JOT
JOURNAL OF
OBJECT TECHNOLOGY

Deprewriter: On the fly rewriting method deprecations

Stéphane Ducasse[*†], **Guillermo Polito**[§], **Oleksandr Zaitsev**[‡*], **Marcus Denker**[*†], **and Pablo Tesone**[†*]
[*]Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL, France
[§]Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 - CRIStAL, France
[†]Pharo Consortium
[‡]Arolla, France

ABSTRACT Deprecations are a common way to indicate that a given feature or API will not be available in subsequent versions of a library or framework. While raising deprecation warnings lets developers of *libraries* evolve their APIs, the developers of *client applications* often have to *manually* rewrite their applications to adapt to the deprecation (removal, new APIs. . .). Some may use static analysis tools to support the rewriting. However, dynamically-typed languages or the use of reflective features often produce incorrect rewrite candidates. This is a costly activity that can lead to bug introductions.

In this article, we present a *method* deprecation approach and a tool called DEPREWRITER that can automatically rewrite the callers of deprecated methods during program execution. Clients of a deprecated API execute their program and associated tests, and DEPREWRITER dynamically rewrites the source code of methods that called a deprecated API to use the new API. The implementation of DEPREWRITER is based on dynamic program transformation: when a deprecated method is *executed*, a program transformation engine rewrites and recompiles the caller's code before continuing the execution.

The approach presented in this article has been developed by the Pharo consortium. Since 2016, DEPREWRITER is used in production in multiple distributions of the Pharo programming language: Pharo 6, 7, 8, and 9 alpha. This article presents and validates this approach.

The validation is done in two steps: first with an analysis of deprecations available in Pharo 8 and second with an open survey of software developers about DEPREWRITER. We studied 367 Pharo 8 deprecations, among which we analyzed the 218 rewriting deprecations that use transformation rules. We identified the validity conditions and reported defects to the community. We also proposed 33 transformation rules to be added to the non-rewriting deprecations. Both contributions were accepted into Pharo 9 alpha. We classified the rules and identified possible points of improvement. In addition, we performed a user survey and collected information from 46 software developers: some of them used existing DEPREWRITER' rules and executed them on their code, others used DEPREWRITER to create rewriting deprecations, and finally, some were not aware of DEPREWRITER. 28 of 46 developers (60%) reported that the rewriting deprecations helped them, while 10 stated the inverse and 8 were uncertain. After discussing the current implementation, we sketch possible implementations for other languages than Pharo, showing that the approach is general enough to be applied to other languages.

KEYWORDS Dynamically-typed language, deprecation, automatic rewriting, program transformation, runtime, refactoring, reflection, Pharo.

1. Introduction

Software has to evolve (Lehman 1996; Demeyer et al. 2002; Mens et al. 2004). Application programming interfaces (APIs) change and their users get impacted (Robbes, Lungu, & Röthlisberger 2012). One common practice for supporting the evolution of libraries and their APIs is to use *deprecations*. Instead of

JOT reference format:
Stéphane Ducasse, Guillermo Polito, Oleksandr Zaitsev, Marcus Denker, and Pablo Tesone. *Deprewriter: On the fly rewriting method deprecations*. Journal of Object Technology. Vol. 21, No. 1, 2022.
http://dx.doi.org/10.5381/jot.2022.21.1.a1

immediately modifying an API from release n to release $n+1$ of a software, deprecations suggest releasing an intermediate version of the software, indicating that a given feature or API will be changed or unavailable in subsequent versions (A. Brito et al. 2018; G. Brito et al. 2018). In other words, to change or remove a feature available in the release n, it is first deprecated in release $n+1$, and then the actual change takes place in release $n+2$. Such an intermediate version gives time to the library clients to update their code and adapt to the forthcoming changes.

Before going any further, we define the vocabulary for the rest of this article. We call

- *Client developers:* the developers of an application or library that is using another library whose API is changing and subject to deprecation practices.
- *Library developers:* the developers of an application or library that is changing and may use DEPREWRITER to help *client developers* adapt their application to changes.
- *Transformation rule:* a rule that describes the transformation of the source code in the form *antecedent* → *consequent*. Antecedent is the expression that matches the code that should be replaced. Consequent defines the replacement.
- *Rewriting deprecation:* a deprecation that contains a transformation rule and can automatically rewrite the call sites.
- *Non-rewriting deprecation:* a deprecation that does not contain a transformation rule.

Deprecations are reported to client developers in many different ways. Some deprecations are just listed in the documentation of the new release, written in prose. Other deprecations are integrated into the source code; thus, compilers and IDEs will report them to the client developer (A. Brito et al. 2018). However, it is up to the developers of client applications to manually transform and update their code to the new APIs (Kim et al. 2007). Some static analysis tools support such migrations, although dynamically-typed languages and the use of reflective features produce either too many or too few rewritings because of known limits of static analyses. Client developers are left to manually identify and rewrite deprecated call sites, a costly activity that leads to bug introduction.

In this article, we present DEPREWRITER: a *method* deprecation approach that *automatically rewrites the callers of the deprecated* APIs during the program execution of client application. Runtime information helps to distinguish the real deprecated method callers from other method calls with the same name. Using DEPREWRITER (which stand for DEPrecation REWRITER):

1. Library developers annotate the deprecated methods with transformation rules.
2. Then, when clients of the deprecated API execute their program, DEPREWRITER dynamically rewrites the source code of the methods that have called the deprecated APIs.

The implementation of DEPREWRITER is based on call-stack reification and navigation (Rivard 1996), dynamic program update, and tree matching rewriting (D. Roberts et al. 1996, 1997), however the approach is generic enough to be implemented using different mechanisms as explained in Section 8. When a deprecated method is executed, DEPREWRITER walks the stack to find the caller method and the call site from which the deprecated method is ivoked. Then it uses a program transformation engine to rewrite the call site's source code. Finally, the execution of the deprecated method continues. All changes are done automatically. These changes are editable in the IDE, just like any other change performed interactively by the programmer. Developers review such automatic changes, modify them and commit a potentially further adapted version of their updated code. As DEPREWRITER is integrated into the IDE, its usage does not impose the need for new abilities on client developers.

In recent years, many studies have shown how the migration rules can be extracted from source code (Kim et al. 2007; Xing 2007; Schäfer et al. 2008; Wu et al. 2010; Nguyen et al. 2010; Pandita et al. 2015) and the commit history (Dagenais & Robillard 2008; S. Meng et al. 2012; Teyton et al. 2013; Hora et al. 2014; Alrubaye & Mkaouer 2019). These approaches support the *developers of client applications* and help them update their systems to the latest versions of the external libraries without relying on the library developers to provide a set of migration rules. The DEPREWRITER approach, which we present in this paper, focuses on supporting *library* and *client* developers in the process of library evolution. DEPREWRITER allows library developers to annotate deprecated methods with transformation rules that are used to update client systems either fully automatically or semi-automatically. Our approach is related to the one of Chow and Notkin (Chow & Notkin 1996), who designed a language for expressing code transformations in C. They implemented a semi-automatic tool that allows library developers to annotate changed functions with transformation rules which later are applied statically to update the client code. The DEPREWRITER approach is better suited for highly polymorphic object-oriented languages because it dynamically identifies call sites that need to be updated.

Since 2016, DEPREWRITER is used in production in the Pharo distributions: Pharo 6, 7, 8, and 9 alpha. Pharo[1] is an open-source dynamically-typed reflective, object-oriented language inspired by Smalltalk (Black et al. 2009). Its latest stable release, 8, includes 667 packages. Pharo has more than 3000 external projects, 220 forks on GitHub, and 121 committers, among which there are 12 core contributors. It is managed by an industrial consortium.[2]

In the context of Pharo industrial consortium, we implemented the DEPREWRITER approach to ease software evolution within the Pharo ecosystem using rewriting deprecations. After describing the approach, we answer two categories of research questions: the first category is about the use and flexibility of the rewriting deprecations as used by library providers (*e.g.,* the Pharo consortium) and second category is about the perception of external users of the rewriting deprecations. We then present two validations: an analysis of the DEPREWRITER rules used in the deprecations of Pharo 8 and a user study.

First, we studied the 367 deprecations (not only rewriting

[1] http://www.pharo.org
[2] http://consortium.pharo.org

ones) in Pharo 8, among which we found and analyzed 218 rewriting deprecations. This helped us understand how DEPREWRITER is used in practice and identify its limitations. Second, we performed a user survey and collected information about 46 practitioners, both writing deprecation rules for DEPREWRITER and using libraries with deprecated source code. Of all 46 persons, 28 (60%) report that the rewriting deprecation helped them while 10 the inverse and 8 are uncertain. The analysis of the comments shows that some users really understand the benefit of runtime transformations as a way to scope really precisely the call sites to be rewritten. Others acknowledge the advantages of using unit tests to exercise automatic deprecations, concluding that having a good test coverage helps with migration.

The contributions of the paper are:

- Implementation and description of the run-time rewriting deprecation approach.
- Analysis of the 367 existing deprecations (rewriting or not) of Pharo 8.
- Identification of 33 non-rewriting deprecations that can be turned into rewriting ones.
- Report on an open survey of developers using such an automatic deprecation approach either as client or library developer or both.

The outline of the article is the following one: In Section 2 we discuss the challenges faced to rewrite code in dynamically-typed languages automatically. In Section 3, we introduce Pharo syntax in a nutshell, then we present DEPREWRITER from the perspective of a user and a library developer. Section 4 describes the architecture and an high-level view of the mechanisms used. It also defines the validity conditions for the transformation rules and discusses different scenarios that can be handled by DEPREWRITER. As validation of the approach, Section 5 presents an analysis of the rewriting deprecations of Pharo 8 and the deprecations that were not rewriting ones. Section 6 presents the second part of the evaluation: we report the results of a survey that was completed by 46 Pharo developers. In Section 7, we discuss the key aspects of the current implementation. Since our approach is not bound to Pharo, Section 8 presents sketches of implementation using, for example, AOP. In Section 9, we discuss the limitations of our approach and the future work. After related work, the paper concludes.

2. Problem: the Limits of Static Analysis for Dynamically-Typed Languages

Static analyzers are useful to precisely identify and transform callers of deprecated methods in the context of programming languages with static type information. However, the same techniques do not yield precise results in dynamically-typed languages such as Python, Ruby, Javascript, or Pharo (Suzuki 1981; Milner 1978). Such languages do not have static type information that can be used to drive the static analyses, so static analyzers need to speculate. For example, consider Figure 1 showing the log: methods coming from two different libraries: a logging library and a math library, and a user of them. The maintainers of the logging library decide to deprecate their log: method in favor of a version with an extra argument for the logging level. However, the math log: method remains unchanged. In such a case, a deprecation should be notified only for the logger usages and not for the math usages.

```
Math >> log: aNumber [
   "logarithm implementation"
]

Logger >> log: aMessage [
   "Deprecated in favor of log:level:"
]

User >> main [
   m := Math new.
   logger := Logger new.
   logger log: (m log: 8)
]
```

Figure 1 The limits of static analyses. An analyzer cannot simply statically determine which call to log: corresponds to the deprecated method.

Due to the absence of precise type information, a static analyzer cannot determine which of all the existing calls to log: corresponds to the deprecated method. As shown in the example, a single client method might have many calls to log:. The deprecation not only needs to identify the caller method but the correct call sites inside the method.

This situation is exacerbated in presence of highly overridden methods. For instance, in Pharo 8, the message name is sent 3109 times and implemented in 346 classes, isEmpty has 1595 senders and 103 implementors.[3]

In addition, inheritance and polymorphism across different hierarchies produce code where a single user ends up using multiple implementations during execution.

