply method calls with some information about the receiver and arguments (it is hard in dynamic languages because we do not know the type and using the value is too restrictive), position in source code, or part of the AST related to the method call.

VI. RELATED WORK

A. Expert-guided library update

The first studies of how to support client developers in updating their systems to the new versions of evolving libraries were based on collecting the expertise of library developers in the form of transformation rules and applying them to client code in a semi-automatic manner. Chow and Notkin [34] proposed library developers to annotate changed functions with transformation rules that can be applied to client code. They designed a language for expressing code transformations and implemented a semi-automatic tool that applies transformations to AST and generates the modified source code. Henkel et al. [35] decided to reduce the added cost for library developers. They proposed to support the API evolution by recording the refactorings as they are performed on the library and “replaying” the recorded changes on the client codebase.

Deprecation messages are among the most common ways for library developers to recommend replacements for the removed functionality. Modern programming languages provide powerful support for annotating deprecated elements of API with references to the possible replacements. In their large-scale empirical study of deprecation messages in Java and C# projects, Brito et al. [11] reported that 66.7% of Java deprecations and 77.8% C# deprecations contain replacement messages.

B. Automatic library update and migration

In recent years, many approaches have been proposed that do not require the direct involvement of library developers. Those approaches extract all necessary information from the source code, commit history, or code documentation. Kim *et al.,* [36] proposed to find matches between the two versions of API by calculating textual similarities of method signatures collected from the source code of the two versions of the library. They defined a set of low-level API transformations (*e.g.,* package replacement, argument deletion, etc.) and performed a rule-based matching to find a mapping between the two versions of library API. Xing et al. [37] proposed a Diff-CatchUp tool that compares two versions of the library’s source code, detects changes to the API, and proposes transformation rules together with working usage examples. Unlike Kim et al., they calculated the structural similarity of source code and not only the textual similarity of method signatures. Unlike previous automatic approaches to library update, which compared two versions of library’s source code, the novel SemDiff tool proposed by Dagenais and Robillard [38] extracted the necessary information on a more granular level from the commit history of a library. It recommended changes to client systems based on how the library reacted to its own evolution. SemDiff could recognise changes that were more complex than simple refactorings, for example, method additions and deletions, and recommended multiple replacements for methods that no longer existed in API, supplying each one with a confidence score.

Schäfer *et al.,* [13] were the first authors who proposed mining library update rules from already updated client systems. They used the library’s unit tests as an additional source of information. Unit tests describe the use cases of the library’s API and therefore, can be treated as one of the clients that must react to API changes. Schäfer et al. generated rules for the library changes that were caused not only by refactorings

but also by conceptual changes (changed or replaced concepts, altering the responsibilities of the building blocks, removing certain behaviour, etc.). They used the A-Priori algorithm to mine the transformation rules from two versions of client code; however, their approach was only suitable for generating one-to-one rules. Wu et al. [39] proposed a hybrid approach called AURA (AUtomatic change Rule Assistant) that combined call dependency and text similarity analyses. Nguyen et al. [40] proposed a tool for library update (LibSync) that first uses the textual and structural similarities to find the mapping of the library functions and then extracts the usage graphs from already updated client systems and mines the transformations that need to be made to the client code. Meng et al. [41] proposed a history-based matching approach HiMa which compares consecutive revisions of a library obtained from its commit history and supplies this information with the analysis of commit messages to generate transformation rules for client systems.

Teyton *et al.,* [14] turned to the problem of library migration — replacing client dependency on a third-party library in favour of a competing library. They adopted and improved the approach of Schäfer et al. [13], but extracted method call changes from a commit history of clients that were already migrated. Independently, Hora *et al.,* [15] proposed a similar approach to find method mappings between different releases of the same library. They analysed the commit history of a library to detect frequent method call replacements. This way, they mined the transformation rules by learning from how the library adapts to the changes of its own API. Those techniques are similar to the one that we present in this paper, however, our approach is designed for deprecations and meant to be used by library developers before the release, not client developers after release.

The most recent research has focused on the problem of library migration. Pandita *et al.,* [16] calculated the textual similarity of documentation of the API entities from different external libraries and recommended the entities that were most similar to one another as possible replacements. They built a tool called TMAP (Text Mining based approach to discover likely API mappings) and used it to discover mappings from Java to C# API and from Java ME to Android API. Alrubaye et al. [17] mined the commit history of client systems that were already migrated from one third-party library to a different one and generated mappings for method replacements. They improved their results by calculating the textual similarity of method descriptions taken from library documentation. In their next study, Alrubaye et al. [42] proposed a novel machine learning approach RAPIM for the task of library migration. They extracted features such as the similarity of method signatures and documentation, represented them as numerical vectors, and trained a machine learning classifier to label method mappings as "valid" or "invalid".

VII. Conclusion

Method deprecations are a powerful technique for supporting the evolution of software libraries and informing client developers about the upcoming breaking changes to the API. Using the deprecation messages or code transformation rules that are supported by some modern IDEs, library developers can specify correct replacements for the deprecated functionality. However, in practice breaking changes are often introduced without deprecation and finding the correct replacements post-factum can be hard even for the same developers who introduced them. In this work, we propose 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.

VIII. Acknowledgements

We would like to thank Guilermo Polito, Pablo Tesone, Marcus Denker, and Benoît Verhaeghe for evaluating the deprecations that were generated for the open-source projects. We are also grateful to the Arolla software company for financing this research.

References

[1] M. T. Baldassarre, A. Bianchi, D. Caivano, and G. Visaggio, "An industrial case study on reuse oriented development," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 283–292.