The research community has proposed to use type inference for dynamically-typed languages (Suzuki 1981; Furr et al. 2009; Ren & Foster 2016; Spoon & Shivers 2004; Pluquet et al. 2009; Passerini et al. 2014) or to use Dynamic Type information collected by the Virtual Machine to get concrete types (Milojković et al. 2016). Such type inferencers often do not cover the whole language (Suzuki 1981) or are not applicable to large codebases (Spoon & Shivers 2004). Type speculation combined with runtime statistics is used to implement speculative Just in Time compilers and speculative optimizers (Hölzle et al. 1991). However, when such information is not available, polymorphic messages force developers to manually identify and replace the correct deprecated calls, leading to the introduction of bugs.

In this paper, we present an automatic deprecation-rewriting approach using runtime information to identify the correct call sites to rewrite. Our approach is based on call-stack navigation and program transformation at runtime. This approach has been

[3] We did not count methods such as initialize or printOn: because they are invoked on the receiver, and static analysis using this fact can identify the correct calls.

used effectively in released versions of Pharo used on deployed productive systems since 2016.

3. Rewriting Method Deprecation Illustrated

We present the Pharo syntax in a nutshell then we start with an example showing how late-bound program transformations are applied to method deprecations. In this section, we illustrate two scenarios of using DEPREWRITER: from the perspective of a library developer and from the view of a client system developer.

3.1. Pharo syntax in a nutshell

To help readers follow the code snippets we present briefly the syntax of Pharo.[4] In Pharo everything is an object that receives messages. Literal objects are created by the parser: strings, symbols, numbers, booleans, nil, literal arrays are literal objects. Other objects are created by sending message new to a class. In addition, lexical closures are defined using [:param | body] syntax.

The following table lists the reserved syntactic constructs of the language.

Literal objects & reserved syntactic constructs	
"comment"	
true, false	the Boolean objects
nil	the undefined object
'string'	sequence of characters
#symbol	unique string
$a	a character
12 2r1100 16rC	twelve (decimal, binary, hexa)
3.14 1.2e3	floating-point numbers
#(#abc 123)	literal array with the symbol #abc and the number 123
{ 'abc' . 3 + 2}	dynamic array built from 2 expressions
#[123 21 255]	byte array
| foo bar |	declaration of two temporary variables
var := *expr*	assignment
exp1. *exp2*	period - statement separator
[:param | *expr*]	lexical closure with a parameter
self and super	receiver of the message with different method lookups
a unary	Unary message sent to a
a + b	Binary message sent to a with b as argument
a at: #key put: val	keyword-based messages equivalent to a.atput(#key,val) in C
a foo ; bar	message cascade (;). All messages (foo, bar) are sent to the cascade receiver (*i.e.,* a)
<message>	method annotation
^ *expr*	caret - return/answer a result from a method

Messages are central to Pharo syntax. All control flow behavior (conditional, loops, iterators) are expressed using messages sent to objects or closures. There are three kind of messages: *unary*, *binary*, and *keyword-based* messages: Unary messages are messages without argument (*e.g.,* dict keys, x class). Binary messages are messages with one argument and a non alphanumerical selector (*e.g.,* 1+2, 1/10). Keyword-based messages are messages with one or more arguments. The arguments are placed within the selector: aDict at: #key put: 33 will execute the method whose selector is at:put:. The equivalent in C-like syntax is aDict.atPut(#key,33). Unary messages takes precedence over binary, and binary over keyword-based messages. Here is the definition of the method slowFactorial defined on the Integer class.

```
1 Integer >> slowFactorial [
2    "Answer the factorial of the receiver."
3    self = 0 ifTrue: [^ 1].
4    self > 0 ifTrue:
5          [^ self * (self - 1) slowFactorial ]
6 ]
```

3.2. Library Maintainer Perspective

Now we explain what a library maintainer should do to deprecate a method. Framework or library maintainers mark the given method as deprecated calling the special method deprecated: transformWith:. This method has two arguments. The first one is a message used to provide documentation about the deprecation. It is shown to the client of the library when the deprecation is notified or logged if the deprecation is applied silently. The second argument is a transformation rule. As DEPREWRITER performs an automatic rewriting of the call site, the library maintainers express the rewriting of the call sites.

Figure 2 shows how to specify such a deprecation to the method log: of the class Logger from the previous example.

```
1  Logger >> log: aMessage [
2
3    "The deprecation definition"
4    self
5      deprecated: 'use #log:level: instead'
6      transformWith: '`@rec log: `@argument'
7        -> '`@rec log: `@argument level: #info'.
8
9    "The body of the method"
10   ^ self log: aMessage level: #info
11 ]
```

Figure 2 An example of deprecation and its automatic rewriting behavior. The method log: is deprecated and callers are rewritten into log:level:. `@ defines variables: rec and argument. The last line is the method body.

- Line 1 defines the signature of the deprecated method as usual.
- Lines 4 to 7 define the deprecation and its companion transformation rule
  - Line 5 defines the text that we will use to log the deprecation or to display it to the programmer.
  - Line 6 starts the definition of the transformation rule delimited by the keyword transformWith:. The *antecedent* of a rule identifies the AST node that will be replaced. The string `@rec log: `@argument defines variables for the receiver and the parameters of the deprecated call. The antecedent should match the

[4] For a more detailed introduction into the syntax of Pharo, please visit https://pharo.org/documentation.html

method signature (same name and number of parameters). Here, the developer named the two variables rec and argument using `@.
  - Line 7 defines the *consequent* of a rule, *i.e.,* the resulting AST nodes of the transformation. It specifies that nodes matching the *antecedent* should be replaced by the nodes of the consequence expression with the matched variables expanded. Here the consequent is `@rec log: `@argument level: #info. It means that any expression *e.g.,* expr1 log: expr2 will be rewritten as calls to expr1 log: expr2 level: #info.

– Finally, after the deprecation definition, Line 10 is a call to the new API. Note that this is up to developers to decide if they want to keep the old deprecated call or immediately invoke the new API. Practically, calling the new API is better since it reduces the number of calls to the old deprecated functionality.

Transformation rules are written using an embedded parse-tree rewriter DSL, part of the Refactoring Engine developed by J. Brant and D. Roberts (D. Roberts et al. 1996, 1997; D. B. Roberts 1999; Renggli 2010). This DSL and its syntax are not part of the contributions of this article. We use it since it was available. Here the `@ defines metavariables that refer to the receiver and the message arguments. The language for expressing transformation rules will be discussed in more detail in Section 4.2. In particular, if the arguments of the log: method contain other log: messages.

3.3. Library Client Perspective

From the perspective of a user of a deprecated method, the process is simple. Developers execute their application, for example, by invoking the main program or by running its tests. At runtime, calls to deprecated methods are rewritten using the transformation rules, and then the application continues its execution. In all cases, the deprecated method is invoked and executed normally. Code transformation of the client method happens as a side effect of its execution.

At runtime, when the deprecated method is executed, the following steps occur:

1. DEPREWRITER is called before the actual body of the method is executed.

2. It accesses the caller method and the exact call site (*i.e.,* the exact location in the text of the method call) where this method was called from - Our implementation introspects the call stack, but alternate approaches such as AOP are possible.

3. It then triggers a code rewriting using as input the transformation rule, the method to rewrite, and the position of the call site in the source code. The caller method is recompiled on the fly.

4. Once DEPREWRITER rewrites the caller's call site, the execution continues in the body of the deprecated method, calling the new non-deprecated code.

In the Logger example (see Figure 2), as soon as the client system calls method Logger >> log:, three things happen: (1) the deprecation warning is signaled (warnings can be disabled); (2) the transformation rule rewrites the client code at the call site and replaces the method call to log: with a call to log:level:, then the method is recompiled; (3) execution of the log: method is resumed, and the last line ^self log: aMessage level: #info is executed, thus calling the correct method.

All automatic code transformations are applied to the original source code. Finally, developers review the changes and decide which ones to keep and version them using traditional tools.

By default, transformations are applied to client code automatically. However, DEPREWRITER is configurable to only show a warning instead.

4. DEPREWRITER

The proposed solution has the responsibility to perform three main tasks.

– Detect the correct callers to the deprecated methods, identifying the exact call sites (for this it accesses the source code to rewrite (Section 7.1).
– Provide a way for the library developers to express the transformation rules and then apply these rules to the identified call sites (Section 4.2).
– Provide the client developer a way of selecting the automation level of the transformations. We consider a key point of DEPREWRITER the ability to configure how much it is affecting the programming ways of the client developer (Section 4.3).

4.1. General Architecture in a Nutshell

Our approach works as follows (See Figure 3):

1. The deprecated method is marked with deprecation and its runtime transformation as shown by (1) in Figure 2.

2. During program execution, the execution of the deprecated method raises an exception (2) in Figure 2.

3. DEPREWRITER captures the exception (3) in Figure 2.

4. When configured to rewrite on the fly, DEPREWRITER identifies the exact call site in the calling method; and it rewrites the method using the deprecation transformation rules ((4) in Figure 2).

5. The execution of the deprecated (halted by exception) method is resumed. The body of the deprecated method is executed ((5) in Figure 2): in most cases, the execution simply calls the new API.

6. From then on, the next execution of the rewritten method will execute the transformed code, and as such, the deprecated method will not be invoked from that call site anymore.

Section 7 hereafter describes the key implementation points.

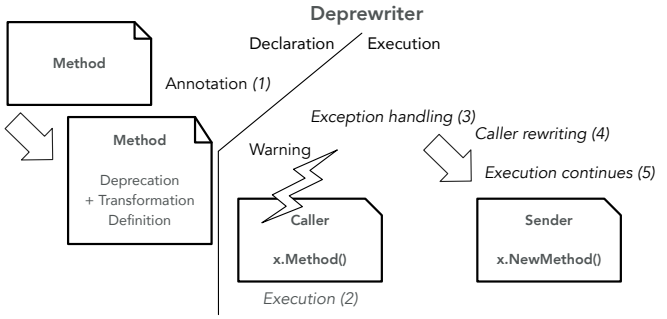

Figure 3 Two moments of rewriting deprecation: declaration and execution. At execution, deprecated method callers are rewritten and execution continues.

4.2. Transformation Engine

To transform methods invoking the deprecated API, we use parse tree pattern rewriting as implemented in ParseTreeRewriter developed by J. Brant and D. Roberts as part of the Refactoring Browser (D. Roberts et al. 1996, 1997; D. B. Roberts 1999). Note that we used this mechanism as it is available and allows developers to express transformations in a syntax close to the ones they are used to. Having a parse tree matcher is not central to our approach: alternate solutions to express the rules and edit the caller methods could be applied.

ParseTreeRewriter is a tree pattern matcher. First it identifies the node to be transformed (source) and then how such node should be recombined during the transformation (target). We briefly present the key aspects of the parse tree matcher. The backquote character ` creates a variable. Several options following the variable declaration can be used to specify the search:

- ` defines a variable. `receiver foo matches x foo, OrderedCollection foo, or self foo.
- `@ matches any subtrees. `@rec foo matches self foo (with rec = self), self size foo (with rec = self size or (x at: 2) foo (with rec = (a at: 2)).
- `. matches any language statement (assignment, return, messages,...).
- `# matches literals (string, boolean, number, symbol in Pharo). `#lit size matches 3 size, 'foo' size, true size.
- { } is used to match the enclosed code (see line 4 in the code below).

The following unit test of class RBParseTreeRewriter illustrates a simple example showing how three-element dynamic arrays (delimited by { and }) are rewritten as static literal arrays.

Lines 3 to 5 configure the rewriter. The first argument of the call to replace:with:, the expression {`@first. `@second. `@third} specifies the kind of dynamic array that we want to transform. The second argument defines the creation of a static array with the same elements.

Lines 7 to 9 check that the rewriter can effectively transform the code and reformat the results on a specific dynamic array: here {(1 @ 255). (Color lightMagenta). 3}

Finally lines 10 to 12 verify that the result is correct.

```
1  testRewriteDynamicArray [
2    | newSource |
3    rewriter := RBParseTreeRewriter new
4      replace: '{`@first. `@second. `@third}'
5      with: 'Array with: `@first  with: `@second
       with: `@third'.
6
7    newSource := (rewriter executeTree:
8                    (self parseRewriteExpression:
       ' {(1 @ 255). (Color lightMagenta). 3}'))
9      ifTrue: [ rewriter tree formattedCode ].
10   self
11     assert: newSource
12     equals: 'Array with: 1 @ 255 with: Color
       lightMagenta with: 3'.
13 ]
```

One particular aspect of ParseTreeRewriter is that it cleverly extends the syntax of the language. Programmers can simply take their code and annotate it with the specific characters, and they obtain a transformation rule. The Pharo consortium decided to use this aspect to ease the adoption by practitioners so that they do not have to learn another language or framework to deprecate their code (Rizun et al. 2015). Alternate extensible solutions have been proposed to support programmers during the definition of transformation patterns (Rizun et al. 2016), but they require the use of a tool to help generating the rule.

4.3. Configuring the rewriting process

The user has the possibility to configure the rewriting process:

- *automatic rewrite*: all the rewriting deprecations automatically rewrite their callers.
- *warning*: developers are warned interactively that their code is invoking deprecated methods, but the caller will not be automatically rewritten.
- *do not do anything*: with this configuration, the deprecated methods are executed normally, and the rewriting behavior or warnings are not effective.
- *logging*: with this, developers can log the deprecated calls (and transformations done). This option is orthogonal to the others and can be turned on-off as needed.

4.4. The Validity of Transformation Rules

Over the years, Pharo codebase has accumulated many transformation rules. But since there was no good documentation of deprecation rewritings, some of those rules make little sense and can be considered invalid. In this section, we define the validity conditions for transformation rules that appear in deprecations:

1. **The antecedent must capture a single message send which must be the same as the deprecated selector**. Transformation rules are arguments of the deprecation messages. Every rule is expected to replace the deprecated message send, captured by the expression in the antecedent, with the valid replacement that is defined by the consequent. Therefore, the antecedent must capture *exactly one message* which must be *the same as the one that is being deprecated*. In other words, the left-hand side

Type	Deprecation selector	Introduced in	Senders
Non-rewriting (149)	deprecated: anExplanation	≤Pharo 2	113
	deprecated: anExplanation on: aDate in: aPharoVersion	≤Pharo 2	36
Rewriting (218)	deprecated: anExplanation on: aDate in: aPharoVersion transformWith: aRule	Pharo 6	4
	deprecated:transformWith: aRule	Pharo 6	214
	deprecated: anExplanation on: aDate in: aPharoVersion transformWith: aRule when: aCondition	Pharo 7	0
	deprecated:transformWith: aRule when: aCondition	Pharo 7	0
Total:			**367**

Table 1 Six deprecation selectors available in Pharo 8 together with number of senders.

expression of a transformation rule must always be in the form `@rec deprecatedSelector: `@arg (with any number of arguments).

2. **Neither antecedent nor consequent can be empty**. If the antecedent is empty, then the transformation rule cannot capture anything and will never be applied. In addition, a consequent cannot be empty because we do not consider it as a good practice to automatically delete method calls from the client code without replacement.

3. **Antecedent and consequent must be different**. If they are not, then the transformation rule has no effect because it replaces the captured message send with exactly the same message send.

4.5. Deprecation Scenarios and How they are Supported

In this section, we discuss multiple scenarios that can lead to deprecations. For every scenario, we specify whether or not it can be handled by DEPREWRITER and explain the cases when additional information is required to automatically rewrite the client code. The list is not exhaustive but it covers the majority of scenarios that we have encountered in practical applications (see Section 5) and partially inspired by the lists of refactoring operations that were analysed by Murphy-Hill *et al.* (Murphy-Hill et al. 2011) and Dig *et al.* (Dig & Johnson 2006).

Rename method — method renaming is the most common refactoring operation (Murphy-Hill et al. 2009). If the renamed method is part of a public API, it is a good practice to add a deprecated method with oldMethodName that calls the newMethodName. Renamed methods retain the same received and same list of arguments. This means that the deprecations introduced as part of method renaming can always be supported with transformation rules in the form `@receiver oldMethodName: `@arguments → `@receiver newMethodName: `@arguments.

Remove argument(-s) — similar to renaming, when one or multiple arguments are removed from a method, the old method name can be deprecated.[5] This scenario can always be supported with a transformation rule in the form `@receiver oldMethodName: `@oldArguments → `@receiver newMethodName: `@newArguments where `@newArguments is a subset of `@oldArguments.

Add argument(-s) — this scenario is similar to the previous one, however, the `@newArguments is a superset of the `@oldArguments. To create a transformation rule that could automatically update method calls in the client code, developers who introduce the deprecation must provide default values for the new arguments. In many cases this is not possible.

Change receiver — when the receiver is changed it means that either the method was moved to a different class (if method name is the same) or that another method from a different class is called instead (if method name is different). This scenario can be handled by a rule only if new receiver is one of the arguments of the old method call, a literal value, or a global variable (*e.g.,* a class name). In other cases, when new receiver must be instantiated, transformation rules are not enough to automatically rewrite the client code. Here is an example of changing the receiver with a transformation rule: `@gradebook gradeOf: `@student → `@student grade. A possible side effect of such a replacement is that a receiver can be initialized and never used.

Split method — a method call needs to be replaced with two or more method calls. Each method can be called from the return value of the previous method: f().g().h(), sent as an argument to the previous method: f(g(h())), or sent as a cascade message — a special syntax in Pharo that allows one to send multiple messages to the same receiver (Black et al. 2009). For example, `@receiver evaluate: `@argument → `@receiver statements: `@argument; evaluate (both messages in consequent are sent to the same receiver). The *split method* scenario can be handled with a transformation rule if all receivers and arguments in the consequent appear in the antecedent or are literal values or global variables.

Delete method — it is common to delete a method without replacement. In this case, the method can be deprecated

[5] Since in Pharo arguments are inserted between the parts of a method name, it is not possible to change the number of arguments without also changing the name of a method.

but there is no transformation rule that can rewrite the client code.

Push down — if the method is pushed down into the subclass, it may be deprecated in the superclass, however, one can not automatically rewrite the callers because it is not clear which of the subclasses should be instantiated instead (as well as how to instantiate the subclass).

Deprecate class — when the class is deprecated, all of its methods can be deprecated as well. In this case, the transformation rule can not be introduced because it is not only the method call that must be replaced but also the code that instantiates the receiver.

Complex replacement — in addition to the scenarios described above, there are also more complex situations when a method deprecation would require client developers to introduce changes into multiple locations in their code (can not be handled with the current implementation of DEPREWRITER) or to replace a method call with a more complex expression that may include block closures, streams, etc. (can be handled if all the variables are known). It is also possible that a complex replacement requires client developers to make extra decisions and introduce different fixes depending on the specific situation.

5. Analysis of Deprecated Methods in Pharo 8

DEPREWRITER has been introduced into Pharo at version 6.0 (in 2017). During the last four years, it has been used by the Pharo community to supply method deprecations with rules that can automatically rewrite client code. To better understand how the DEPREWRITER is used in practice, we have analysed the 367 method deprecations collected from Pharo v8.0,[6] the latest stable version of the Pharo Project.[7]

Here are the research questions we want to answer:

RQ1 **Adoption.** How widely adopted are the rewriting deprecations?

RQ2 **Flexibility.** What are the different types of deprecation scenarios that can be supported by DEPREWRITER?

RQ3 **Limitations.** What are the more complex scenarios that can not be supported by the current implementation of the DEPREWRITER?

In this section, we report the results of our analysis.

[6] The source code of Pharo v8.0 was loaded from an open source repository https://github.com/pharo-project/pharo at commit *bbcdf97*

[7] Pharo is a programming language and an IDE written entirely in itself. This can be a source of confusion. In this section, we analyse how rewriting deprecations, introduced into Pharo (an open-source project with over 140 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.

Cleaning the Data. Before analysing the deprecated methods, we performed several data cleaning steps to only retain those deprecations that are relevant. First, we have removed 8 deprecated methods that were used only for testing (*e.g.,* deprecatedMethod1, deprecatedMethod2). We also removed 2 deprecations that contained an invalid transformation rule, based on the validity criteria discussed in Section 4.4. Finally, we have found one case that was not a real method deprecation but a workaround to deprecate a pragma (a static method annotation; Pharo does not support pragma deprecation). As a result of this step, out of 378 deprecated methods found in the Pharo 8 image, only 367 were retained for analysis.

RQ1. How widely adopted are the rewriting deprecations? To answer this question, we calculate the proportion of method deprecations in Pharo v8.0 that contain transformation rules and the number of different people who introduced them into the source code.

In Pharo, there are six selectors that can be used to deprecate a method. In Table 1, we provide the list of those selectors along with the Pharo version in which they were introduced and the number of senders (the number of times the selector is used) in Pharo v8.0. The first two selectors exist in Pharo since v2. They allow developers to mark a method as deprecated and provide an explanation message that will be displayed in a warning dialog when a deprecated method is invoked. The second selector also allows one to specify the date and library version at which the method was deprecated. Method deprecations that are declared with those two selectors do not contain a transformation rule, which is why we call them the *non-rewriting deprecations*. Those deprecations are similar to the ones declared in Java using the *@Deprecated* annotation or the *@deprecated* Javadoc tag. The other four selectors were introduced in later versions of Pharo. They allow developers to specify the correct replacement for a deprecated method call in a form of a transformation rule that can automatically rewrite the call-sites inside the client code. We call deprecations that were declared with those selectors the *rewriting deprecations*. The last two rewriting selectors also allow one to specify the condition that will be checked before applying the transformation rule. As can be seen in Table 1, out of the 367 deprecated methods that we found in Pharo 8, 149 deprecations are non-rewriting (41%), and 218 are rewriting (59%). This means that the majority of method deprecations in Pharo contain the transformation rules.

For every method deprecation, we have found the commit in which it was introduced into the project. This allowed us to identify the author of each deprecation. 16 out of 367 deprecations remain in Pharo since before v6.0.0. They were introduced with a different version control system, which makes it hard to identify the authors. By analysing the other 351 deprecations, we have found that they were introduced by 15 different developers. Rewriting deprecations were introduced into the Pharo project by 8 different developers. This means that the majority of developers who manage deprecations in Pharo project are familiar with DEPREWRITER.

RQ2. What are the different types of deprecation scenarios that can be supported by DEPREWRITER? To answer this ques-

tion, we analyzed the 218 rewriting deprecations that we have extracted from Pharo v8.0. Each one of those deprecations contains a transformation rule in the form *antecedent* → *consequent*. The antecedent is a left-hand side of the rule, which is an expression that matches the piece of deprecated code that needs to be replaced. The consequent is the right-hand side of the rule which defines the replacement.

We classified the transformation rules according to the deprecation scenarios proposed in Section 4.5. The summary of this classification is presented in Table 2. The first column of the table contains the list of scenarios and the second column specifies whether the deprecation from each scenario can be supplied with a transformation rule. The option *yes** means that in some cases it is possible to express the replacement with a transformation rule, while in the other cases, additional information or a manual fix may be needed. The third column of the table contains the number of rewriting deprecations corresponding to each scenario that were found in Pharo 8. The last two columns of the table will be discussed in the rest of this section.

One can see that *Rename method* is the most common scenario for which the developers of the Pharo Project use transformation rule. 179 out of 218 rewriting deprecations (82%) express method renaming. Developers also use transforming deprecations for other scenarios. This includes 28 *Split method* rules that replace a method call with multiple ones and 5 complex rules, all of which replace a method call with an expression containing a block closure. Examples of the transformation rules that we have collected from Pharo 8 can be found in Table 3.

Such a diverse collection of rules demonstrates the flexibility of DEPREWRITER.