[2] D. Dig and R. Johnson, "How do APIs evolve? a story of refactoring," *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, vol. 18, no. 2, pp. 83–107, Apr. 2006.

[3] "Oracle. how and when to deprecate apis. java se documentation," accessed: 2021-04-19. [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/javadoc/deprecation/deprecation.html

[4] V. Pech, "Handle deprecated code in your project. the mps blog," accessed: 2021-04-19. [Online]. Available: https://blog.jetbrains.com/mps/2019/04/handle-deprecated-code-in-your-project/

[5] D. Roberts, J. Brant, R. E. Johnson, and B. Opdyke, "An automated refactoring tool," in *Proceedings of ICAST '96, Chicago, IL*, Apr. 1996.

[6] D. Roberts, J. Brant, and R. E. Johnson, "A refactoring tool for Smalltalk," *Theory and Practice of Object Systems (TAPOS)*, vol. 3, no. 4, pp. 253–263, 1997.

[7] L. Renggli, T. Gîrba, and O. Nierstrasz, "Embedding languages without breaking tools," in *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*, ser. LNCS, T. D'Hondt, Ed., vol. 6183. Springer-Verlag, 2010, pp. 380–404. [Online]. Available: http://scg.unibe.ch/archive/papers/Reng10aEmbeddingLanguages.pdf

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

[9] L. Xavier, A. Hora, and M. T. Valente, "Why do we break apis? first answers from developers," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 392–396.

[10] A. Brito, M. T. Valente, L. Xavier, and A. Hora, "You broke my code: understanding the motivations for breaking changes in apis," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1458–1492, 2020.

[11] 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.

[12] R. Nascimento, A. Brito, A. Hora, and E. Figueiredo, "Javascript api deprecation in the wild: A first assessment," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 567–571.

[13] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 471–480.

[14] C. Teyton, J.-R. Falleri, and X. Blanc, "Automatic discovery of function mappings between similar libraries," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 192–201.

[15] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "Apievolutionminer: Keeping api evolution under control," in *Proceedings of the Software Evolution Week (CSMR-WCRE'14)*, 2014.

[16] R. Pandita, R. P. Jetley, S. D. Sudarsan, and L. Williams, "Discovering likely mappings between apis using text mining," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2015, pp. 231–240.

[17] H. Alrubaye, M. W. Mkaouer, and A. Ouni, "On the use of information retrieval to automate the detection of third-party java library migration at the method level," in *ICPC'19*, 2019.

[18] N. Suzuki, "Inferring types in smalltalk," in *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1981, pp. 187–199.

[19] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, 1978.

[20] M. Furr, J. hoon (David) An, J. S. Foster, and M. Hicks, "Static type inference for ruby," in *Symposium on Applied Computing (SAC '09)*, 2009.

[21] B. M. Ren and J. S. Foster, "Just-in-time static type checking for dynamic languages," in *Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[22] S. A. Spoon and O. Shivers, "Demand-driven type inference with subgoal pruning: Trading precision for scalability," in *Proceedings of ECOOP'04*, 2004, pp. 51–74.

[23] F. Pluquet, A. Marot, and R. Wuyts, "Fast type reconstruction for dynamically typed programming languages," in *DLS '09: Proceedings of the 5th symposium on Dynamic languages*. New York, NY, USA: ACM, 2009, pp. 69–78.

[24] N. Passerini, P. Tesone, and S. Ducasse, "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.

[25] N. Milojković, C. Béra, M. Ghafari, and O. Nierstrasz, "Inferring Types by Mining Class Usage Frequency from Inline Caches," in *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, Aug. 2016.

[26] H. Wilkinson, "Livetyping in action - automatic type annotation for dynamically typed languages - demo," 2019. [Online]. Available: https://2019.programming-conference.org/details/programming-2019-Demos/5/LiveTyping-in-Action-Automatic-Type-Annotation-for-Dynamically-Typed-Languages

[27] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.

[28] C. Borgelt, "Frequent item set mining," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 2, no. 6, pp. 437–456, 2012.

[29] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*, 3rd ed. Elsevier, 2011.

[30] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do: A large-scale study of the use of eval in javascript applications," in *Proceedings of Ecoop 2011*, 2011.

[31] O. Callau, R. Robbes, D. Rothlisberger, and E. Tanter, "How developers use the dynamic features of programming languages: the case of smalltalk," in *Mining Software Repositories International Conference (MSR'11)*, 2011.

[32] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 241–250.

[33] O. Zaitsev, S. Ducasse, and N. Anquetil, "Characterizing pharo code: A technical report," Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille ; Arolla, Technical Report, jan 2020. [Online]. Available: https://hal.inria.fr/hal-02440055

[34] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes." in *icsm*, vol. 96, 1996, p. 359.

[35] J. Henkel and A. Diwan, "CatchUp!: capturing and replaying refactorings to support API evolution," in *Proceedings International Conference on Software Engineering (ICSE 2005)*, 2005, pp. 274–283.

[36] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 333–343.

[37] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, vol. 33, pp. 818 – 836, 2007.

[38] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 1–35, 2011.

[39] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 325–334.

[40] H. A. Nguyen, T. T. Nguyen, G. Wilson, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph- based approach to api usage adaptation," *ACM Sigplan Notices*, vol. 45, pp. 302 – 321, 2010.

[41] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 353–363.

[42] H. Alrubaye, M. W. Mkaouer, I. Khokhlov, L. Reznik, A. Ouni, and J. Mcgoff, "Learning to recommend third-party library migration opportunities at the api level," *Applied Soft Computing*, p. 106140, 2020.