RQ3. What are the more complex scenarios that can not be supported by the current implementation of the DEPREWRITER? In this section, we explore the non-rewriting deprecations and try to understand (1) if some of them can be automated using DEPREWRITER; (2) what makes deprecations hard to automate. In fourth column of Table 2, we present the number of non-rewriting deprecations that we found in Pharo 8, classified by the deprecation scenarios presented in Section 4.5. Those deprecations were introduced into the image without a transformation rule. We analysed each one of them to see if a rule is impossible in that case or if it could be added but the developers who deprecated the method missed the opportunity to write a rule. For each non-rewriting deprecation that could have a rule, we introduced it. The fifth column of the table presents the number of rules that we introduced to turn non-rewriting deprecations into the rewriting ones.

As can be seen in the table, 69 non-rewriting deprecations belong to either the *Delete method*, *Push down*, or *Deprecate class* scenario, which means that the removed method does not have a replacement and the transformation rule can not be introduced. Those are 46% of all non-rewriting deprecations and 19% of all deprecations that we found in Pharo Project. In Section 4.5, we claimed that *Method rename* and *Remove argument(-s)* scenarios can always be supported by rules. This can be seen in Table 2. We proposed rules for all non-rewriting deprecations from those categories. We could also introduce a rule for one deprecation out of 7 that belong to the *Add argument(-s)* category. In that case, we used an empty literal value as default argument. In other 6 cases, default argument is either impossible or must be defined by the experts.

As can be seen in Table 2, for 2 out of 43 non-rewriting deprecations that represent the complex replacement we introduced a transformation rule thus turning them into the rewriting deprecations. The other 41 complex deprecations are of particular interest for our study because they are the non-trivial scenarios that indicate the limitations of DEPREWRITER. 4 of those deprecations require clients to override an abstract hook method and 1 deprecation requires client to implement a subclass. The current implementation of DEPREWRITER only deals with method calls and the support for object oriented rewriting could be an important direction of our future work. 3 deprecations propose different replacements to the clients depending on certain conditions. 17 deprecations require complex changes in multiple locations of client code (*e.g.,* initialization of objects that will be passed as arguments, removing the initialization of objects that are no longer needed, different treatment of return values, etc.). Such replacements are not easy to automate. They must be performed manually by client developers. Finally, 16 non-rewriting deprecations were either poorly documented or had very complicated replacement instructions that require an expert to understand and apply them.

Pull Requests with Proposed Rules. Out of 149 non-rewriting deprecations, we have identified 33 cases (22%) when the transformation rule was possible but developers missed the opportunity to introduce it. We have added the rules to those deprecations and submitted them as pull requests to the development version of Pharo 9. Out of those 33 deprecated methods, 10 have already been removed from Pharo 9, 6 methods have already been supplied with the transformation rules (same as the ones we have proposed), and 1 deprecation has been reverted (method that was marked as deprecated will not be removed after all). The other 16 transformation rules that we have submitted as pull requests were merged into Pharo 9.[8] In Appendix A, we list some of the rules that we have proposed.

Analysis Conclusion. The analysis presented in this section demonstrates how developers use transforming deprecations in real cases of library evolution. It also helps us identify the strengths and limitations of DEPREWRITER. Below we list the main conclusions of our analysis.

– **Adoption of transforming deprecations by developers.** Out of 367 deprecated methods that we found in Pharo 8, 218 methods (59%) use transformation rules to automatically rewrite their users. Those deprecations were introduced by 8 different developers, which is more than half of the total number of developers who deprecated methods in Pharo 8. This demonstrates that the developers of Pharo project have adopted the technology discussed in this paper.

[8] Pharo is an open source project with more than 150 contributors. Each pull request must be approved by one or multiple reviewers who are members of the core team and are different from the person submitting the PR.

Deprecation Scenario	Can be expressed with a rule?	Rewriting deprecations	Non-rewriting deprecations	Rules introduced by us
Rename method	yes	179	24	24
Remove argument(-s)	yes	3	1	1
Add argument(-s)	yes*	1	7	1
Change receiver	yes*	2	0	0
Split method	yes*	28	5	5
Delete method	no	—	52	—
Push down	no	—	13	—
Deprecate class	no	—	4	—
Complex replacement	yes*	5	43	2
Total:		218	149	33

Table 2 Different scenarios that may require method deprecation. In the second column, we specify if this scenarios can be expressed with a transformation rule. *"yes*"* means that in some cases the rule is possible, but in other cases, additional information may be required. The third column contains the number of rewriting deprecations found in Pharo image. The fourth column contains the number of non-rewriting deprecations related to each scenario — the ones that do not contain transformation rules. The last column contains the number of rules that we introduced and submitted as pull requests.

- **Flexibility of transforming deprecations.** The analysis shows various scenarios of method deprecations that can be supported with DEPREWRITER. In most cases, developers use the tool to deprecate methods that were renamed. They also use it to express more complex rules such as removed or added arguments, splitting one method into multiple methods, etc.
- **Limitations of transforming deprecations.** We have identified several situations that can not be covered by the DEPREWRITER or can not be expressed by the language of transformation rules. Those cases can help us understand how we can improve the mechanism for expressing and applying the rules — a topic that we plan to cover in future work. The most interesting case is the lack of support for object-oriented rewriting (overriding abstract methods, introducing subclasses, adding methods, etc.).

6. User Survey

Making a sound evaluation of DEPREWRITER is difficult because it is often applied to the private code of developers using Pharo. In addition, developers can have different development processes. To understand how DEPREWRITER is used and perceived by developers in the Pharo ecosystem, we performed an open survey. The participants of the survey are developers programming in Pharo. We did not select the participants based on their practices or use of DEPREWRITER. From that perspective, they can either know or not the existence of DEPREWRITER, and be either *client* or *library* developers as defined at the start of this article.

With this survey, we set the following research questions:

RQ4 How is DEPREWRITER used by client developers who are affected by deprecation rewriting?

RQ5 How is it used by the developers who introduced the rewriting deprecations into their systems?

RQ6 What are the configurations of DEPREWRITER that are preferred by users?

The list of survey questions is available in Appendix B. Because of the limited space, we cannot provide all the answers to the open questions. Once the paper is accepted, we will release the data on a public archive. We received the answers from 46 developers.

Population Characterization. We asked developers to characterize their development effort into *application*, *library* and *frameworks* (framework implying some sort of extensibility). The two last choices embed the idea that the developed software is used by other developers. Hence library and framework developers present more concerns about change impact. Table 4 shows that we have a large part of participants making applications (93%, 43 out of 46), and we still have many working on library development (71%, 33 out of 46). In addition, 32 reported to do application *and* library development. Such numbers are not surprising; because library developers are sensitive to deprecations and the impact of their changes. So, it is normal that they got a stronger incentive to participate in the survey.

Then we asked how often they migrate their software. A release cycle of Pharo is one to one and a half years. And, this is a main source of migration. The data in Table 5 show that 58% of developers (27 out of 46) are often migrating their code to newer versions of Pharo. A couple of developers mentioned in the optional comments that they always migrate to bleeding-edge versions.

RQ4. How is DEPREWRITER used by client developers who are affected by deprecation rewriting? We asked several ques-

Scenario	Antecedent	Consequent
Rename method	‘@receiver getAction	‘@receiver action
	‘@receiver selectedPage: ‘@statements1	‘@receiver selectPage: ‘@statements1
	‘@receiver keyword: ‘@arg1 arguments: ‘@arg2	‘@receiver selector: ‘@arg1 arguments: ‘@arg2
Remove argument(-s)	‘@receiver interpretASpec: ‘@statements1 model: ‘@statements2 selector: ‘@statements3	‘@receiver interpretASpec: ‘@statements1 presenter: ‘@statements2
	‘@receiver addSelector: ‘@statements1 withMethod: ‘@statements2 notifying: ‘@statements3	‘@receiver addSelector: ‘@statements1 withMethod: ‘@statements2
Add argument(-s)	‘@receiver getEnv: ‘@arg	‘@receiver at: ‘@arg ifAbsent: [nil]
Change receiver	‘@receiver write: ‘@statements1	‘@statements1 putOn: ‘@receiver
	‘@rec asIcon	self iconNamed: ‘@rec
	‘@receiver openNativeBrowserOn: ‘@arg	NativeBrowserOpenVisitor openOn: ‘@arg
Split method	‘@receiver listItems	‘@receiver model items
	‘@receiver commentsAt: ‘@argument	(‘@receiver compiledMethodAt: ‘@argument) comments
	‘@receiver evaluate: ‘@statements1 in: ‘@statements2 to: ‘@statements3 notifying: ‘@statements4 ifFail: ‘@statements5	‘@receiver source: ‘@statements1; context: ‘@statements2; receiver: ‘@statements3; requestor: ‘@statements4; failBlock: ‘@statements5; evaluate
Complex replacement	‘@receiver whenSelectionIndexChanged: ‘@argument	‘@receiver selection whenChangedDo: [:selection | ‘@argument value: selection selectedIndex]
	‘@receiver whenSelectedItemChangedDo: ‘@argument	‘@receiver whenSelectionChangedDo: [:selection | ‘@argument cull: selection selectedItem]

Table 3 Examples of the transformation rules extracted from Pharo 8.

Development type	Yes	No	Uncertain
application	43 (93%)	0	3 (7%)
library	33 (71%)	6 (13%)	7 (15%)
framework	24 (52%)	13 (28%)	9 (19%)

Table 4 Survey: *"What kind of software do you maintain?"*

Frequency	Very often	Often	From time to time	Not often	Never
	8 (17%)	19(41%)	13 (28%)	6 (13%)	0

Table 5 Survey: *"How often do you migrate your software to newer versions of its dependencies?"*

tions about the migration and the perception of the current automatic migration approach based on rewriting deprecations. The first set of questions focuses on rewriting deprecations from a client developer perspective.

- To the question *'Do you see value in having tools to help with code migrations?'*: Unsurprisingly, all the participants agreed that they see a value.
- To the question *'Do you know Pharo's support for automatic deprecation rewritings?'* 35 (76%) reported that they knew the existence of rewriting deprecation, 4 did not, and 7 were not certain.
- *'Did automatic deprecation rewritings help you in a migration?'* This question is an important one. Table 6 shows that 28 (60%) participants of 46 acknowledged that the rewriting deprecations helped them, while 10 mentioned otherwise and 8 are uncertain.

We then proposed open questions to be able to understand more precisely what the practitioners meant. We present verbatim some of the answers that were added in the comment

	Yes	No	Uncertain
Knowing deprewriter	35 (76%)	4 (8%)	7 (15%)
Did it help migrating?	28 (60%)	10 (21%)	8 (17%)

Table 6 Survey: *"Do you know Pharo's support for automatic deprecation rewritings?"* and *"Did automatic deprecation rewritings help you in a migration?"*

field related to the question *'How did automatic deprecation rewritings help you in a migration?'*. We selected the most representative or critical ones as well as the ones suggesting improvements.

- *'I help me to migrate API in the case where old and new API are overlapping.'*
- *'My software migrated on its own without me having to do anything! If anything, maybe it's too invisible. Packages show up dirty without a clear cause. Although I can't think of an obvious solution except maybe a warning with a setting, like deprecations themselves.'*
- *'They are useful to find all places where a deprecated method is really called. Just they can also be a bit annoying. Especially as they are done automatically, sometimes, I was seeing code changes, and I was not sure if I did that or if it was done by an automated refactoring.'*
- *'It shows the right way to use the new API.'*
- *'I observed it when loading an old package. It was cool. Too often, we lose code from rot as the base image moves in a new direction, and we have old code that will no longer load.'*
- *'Running the tests transformed the code to the new protocol'*
- *'I often can afford to immediately delete an old method and rewrite all users to a new one. From time to time, I use rewrite method to change callers of a method with a common name. In other words, I use it in cases where simple method rename action is difficult to filter by a scope.'*
- *'I remember the automatic rewritings of some Spec2 (or Spec?) rules that migrated my tool without efforts from my side other than code reviewing quite fast before creating the commits. Given that Iceberg supports creating commits from part of the working copy changes, it was easy for me to untangle the changeset between migration changes vs. intended changes (*e.g., *fixing a bug).'*

Analyzing such comments, three main points are noticeable.

- First, the majority of surveyed users appreciate the automation offered by the process.
- Second, some users would like to have more control over the process. It is true that getting a package with changes that are done automatically is surprising without a priori notice.
- Third, and more interesting, some users understand the main strength of DEPREWRITER: the approach helps them to deal with heavily used (megamorphic) methods and that the deprecation transformation at runtime provides a precise scope that only rewrites the executed method.

RQ5. How is DEPREWRITER used by the developers who introduced the rewriting deprecations into their systems? We also asked the practitioners about their experience introducing the rewriting deprecations.

- To the question *'Did you write your own automatic deprecation transformation rules to help migrate your users?*: 16 participants out of 46 answered *yes"* (34.8%), while 23 answered *"no"* (50%) and 7 (15.2%) did not answer the question, which we interpret as no (See Table 7). The high percentage of writers is probably related to the fact that developers replying to this survey were already interested in the topic. Nevertheless, it is surprising to see that developers outside the core team (which is composed of 12 people) are writing deprecations for their own libraries because neither documentation nor announcement have been made about DEPREWRITER prior to this article.
- To the question *'How easy it was to write a rewrite rule?'*, 2 found the expressions very easy, 1 easy, 9 medium, and 1 difficult, 27 did not answer (see Table 8). The other 3 out of 16 participants who mentioned that they write transformation rules, did not report the difficulty level.
- To the question *'Can you tell us about the cases where you found it impossible to use automatic deprecations?'*, 4 developers indicated that they need support for rewriting on the level of class deprecations, 4 developers wrote that they need object-oriented rewriting that we have discussed in the previous section (rewriting implementors of deprecated abstract hooks, introducing subclasses, etc.), 2 developers requested conditional rewriting. Those answers demonstrate the limitations of DEPREWRITER that can be targeted in the future work (see Section 9).

	Yes	No	Uncertain
Did you write rules?	16 (35%)	23 (50%)	7 (15%)

Table 7 Survey: *"Did you write your own automatic deprecation transformation rules to help migrate your users?"*

	Very easy	Easy	Medium	Difficult	No Answer
How easy?	2 (4%)	7 (15%)	9 (20%)	1 (2%)	27 (59 %)

Table 8 Survey: *"How easy it was to write a rewrite rule?"*

RQ6. What are the configurations of DEPREWRITER that are preferred by users? The next series of questions is related to the default configuration of the automatic rewriter. It should be noticed that the two first questions are not opposite of each other. The deprecation can rewrite and transform the callers while at the same time notify the users with a warning. The

default setting proposed in Pharo is to silently rewrite code. This decision is probably not good to communicate with the users, as some have reported in previous feedback.

- To the question *'A setting allows you to control if a deprecated call is automatically rewritten or not. Do you think automatic deprecations should be rewritten by default?'*: 22 on 46 (47%) report that the default behavior to rewrite the deprecated calls at execution is good, 14 (30%) think the inverse and 10 are uncertain (See Table 9).
- To the question *'Another setting controls whether a deprecation should raise a warning or not when found at runtime. Do you think automatic deprecations should raise a warning by default?'* 24 of 46 (52%) report that raising a warning by default is important, 12 (26%) think the inverse, and 10 (21%) are uncertain (See Table 9).
- To the question *What is your most frequent setup?* Developers uses a mix of configurations. No clear choice jumped out.

	Yes	No	Uncertain
Rewriting by default	22 (47%)	14 (30%)	10 (21%)
Raising a warning	24 (52%)	12 (26%)	10 (21%)

Table 9 Survey: What are the configurations f DEPREWRITER preferred by the developers?

Survey Conclusion. The survey shows that the developers from Pharo community are familiar with the basic functionality of the DEPREWRITER and appreciate the automatic rewriting of method calls. Developers report that they need a better documentation that would inform them on how to write better transformation rules.[9] A better default configuration also is desired, and some tooling easing the understanding of the automatic recompilation should help make the process less mysterious to the users. Finally, a survey demonstrates several shortcomings of DEPREWRITER that can be targeted in the future work (see Section 9).

7. Implementation

We present the key points of the Pharo implementation of the automatic rewriting. However, since the general idea can be applied to other languages. In Section 8, we also present some sketches of possible implementations using exception or AOP (Kiczales et al. 2001; Colyer & A. 2005; Chern & De Volder 2007).

7.1. Call Site Identification using Stack Reification

The call site identification is performed using the reflective capabilities of Pharo to access the execution stack as a chain of linked objects. Pharo's thisContext pseudo-variable creates on the fly an object representing the current C stack frame. This object is causally-connected to the C stack frame (Smith 1984). Each stack frame has the knowledge of how to traverse the stack to create on the fly the caller stack frame. This gives us access to the whole chain of stack frames. Using this feature, we access the stack frame of the caller. Also, the reified stack frame has the responsibility to resolve its activated method. By using this, we are able to access the method to rewrite and its source code.

Deprecation is a Warning. As shown in Listing 1, the execution of a deprecated method will invoke the method deprecated: anExplanationString transformWith: aRule. This method creates an exception, configures it (passes as argument the sender of the deprecated method and the transformation rule) and invokes the method transform. The expression thisContext sender reifies the execution stack frame using the special variable thisContext, and it returns the call-stack frame of the caller of the current call. An important point is that the Deprecation class is a subclass of Warning. Warnings do not stop program execution. They just execute the default signal method when signalled, and then the program execution continues.

It means that in our case, the program executes, a deprecation is raised, and during the exception execution, the caller method is rewritten: when the deprecation warning finishes its execution, the program execution continues as normal, executing the body of the deprecated method. The rewriting happens as a side effect, as we do not perform the on-stack replacement. The rewritten method will be used for the next execution.

```
1 Object >> deprecated: anExplanationString
2   transformWith: aRule [
3       Deprecation new
4           context: thisContext sender;
5           explanation: anExplanationString;
6           rule: aRule;
7           transform
8 ]
```

Listing 1 The method deprecated: transformWith:

Rewriting steps. The transform method does the following (Listing 2):

- Lines 3-4, it checks if it has been configured to transform the caller or not. If the user wants a deprecation exception to be managed normally, the transformation does not happen. The exception is signaled.
- Line 5, the caller method is an identifier. The method contextOfSender is just doing context sender, it means that given the method that invoked the deprecated:transformWith :, it will find the stack element representing the caller call.
- Line 6, some internal usage of the code evaluator are ignored (code snippet executions are mapped to anonymous methods and we do not want to rewrite snippets therefore the implementation has to check that a caller is not a snippet execution).
- Line 7 grabs the program node of the AST representing the caller of the deprecated method.
- Line 8 the transformation rule (called a rewrite rule) is created based on the specifications of the deprecation.

[9] At the time of the article writing, no accurate documentation is available, only some unit tests.

– Lines 9-10 once we have the node and the rule, the method asks if the rule can be applied to the node. This is what is done in the message executeTree:. When the node cannot be rewritten, the default exception is raised.
– Lines 11-12, when the node can be rewritten, it is replaced by the corresponding expression and the method is recompiled. At this point, the currently executed method is still on the stack, and it continues its execution. This means that the rewritten code will only be executed on the next execution.

Note that if we have two calls to the same deprecated method in one method, we will execute this method twice to replace both cases.

```
1  Deprecation >> transform [
2      | node rewriteRule aMethod |
3      self shouldTransform
4       ifFalse: [ ^ self signal ].
5      aMethod := self contextOfSender method.
6      aMethod isDoIt ifTrue: [ ^ self ].
7      node := self contextOfSender
       sourceNodeExecuted.
8      rewriteRule := self rewriterClass new
       replace: rule key with: rule value.
9      (rewriteRule executeTree: node)
10         ifFalse: [ ^ self signal ].
11     node replaceWith: rewriteRule tree.
12     aMethod origin compile: aMethod ast
       formattedCode classified: aMethod protocol.
13 ]
```

Listing 2 The core of the rewriting behavior method transform :.

The implementation is relatively simple. It assembles existing functionality available in the language: exception mechanism, call-stack reification, and a parse tree rewriting engine.

8. Sketches of Possible Alternative Implementations

The approach proposed in this article requires two main features to be implemented in alternative languages and environments: (1) being able to detect the caller method of a deprecated one, and (2) being able to update the code of caller method to use the newer versions of deprecated methods.

To show the generality of our proposed solution, we present in this section possible alternative implementations for both required technical features. Each subsection presents different alternatives to the one used by our implementation. They are presented in order from more specific and powerful to more simple but still useful and generally available.

8.1. Caller Method Detection

When a deprecated method is called, our approach needs to identify the method that has called the deprecated method. Once the caller method is correctly identified, it is updated using the transformation rule expressed in the deprecated method.

Call-Stack Reification. Our proposed solution is implemented by using the call-stack reification (Rivard 1996) mechanism that is present in Pharo. This mechanism allows the inspection and manipulation of the current executed stack. By inspecting the current call-stack, we identify the caller method. The availability of this technique produces a cleaner and simpler implementation but it is not required to implement our approach as we show with the subsequent implementations below.

Aspect-Oriented Programming Pointcuts. Aspect-Oriented Programming (AOP) (Kiczales et al. 1997) pointcuts have the ability to mark deprecated methods to execute caller update logic. AOP pointcuts identify the deprecated methods in a declarative way, without needing to modify the code of the deprecated methods. This information is expressed in metadata that is read by the AOP framework. In our proposed solution, we modify the deprecated method to explicitly start the update process of the caller. Using pointcuts, it is possible to identify deprecated methods without modifying the source code of these methods. The methods are modified by the AOP framework during the weaving process, without modifying the original source code. Also, AOP frameworks allow the developers to mark method call-sites (*e.g.,* making a pointcut every time a given method is called), by doing so it is possible to identify all calling sites to a deprecated method. This technique is applicable to replace our inspection of the current call-stack.

Exceptions with Stack Trace Information. As said, having high-level support for inspecting the call-stack is not needed for implementing our proposed approach. Using exception handling and logging is a possible way to identify the caller method. Using the fact that an exception thrown will capture the current stack trace information, it is possible to extract the caller method. Listing 4 presents a possible pseudo-code Java implementation that throws an exception and processes the stack trace information to extract the method caller (even if it is a String representation).

8.2. Caller Method Update

Once the caller method is identified, this method should be modified following the transformation rule in the deprecated method.

Modifying the Source Code. Our current implementation use tree matching rewriting (D. Roberts et al. 1996, 1997). Although using tree matching rewriting allows us to write a rich set of possible deprecation rules, it is not necessary to implement our approach. Caller method source code might be modified by just manipulating strings. A simpler approach might use regular expression matching and replacing.

Dynamic Software Update Support in Languages. Once the source code is modified, we update the caller method with the new implementation. We use the dynamic software update (DSU) (Sandewall 1978) support that is present in Pharo. This ability to modify running program is not only present in Pharo but it is present in other dynamic languages (*e.g.,* Lisp, Javascript, Ruby, Python).

```
public class CallerIdentifier {

  public void identifyCaller(String aMethodName)
    {
    Exception fakeException;
    StackTraceElement[] stackTrace;

  /* We throw an exception and we catch
  it in the same method, by doing so,
  we force the creation of the exception
  and the logging of the current
  stack trace. */

    try {
      throw new Exception();
    } catch (Exception e) {
      fakeException = e;
    }

    stackTrace = fakeException.getStackTrace();

  /* With the information of the stack
  we obtain the calling method.
  Even if the information is present
  as a string, caller identification
  is a feasible operation */

    return this.lookupCallerOf(aMethodName,
    stackTrace);
  }
}
```

Figure 4 Java pseudo-code showing how to identify caller method by only using existing exception support.

Dynamic Software Support in Tools. In the scenario we are using a language that does not natively support DSU, we still are able to use existing DSU tools for these languages. These tools manage the update of executing code allowing us to modify the caller method. Such tools exist in numerous languages and environments ranging from low-level languages as C (*e.g.,* Kitsune (Hayden et al. 2012), Ginseng (Neamtiu et al. 2006)) to object-oriented languages running in a VM like Java (*e.g.,* JRebel (ZeroTurnAround 2012), Rubah (Pina & Hicks 2013)).

Original Source Code Rewriting. If there is no support for dynamic update of the running application, our approach is still applicable: A possible implementation is one that updates the source code of the application and recompiles it to be executed during the next execution of the application. This possible implementation might be integrated as an IDE plugin. By doing so, users take advantages of the approach as their application is automatically migrated when the tests are run in the IDE.

9. Limitations and Discussion

9.1. Limitations in Presence of Extensibility

The approach proposed in this article works well for method-level API changes. It has two main limitations: it does not support hook (in the sense of Hook-Template Design pattern) changes and intertwined changes that should be done in isolation. In addition, it does not support class nor instance variables deprecation. We illustrate the situations.

Deprecation of hook methods. Often a framework requests that a programmer overrides a given abstract method or method with a default (Alpert et al. 1998). Such a method is often not directly used by the framework *user* but by the framework internal logic following the Hollywood principle. For example the hook() method in Figure 5 is only called by the method template() and should be overridden in subclasses.

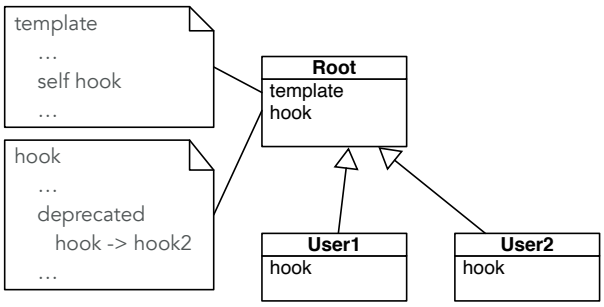

Figure 5 A simple hook and template situation where the default hook method is deprecated.

There are two problems with deprecating hooks:

- First, the fact that the redefinition of the hook in a subclass may simply not invoke the original hook definition in the superclass containing the deprecation declaration. In Figure 5, the method User1.hook() may not perform a super call to execute Root.hook(). Therefore the runtime rewriting is not executed and performed.
- The migration requires that redefined hook methods defined by the user should be renamed to follow the new name of the hook method. In Figure 5, deprecating hook() into hook2() should lead to the renaming of Root.hook() into Root.hook2() *and* User1.hook() into User1.hook2() and User2.hook() into User2.hook2().

DEPREWRITER does not handle this situation. In the context of library update, Schäfer *et al.* (Schäfer et al. 2008) proposed an approach to mine changes in the clients of a framework, including the changes in subclasses, abstract hooks, etc. Their work can serve as inspiration for detecting outdated overrides in client system and rewriting them with transformation rules.

Deprecation of Polymorphic Methods. Since DEPREWRITER renames call-sites based on callee-side deprecations, the developer must take special care of polymorphic call-sites to avoid incorrect rewrites. A polymorphic call-site is a call-site where many potential methods may be invoked, depending on the type of the receiver object. Indeed, there could exist the situation where a call-site targets during execution many methods, and that not all those methods are consistently deprecated. In that case, the first execution of a rewriting deprecation will rewrite the call site, breaking the compatibility with other, seemingly polymorphic, yet non-deprecated code. In our current solution,

it is up to the developer to deprecate all potential called methods of polymorphic call-sites.

Deprecating Intertwined Changes as Separated Single Changes. Some changes may be intertwined in the sense that a resulting transformations may be confused with another method to be migrated. For example, in Commander,[10] a command design pattern framework, the following migrations were necessary: basicName should be renamed as name, and name as dynamicName. In particular, basicName should not be migrated to dynamicName. Right now, our approach applies transformation rules one by one without a notion of a larger context. And, they cannot handle this case since, for a given basicName message, it will be rewritten down to dynamicName.

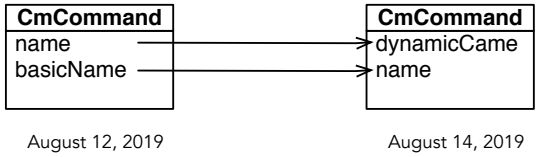

Figure 6 Method name was renamed to dynamicName and basicName was renamed to name. This means that the old method name cannot be deprecated, because the selector (name) of this method is taken by a new method. It also means that the migration refactorings must be applied in the correct order: (1) replace all calls to name with dynamicName and then (2) replace all calls to basicName with name.

Runtime Coverage. DEPREWRITER will only rewrite methods that are executed. If a deprecated method call is not covered by runtime nor by unit tests, it will not be rewritten. In that case, the only resort is static analysis and in such a case, developers get exposed to the limitations described in Section 2. Static analysis may introduce runtime bugs, while not executing methods may leave them outdated.

Class Deprecation. The deprecation of classes is a feature that has been introduced in Pharo 8 but got rather unnoticed. Its implementation and effective use should be documented and evaluated. Prior to it, developers followed an ad-hoc pattern. They defined the new class and let the deprecated class as a subclass of this new class. Deprecating the usage of a class is definitively more than that behavior, and it should be further studied. DEPREWRITER does not support class deprecation.

9.2. About the Rewriting Logic

Syntax. The definition of transformation rules is a bit verbose. It forces the developer to provide an antecedent that is exactly the one of the deprecated method, while it could use the method signature. This can lead to mistakes due to mismatch between the method signature and the rule antecedent. This can be solved using some compile-time code manipulations. It is an improvement that we will address in the future.

[10] https://github.com/pharo-spec/Commander2

About Continuing the Deprecated Execution. Our solution does implement call-stack replacement of deprecated method calls. When a deprecated method is reached, its call-site in the caller method is rewritten and the execution of the deprecated method continues. Indeed, the replacement(s) specified in the rewrite transformation are not called just after the deprecation transformation is applied. Because of this limitation, developers must re-run their tests to verify that the rewriting did not break them.

A future work is to investigate if it is possible to perform on-stack replacement and support execution of the new method from the deprecated call-site. However, this is challenging: the number and types of the arguments may differ between the old and new versions. New arguments may need to be computed, and old arguments removed. A deeper analysis is required to know which elements on the stack should stay or be replaced by other elements (potentially resulting from other messages sent).

About AOP and Automatic Deprecations. The way in which deprecations and deprecation transformations are expressed are orthogonal to our approach. In our current implementation, we have decided to add deprecation annotations next to the deprecated methods to make it simpler to manage. However, as mentioned in Section 8, our approach is also implementable using AOP frameworks: it is feasible to instrument the deprecated methods through weaving.

9.3. Discussing Survey Results

In this section, we discuss the factors that might have affected the result of the survey. We have identified a number of characteristics of the Pharo community and the way the survey has been done.

We consider that these elements do not affect the validity of the presented survey nor the validity of the proposed solution. However, these should be taken in account.

As the application of a tool and its perception by the community is affected by the culture and nature of the community, language, and platform; the presented result might not produce the same results when extrapolated to other languages and communities.

Integration with other Tools. The presented approach is integrated with the other tools existing in Pharo. Even though DEPREWRITER is not exposed with a tool, its effects are visible through all other tools in the IDE. For example, the versioning system tool is able to correctly identify the changes performed by DEPREWRITER and show them to the user to commit them. This natural integration with the tools may minimize the negative impact of the users seeing their code rewritten by its execution.

Integration in Different Versions. The tool has been integrated in Pharo 6, 7, 8, and 9 alpha. This might have minimized the impact of self-changing code to the users. The users are not surprised by the error messages or the modification appearing in the code, as they have internalized that this is normal behavior of Pharo. This might explain the high percentage of positive answers and the lack of answers describing it as surprising.

High Occurrence of Deprecations. The tool and its analysis have been applied on Pharo as shipped by the consortium. The Pharo community presents the characteristics of the continuous evolution of its APIs and implementations. The high occurrence of deprecations is an example of this, also the users of Pharo are used to find more and newer features in it. Pharo users not only are used to these migrations, but also they want to keep using the latest versions, even some using the version currently under development. So, this may explain the need for an automatic tool and the general good reception of it from the users.

Bias for the Topic. The survey was open, and developers replied to it based on their time and free will. Therefore, it probably attracted developers that were concerned by the problems or that already used the deprecation mechanisms. This would explain some high percentage of positive answers.

Lack of Tool and Knowledge. The usability of the rewrite engine may be a concern because there is no up to date documentation nor a dedicated tool to support the definition of rewriting deprecations. Developers proceed often by copy and paste existing deprecations and limit themselves to simple rewriting. It is our future plan to evaluate the impact of such tooling on the rule expressions.

Openness and Discoverability. Pharo users are used to discover tools, how they should be used, and their inner-workings by exploring the implementation, existing uses, and tests. This is a product of the open nature of Pharo, as all the code of the system and libraries is accessible from the IDE. This culture may have limited the fact that the tool did not have any proper documentation and that such absence of documentation hampered its use by unfamiliar developers.

10. Related work

In this section, we overview approaches and research that are related to our problem area. First, we discuss studies on refactoring and breaking changes. Then we overview empirical studies of deprecations and replacement messages. We proceed by discussing the work that was done around the code transformation and rewriting, as well as hot update and just-in-time (JIT) compilation, as it implies a kind of code transformation at execution time. Finally, we overview the work that is related to library migration and update, as well as the automatic generation of deprecations and replacement messages.

Refactorings. Since the seminal work of Opdyke (Opdyke 1992) on refactoring and the first refactoring engine developed by Brant and Robert (D. Roberts et al. 1996, 1997), refactorings have been the focus of multiple studies which focused on their usage and missed opportunities (Tom Tourwé & Mens 2003; Tsantalis & Chatzigeorgiou 2009; Bavota et al. 2011, 2010; Tsantalis & Chatzigeorgiou 2010). More recently, Liu *et al.* roposed a monitor-based instant software refactoring (Liu & Shao 2013). To the best of our knowledge, none of such approaches worked on supporting deprecations.

Breaking Changes. Breaking changes are the code modifications in all API elements that break backward compatibility (Dig & Johnson 2006). There were several empirical studies about the nature of those changes and their effect on client systems. Xavier *et al.* (Xavier, Brito, et al. 2017) performed a large-scale analysis of Java libraries to assess the frequency of breaking changes as well as their impact on clients. They 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.* (Xavier, Hora, & Valente 2017) 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.* (A. Brito et al. 2020) where they reported that 47% of breaking changes are due to refactorings. These results can be complemented by the previous study by Dig and Johnson (Dig & Johnson 2006) 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.

Deprecations and Replacement Messages. Robbes *et al.* (Robbes, Röthlisberger, & Tanter 2012) studied the impact of API changes, and in particular deprecations, on Pharo and Squeak ecosystems. They found out 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.* (Sawant et al. 2016) extended this study to Java. Despite collecting a larger dataset from Java ecosystem, authors report similar results to those of Pharo. Hora *et al.* (Hora et al. 2015, 2018) complemented the previous studies by analysing the impact of API evolution on the 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 the replacement messages in deprecations. Large-scale empirical studies of software written in Java and C# (G. Brito et al. 2016, 2018) as well as JavaScript (Nascimento et al. 2020) revealed that a large portion of deprecations in those languages (22-33%) are not supported with replacement messages. In their study of Pharo ecosystem, Robbes *et al.* (Robbes, Röthlisberger, & Tanter 2012) also showed that almost 50% of deprecation messages do not help to identify the correct replacement.

On the Fly AST Transformations. Just-in-time (JIT) compilers transform source code at runtime (usually bytecode to assembly) using information that is only available during execution. While JIT compilers are often using low-level information (modelled assembly and basic blocks), Truffle interpreters(Wöss et al. 2014) work at the AST level. When executed on the Graal VM (Würthinger et al. 2013), Truffle ASTs are transformed on the fly and recompiled to more efficient machine code. Reflectivity (Denker 2008; Costiou et al. 2020) is an advanced reflective layer that supports the transformation of AST at execution. Users of Reflectivity use it for building debuggers, AOP

systems, before/after methods, or program spies. Reflectivity offers AST transformation and decoration at the AST node level granularity. However, it does not offer a DSL to express transformations, and, from that perspective, it is more low-level for end-users.

Hot Update. The goal of the work presented in this paper is to help developers migrating their applications, and not to update executed code on the fly. This point is central because while the same approach could work by rewriting byte-code in the case on Java-like language, developers would still have to rewrite their code. Nevertheless, there is some related works on hot code updates. Ginseng is a dynamic software update implementation for C (Neamtiu et al. 2006). It implements change safety and updatability analysis (Hicks & Nettles 2005). It compiles programs specifically so that they can be dynamically patched and generates most of a dynamic patch automatically. Ginseng performs a series of analyses which, when combined with some simple runtime support, ensure that an update will not violate type-safety while guaranteeing that data is kept up-to-date. Rubah is a Dynamic Software Updater for Java, but it cannot update core libraries (Pina et al. 2014). Espell is an infrastructure that supports dynamic software update even for core libraries (Polito et al. 2015). Tesone *et al.* (Tesone et al. 2018) propose gDSU, a dynamic software update that works for both live programming and production environments. The solution implements safe update point detection using call stack manipulation and a reusable instance migration mechanism to minimize manual intervention in patch generation. Moreover, it also offers updates of core language libraries and the update mechanism itself. This is achieved by the incremental copy of the modified objects and an atomic commit operation.

Library Migration and Update. Our research is closely related to the problems of library update (updating a 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 (Kim et al. 2007; Xing 2007). Wu *et al.* (Wu et al. 2010) proposed an approach that combined call dependency and text similarity analyses. Schäfer *et al.* (Schäfer et al. 2008) 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 (Dagenais & Robillard 2011; S. Meng et al. 2012) and proposed changes that should be applied to client systems. Teyton *et al.* (Teyton et al. 2013) used data mining and propose rules for library migration by learning from the commit histories of already migrated clients. Hora *et al.* (Hora et al. 2014) proposed a similar approach to find method mappings between different releases of the same library. Pandita *et al.* (Pandita et al. 2015) approached the problem of library migration by analysing textual similarity of documentation from different libraries. Alrubaye *et al.* (Alrubaye et al. 2020) 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.

Meng *et al.* (N. Meng et al. 2013) help developers apply systematic edits (similar, but not identical, changes to many locations in source code) by learning from examples to find the correct location and apply the correct change. They present a tool called LASE that creates a context-aware edit script from two or more examples provided by developers and uses the script to automatically identify other edit locations and to transform the code.

Recommending Replacements for Deprecated Elements. There were several studies that automatically recommend replacements for the deprecated functionality. Brito *et al.* (G. Brito et al. 2018) recommend replacement messages for deprecations by learning from client systems that have already identified the correct replacements and updated their code. Xi *et al.* (Xi et al. 2019) designed an automatic process for migrating deprecating API to its replacement in client systems. Their approach is based on the replacement messages in code documentation. One possible extension of the study presented in this paper is automatic recommendation of transformation rules for deprecations.

11. Conclusion

This article presented a solution developed in joint work with the Pharo consortium to support deprecation for API changes within a dynamically-typed language and its ecosystem. The maintainer of a library annotates the changed method with a transformation rule. During client program execution, DEPREWRITER uses the annotation to find the call site of the deprecated method, and using the transformation rule, it rewrites and recompiles on the fly the method calling the deprecated method. Client programmers can then assess the produced changes and version their migrated code.

We presented an analysis of the 367 deprecations available in Pharo 8. Among them, we analyzed 218 rewriting deprecations to understand how they were used. Finally, we performed a user survey to understand the use pattern among developers programming in Pharo. We collected information from 46 software developers: Some knew DEPREWRITER and executed it on their code, some others were writers of rewriting deprecations, and finally, some developers were not aware of DEPREWRITER. 28 of 46 (60%) reported that the rewriting deprecations helped them, while 10 stated the inverse and 8 were uncertain.

DEPREWRITER only supports the deprecation of methods, and in the future, we would like to propose a solution for classes but also instance variables as well as offer solutions for the limitations of the approach we identified in Section 9.

Acknowledgments

The work is supported by the I-Site ERC-Generator Multi project 2018-2022. We gratefully acknowledge the financial support of the Métropole Européenne de Lille CPER DATA3. The authors thank Arolla for funding Oleksandr Zaitsev. We thank the Pharo consortium for its support.

References

Alpert, S. R., Brown, K., & Woolf, B. (1998). *The design patterns Smalltalk companion*. Boston, MA, USA: Addison Wesley.

Alrubaye, H., & Mkaouer, A., Mohamed Wiemand Ouni. (2019). On the use of information retrieval to automate the detection of third-party java library migration at the method level. In *ICPC'19*.

Alrubaye, H., Mkaouer, M. W., Khokhlov, I., Reznik, L., Ouni, A., & Mcgoff, J. (2020). Learning to recommend third-party library migration opportunities at the API level. *Journal of Applied Software Computing*, 106-140.

Bavota, G., De Lucia, A., & Oliveto, R. (2011). Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, *84*(3), 397–414.

Bavota, G., Lucia, A. D., Marcus, A., & Oliveto, R. (2010). Software re-modularization based on structural and semantic metrics. In *Working Conference on Reverse Engineering (WCRE'10)* (p. 195-204).

Black, A. P., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., & Denker, M. (2009). *Pharo by example*. Kehrsatz, Switzerland: Square Bracket Associates. Retrieved from http://books.pharo.org

Brito, A., Valente, M. T., Xavier, L., & Hora, A. (2020). You broke my code: understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, *25*(2), 1458–1492.

Brito, A., Xavier, L., Hora, A. C., & Valente, M. T. (2018). APIdiff: Detecting API breaking changes. *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 507-511.

Brito, G., Hora, A., Valente, M. T., & Robbes, R. (2016). Do developers deprecate APIs with replacement messages? a large-scale analysis on java systems. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (pp. 360–369).

Brito, G., Hora, A., Valente, M. T., & Robbes, R. (2018). On the use of replacement messages in API deprecation: An empirical study. *Journal of Systems and Software*, *137*, 306–321.

Chern, R., & De Volder, K. (2007). Debugging with control-flow breakpoints. In *International Conference on Aspect-Oriented Software Development (AOSD'07)* (pp. 96–106). New York, NY, USA: ACM. doi: 10.1145/1218563.1218575

Chow, K., & Notkin, D. (1996). Semi-automatic update of applications in response to library changes. In *International Conference on Software Maintenance (ICSM)* (Vol. 96, p. 359).

Colyer, A., & A., C. (2005). Aspect-oriented programming with AspectJ. *IBM Systems Journal*, *44*(2), 301-308. doi: 10.1147/sj.442.0301

Costiou, S., Aranega, V., & Denker, M. (2020, February). Sub-method, partial behavioral reflection with reflectivity: Looking back on 10 years of use. *The Art, Science, and Engineering of Programming*, *4*(3). doi: 10.22152/programming-journal.org/2020/4/5

Dagenais, B., & Robillard, M. P. (2008). Recommending adaptive changes for framework evolution. In *International Conference on Software Engineering (ICSE'08)* (pp. 481–490). New York, NY, USA: ACM. doi: 10.1145/1368088.1368154

Dagenais, B., & Robillard, M. P. (2011). Recommending adaptive changes for framework evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *20*(4), 1–35.

Demeyer, S., Ducasse, S., & Nierstrasz, O. (2002). *Object-oriented reengineering patterns*. Morgan Kaufmann.

Denker, M. (2008). *Sub-method structural and behavioral reflection* (PhD thesis). University of Bern.

Dig, D., & Johnson, R. (2006, April). How do APIs evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, *18*(2), 83–107.

Furr, M., hoon (David) An, J., Foster, J. S., & Hicks, M. (2009). Static type inference for Ruby. In *Symposium on Applied Computing (SAC'09)*.

Hayden, C. M., Smith, E. K., Denchev, M., Hicks, M., & Foster, J. S. (2012). Kitsune: Efficient, general-purpose dynamic software updating for C. In *International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'12)* (pp. 249–264).

Hicks, M., & Nettles, S. (2005, nov). Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, *27*(6), 1049–1096. doi: 10.1145/1108970.1108971

Hölzle, U., Chambers, C., & Ungar, D. (1991, July). Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America (Ed.), *Proceedings ecoop '91* (Vol. 512, pp. 21–38). Geneva, Switzerland: Springer-Verlag. doi: 10.1007/BFb0057013

Hora, A., Etien, A., Anquetil, N., Ducasse, S., & Valente, M. T. (2014). Apievolutionminer: Keeping api evolution under control. In *Proceedings of the software evolution week (CSMR-WCRE'14)*.

Hora, A., Robbes, R., Anquetil, N., Etien, A., Ducasse, S., & Valente, M. T. (2015). How do developers react to api evolution? the Pharo ecosystem case. In *International Conference on Software Maintenance (ICSM'15)* (pp. 251–260). doi: 10.1109/ICSM.2015.7332471

Hora, A., Robbes, R., Tulio Valente, M., Anquetil, N., Etien, A., & Ducasse, S. (2018, March). How do developers react to api evolution? a large-scale empirical study. *Software Quality Journal*, *26*, 161-191. doi: 10.1007/s11219-016-9344-4

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. G. (2001). An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP' 01)* (pp. 327–353). Springer Verlag.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997, June). Aspect-Oriented Programming. In M. Aksit & S. Matsuoka (Eds.), *European Conference on Object-Oriented Programming (ECOOP'97)* (pp. 220–242). Springer-Verlag. doi: 10.1007/BFb0053381

Kim, M., Notkin, D., & Grossman, D. (2007). Automatic inference of structural changes for matching across program versions. In *International Conference on Software Engineer-*

ing (ICSE'07) (pp. 333–343).

Lehman, M. (1996). Laws of software evolution revisited. In *European workshop on software process technology* (pp. 108–124). Berlin: Springer.

Liu, H., & Shao, X. G. W. (2013). Monitor-based instant software refactoring. *Transactions on Software Engineering*, *39*(8).

Meng, N., Kim, M., & McKinley, K. S. (2013). LASE: Locating and applying systematic edits by learning from examples. In *International Conference on Software Engineering (ICSE)* (pp. 502–511).

Meng, S., Wang, X., Zhang, L., & Mei, H. (2012). A history-based matching approach to identification of framework evolution. In *International Conference on Software Engineering (ICSE)* (pp. 353–363).

Mens, T., Ramil, J. F., & Godfrey, M. W. (2004, November). Analyzing the evolution of large-scale software: Issue overview. *Journal of Software Maintenance and Evolution: Research and Practice*, *16*(6), 363–365.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, *17*, 348–375.

Milojković, N., Béra, C., Ghafari, M., & Nierstrasz, O. (2016, August). Inferring Types by Mining Class Usage Frequency from Inline Caches. In *International Workshop on Smalltalk Technologies IWST'16.* Prague, Czech Republic. doi: 10 .1145/2991041.2991047

Murphy-Hill, E., Parnin, C., & Black, A. P. (2009). How we refactor, and how we know it. In *International Conference on Software Engineering (ICSE)* (pp. 287–297).

Murphy-Hill, E., Parnin, C., & Black, A. P. (2011). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, *38*(1), 5–18.

Nascimento, R., Brito, A., Hora, A., & Figueiredo, E. (2020). JavaScript API deprecation in the wild: A first assessment. In *International conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 567–571).

Neamtiu, I., Hicks, M. W., Stoyle, G., & Oriol, M. (2006). Practical dynamic software updating for C. In *Programming language design and implementation (PLDI)* (p. 72-83). doi: 10.1145/1133255.1133991

Nguyen, H. A., Nguyen, T. T., Wilson, G., Nguyen, A. T., Kim, M., & Nguyen, T. N. (2010). A graph-based approach to API usage adaptation. In *Conference on Object-Oriented Programming, Systems and Applications (OOPSLA'10)* (pp. 302 – 321).

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks* (Ph.D. Thesis). University of Illinois.

Pandita, R., Jetley, R. P., Sudarsan, S. D., & Williams, L. (2015). Discovering likely mappings between APIs using text mining. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)* (pp. 231–240).

Passerini, N., Tesone, P., & Ducasse, S. (2014, August). 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).*

Pina, L., & Hicks, M. (2013). Rubah: Efficient, general-purpose dynamic software updating for java. In *International Workshop on Hot Topics in Software Upgrades (HotSWUp).*

Pina, L., Veiga, L., & Hicks, M. (2014). Rubah: DSU for Java on a stock JVM. In *International conference on object-oriented programming, systems and applications (OOPSLA).*

Pluquet, F., Marot, A., & Wuyts, R. (2009). Fast type reconstruction for dynamically typed programming languages. In *Dynamic Languages Symposium (DLS)* (pp. 69–78). New York, NY, USA: ACM. doi: 10.1145/1640134.1640145

Polito, G., Ducasse, S., Fabresse, L., Bouraqadi, N., & Mattone, M. (2015). Virtualization support for dynamic core library update. In *Onward! 2015.*

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

Renggli, L. (2010). *Dynamic language embedding with homogeneous tool support* (PhD thesis, University of Bern). Retrieved from http://scg.unibe.ch/archive/phd/renggli-phd.pdf

Rivard, F. (1996, April). Smalltalk: a reflective language. In *Proceedings of REFLECTION'96* (pp. 21–38).

Rizun, M., Bach, J.-C., & Ducasse, S. (2015). Code transformation by direct transformation of asts. In *International Workshop on Smalltalk Technologies (IWST).*

Rizun, M., Santos, G., Ducasse, S., & Teruel, C. (2016, August). Phorms: Pattern Combinator Library for Pharo. In *International Workshop on Smalltalk Technologies IWST'16.* Prague, Czech Republic. doi: 10.1145/2991041.2991057

Robbes, R., Lungu, M., & Röthlisberger, D. (2012). How do developers react to API deprecation?: The case of a smalltalk ecosystem. In *International symposium on the Foundations of Software Engineering (FSE)* (pp. 56:1–56:11). New York, NY, USA: ACM. doi: 10.1145/2393596.2393662

Robbes, R., Röthlisberger, D., & Tanter, E. (2012). Extensions during software evolution: do objects meet their promise? In *European Conference on Object-Oriented Programming (ECOOP)* (pp. 28–52). Berlin, Heidelberg: Springer-Verlag. doi: 10.1007/978-3-642-31057-7_3

Roberts, D., Brant, J., & Johnson, R. E. (1997). A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, *3*(4), 253–263.

Roberts, D., Brant, J., Johnson, R. E., & Opdyke, B. (1996, April). An automated refactoring tool. In *Proceedings of ICAST '96.*

Roberts, D. B. (1999). *Practical analysis for refactoring* (Unpublished doctoral dissertation). University of Illinois.

Sandewall, E. (1978, March). Programming in an interactive environment: The "lisp" experience. *ACM Comput. Surv.*, *10*(1), 35–71. doi: 10.1145/356715.356719

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

Schäfer, T., Jonas, J., & Mezini, M. (2008). Mining framework usage changes from instantiation code. In *International Conference on Software Engineering (ICSE)* (pp. 471–480). New York, NY, USA: ACM. doi: 10.1145/1368088.1368153

Smith, B. C. (1984). Reflection and semantics in Lisp. In *Proceedings of POPL'84* (pp. 23–3). doi: 10.1145/800017.800513

Spoon, S. A., & Shivers, O. (2004). Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *Proceedings of ECOOP'04* (pp. 51–74).

Suzuki, N. (1981). Inferring types in smalltalk. In *Symposium on Principles of Programming Languages (POPL'81)* (pp. 187–199). New York, NY, USA: ACM Press. doi: 10.1145/567532.567553

Tesone, P., Polito, G., Fabresse, L., Bouraqadi, N., & Ducasse, S. (2018). Dynamic software update from development to production. *Journal of Object Technology*, *17*, 1–36. doi: 10.5381/jot.2018.17.1.a2

Teyton, C., Falleri, J.-R., & Blanc, X. (2013). Automatic discovery of function mappings between similar libraries. In *Working Conference on Reverse Engineering (WCRE)* (pp. 192–201).

Tom Tourwé, & Mens, T. (2003, March). Identifying refactoring opportunities using logic meta programming. In *European Conference on Software Maintenance and Re-engineering (CSMR'03)* (pp. 91–100). IEEE Computer Society Press.

Tsantalis, N., & Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, *35*(3), 347–367.

Tsantalis, N., & Chatzigeorgiou, A. (2010). Identification of refactoring opportunities introducing polymorphism. *Journal of Systems and Software*, *83*(3), 391–404.

Wöss, A., Wirth, C., Bonetta, D., Seaton, C., & Humer, C. (2014). An object storage model for the Truffle language implementation framework. In *PPPJ'14*.

Wu, W., Guéhéneuc, Y.-G., Antoniol, G., & Kim, M. (2010). Aura: a hybrid approach to identify framework evolution. In *International Conference on Software Engineering (ICSE)* (Vol. 1, pp. 325–334).

Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., … Wolczko, M. (2013). One vm to rule them all. In *International symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD'13)*.

Xavier, L., Brito, A., Hora, A., & Valente, M. T. (2017). 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).

Xavier, L., Hora, A., & Valente, M. T. (2017). Why do we break APIs? first answers from developers. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 392–396).

Xi, Y., Shen, L., Gui, Y., & Zhao, W. (2019). Migrating deprecated API to documented replacement: Patterns and tool. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware* (pp. 1–10).

Xing, E., ZhenchangandStroulia. (2007). API-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, *33*, 818 - 836.

ZeroTurnAround. (2012). *What developers want: The end of application redeploys.* http://files.zeroturnaround.com/pdf/JRebelWhitePaper2012-1.pdf.

About the authors

Dr. Stéphane Ducasse is Inria research director and head of the Pharo consortium. You can contact him at stephane.ducasse@inria.fr or visit http://stephane.ducasse.free.fr.

Dr. Guillermo Polito is CNRS research engineer and he is part of the Pharo board. You can contact him at guillermopolito@gmail.com or visit https://guillep.github.io.

Oleksandr Zaitsev is PhD student working for Arolla. You can contact him at oleksandr.zaitsev@inria.fr or visit http://oleks.fr.

Dr. Marcus Denker is Chargé de Recherche at Inria and he is part of the Pharo board. You can contact him at marcus.denker@inria.fr.

Dr. Pablo Tesone is Pharo consortium engineer and he is part of the Pharo board. You can contact him at pablo.tesone@inria.fr.

A. Appendix: Transformation rules proposed to Pharo

Table 10 presents the rules that have been proposed to Pharo developers. They were accepted and included in latest versions of Pharo v9.0.0. Pharo is an open source project with more than 150 contributors. Each pull request must be approved by one or multiple reviewers who are members of the core team and are different from the person submitting the PR.

B. Appendix: User Survey

Here is a verbatim of the questionary we sent.

B.1. Characterization

This group of questions intends to characterize the kind of user/developer you are regarding automatic deprecation rewritings.

Q0. What kind of software do you maintain? Help: Remember, frameworks differ from libraries because they are heavy users of the "Hollywood principle", you may mark YES in different categories, as you may have different roles in different situations.

Possible answers:

- I am an application developer
- I am a library developer
- I am a framework developer

Q1. How often do you migrate your software to newer versions of its dependencies? Help: Think about dependencies as a new version of the platform (*e.g.,* a Pharo version) or a new version of library/framework Possible answers: Very often, Often, From time to time, Not often, Never

Q2. Do you see value in having tools to help with code migrations?

B.2. Pharo Deprecation Automatic Rewritings

This group contains questions about the existing support in Pharo on deprecation automatic rewritings.

Q3. Do you know Pharo's support for automatic deprecation rewritings? Help: Have you ever used, found usages, or have your code rewritten by this support?

Q4. Did automatic deprecation rewritings help you in a migration?

Q5. How did automatic deprecation rewritings help you in a migration?

Q6. Did you write your own automatic deprecation transformation rules to help migrate your users?

Q7. How easy was it to write a rewrite rule? Possible answers: Very easy, Easy, Medium, Difficult, Impossible

Q8. Can you tell us about the cases where you found it impossible to use automatic deprecations?

B.3. Automatic Rewritings Settings

This group includes questions related to the user configurability of deprecation automatic rewritings.

Q9. A setting allows you to control if a deprecated call is automatically rewritten or not. Do you think automatic deprecations should be rewritten by default?

Q10. Another setting controls whether a deprecation should raise a warning or not when found at runtime. Do you think automatic deprecations should raise a warning by default?

Q11. What is your most frequent setup?

B.4. The Future of Deprecations

This group includes questions related to how to enhance existing support.

Q12. How would you enhance the existing support?

- Scoping of settings (to some packages, classes)
- Class deprecations
- Other

Category	Class	Method	Deprecation message	Antecedent	Consequent
Rename	Archive	canWriteToFileNamed:	use ... canWriteToFile: instead	`@rec canWriteToFileNamed: `@arg	`@rec canWriteToFile: `@arg
	Class	subclass: layout: slots: classVariables: category:	use ... package: instead	`@rec subclass: `@arg1 layout: `@arg2 slots: `@arg3 classVariables: `@arg4 category: `@arg5	`@rec subclass: `@arg1 layout: `@arg2 slots: `@arg3 classVariables: `@arg4 package: `@arg5
	FFIExternalReference	registerAsExternalResource	Use #autoRelease instead.	`@rec registerAsExternalResource	`@rec autoRelease
Replace	GTSpotter	isEmpty	Use hasSteps instead	`@rec isEmpty	`@rec hasSteps not
	MetacelloAbstract–PackageSpec	repository	Use repositories or repositorySpecs	`@rec repository	`@rec repositorySpecs first
	MetacelloMethod–Section	attribute :	Use attributes: instead	`@rec attribute: `@arg	`@rec attributes: ( OrderedCollection with: `@arg)
	SpPresenter	newMultiColumnList	Use newTable instead	`@rec newMultiColumnList	`@rec newTable
Different receiver	SystemNavigation	allCallsOn: from:	use #allCallsOn: of class directly	`@rec allCallsOn: `@arg1 from: `@arg2	`@arg2 allCallsOn: `@arg1
	SystemNavigation	allReferencesToPool:	use #usingMethods on the Pool	`@rec allReferencesToPool: `@arg	`@arg usingMethods
	SystemNavigation	isUsedClass:	just use #isUsed	`@rec isUsedClass: `@arg	`@arg isUsed
Remove argument	MetacelloPlatform	primeStackCacheFor: doing: defaultDictionary:	use #primeStackCacheWith: doing:	`@rec primeStackCacheFor: `@arg1 doing: `@arg2 defaultDictionary: `@arg3	`@rec primeStackCacheWith: `@arg3 doing: `@arg2

Table 10 Some of the 33 transformation rules that we have proposed to add to the non-rewriting deprecations found in Pharo 8.