Université de Lille, Faculté des sciences et technologies

Département de formation doctorale en informatique **École doctorale MADIS Lille**
UFR IEEA

Outils basés sur l'exploration de données pour soutenir la mise à jour des bibliothèques

(Data Mining-based Tools to Support Library Update)

THÈSE

présentée et soutenue publiquement le 28/10/2022

pour l'obtention du

Doctorat de l'Université de Lille

(spécialité informatique)

par

Oleksandr ZAITSEV

Composition du jury

Président :	Olga KOUCHNARENKO	Professeur, Université de Franche-Comté
Rapporteurs :	Romain ROBBES	Associate Professor, Free University of Bozen-Bolzano
	Coen de ROOVER	Associate Professor, Vrije Universiteit Brussel
Directeur de thèse :	Stéphane DUCASSE	Directeur de Recherche, INRIA Lille
Co-Encadrant de thèse :	Nicolas ANQUETIL	Maître de Conferences, Université de Lille

Centre de Recherche en Informatique, Signal, et Automatique de Lille — UMR 9189 CRIStAL
INRIA Lille - Nord Europe

I want to thank my parents, who encouraged and supported me throughout my studies. And all my friends who turned this into a pleasant journey. I am also grateful to the Arolla software company for sponsoring my PhD and the RMoD team of Inria for providing me with the best working environment during those four years. In particular, to my supervisors, Stéphane Ducasse and Nicolas Anquetil, for their guidance, wisdom, and patience.

But above all, I want to dedicate this thesis to the Armed Forces of Ukraine. To soldiers and partisans who heroically fight for our freedom as I write those lines. Medics and volunteers for all the tireless work they have not stopped since the first minutes of the war. To all our foreign friends and allies for the incredible support they are providing my people in our darkest hour. To everyone who believes in our victory and everyone who makes it possible.

Glory to Ukraine!

Abstract

Modern software can be characterized by a high degree of reuse of external artifacts. Applications depend on multiple libraries and frameworks, which in turn can also depend on other libraries and frameworks. Like any other software, libraries evolve. New versions are released, often incompatible with the previous ones. This forces client applications that depend on those libraries to update their code in response to library evolution.

Updating the dependencies can be a difficult and time consuming task for client developers. It involves repetitive operations and requires knowledge about the changes that were made to the library. Knowledge that can be shared by library developers or extracted by automated tools from source code or commit history.

In recent years, multiple approaches have been proposed to mine the data or apply machine learning techniques and extract knowledge about library update in the forms of rules. However, most of those approaches only focus on client developers and do not consider the expertise of library developers. They consider only simple method-to-method replacements and are only designed for statically-typed programming languages.

In this thesis, we address this gap in literature with five main contributions: (1) a survey of library and client developers from two industrial companies and an open-source community; (2) first detailed documentation of the Deprewriter approach and tool in Pharo which introduces deprecations that dynamically update client code with transformation rules; (3) a study of how Deprewriter was adopted by the Pharo community through the analysis of source code in Pharo 8 and a developer survey; (4) DepMiner — a novel approach to infer the rules for Deprewriter based on the commit history of a project; (5) a generalization of DepMiner as a new holistic approach to support library developers in the task of library update.

The results of the research reported in this thesis will advance the field of automated library update by exploring the perspective of library developers and the context of the dynamically-typed languages that were often overlooked in the previous studies.

Keywords: software evolution, library update, library migration, data mining.

Résumé

Les logiciels modernes peuvent être caractérisés par un haut degré de réutilisation d'artefacts externes. Les applications dépendent de plusieurs bibliothèques et frameworks, qui peuvent à leur tour dépendre d'autres bibliothèques et frameworks. Comme tout autre logiciel, les bibliothèques évoluent. Elles publient de nouvelles versions, souvent incompatibles avec les précédentes. Cela oblige les applications clientes qui dépendent de ces bibliothèques à mettre à jour leur code en réponse à l'évolution des bibliothèques.

La mise à jour des dépendances peut être une tâche difficile et chronophage pour les développeurs clients. Elle implique des opérations répétitives et nécessite la connaissance des modifications apportées à la bibliothèque. Ces connaissances peuvent être partagées par les développeurs de la bibliothèque ou extraites par des outils automatisés à partir du code source ou de l'historique des livraisons.

Ces dernières années, de nombreuses approches ont été proposées pour exploiter les données ou appliquer des techniques d'apprentissage automatique et extraire des connaissances sur les mises à jour des bibliothèques sous forme de règles. Cependant, la plupart de ces approches se concentrent uniquement sur les développeurs des applications clients et ne prennent pas en compte l'expertise des développeurs de bibliothèques. Elles ne considèrent que les remplacements simples de méthode à méthode et ne sont conçues que pour les langages de programmation à typage statique.

Dans cette thèse, nous abordons cette lacune dans la littérature avec cinq contributions principales : (1) une enquête auprès des développeurs de bibliothèques et de clients de deux entreprises industrielles et d'une communauté open-source ; (2) la première documentation détaillée de l'approche et de l'outil Deprewriter dans Pharo qui introduit des dépréciations qui mettent automatiquement à jour le code client avec des règles de transformation ; (3) une étude de la façon dont Deprewriter a été adopté par la communauté Pharo à travers l'analyse du code source dans Pharo 8 et une enquête auprès des développeurs ; (4) DepMiner — une nouvelle approche pour déduire les règles de Deprewriter à partir de l'historique des livraisons d'un projet ; (5) une généralisation de DepMiner comme une nouvelle approche holistique pour aider les développeurs de bibliothèques dans la tâche de mise à jour des bibliothèques.

Les résultats rapportés dans cette thèse font progresser le domaine de la mise à jour automatique des bibliothèques en explorant la perspective des développeurs de bibliothèques et le contexte des langages à typage dynamique qui ont souvent été négligés dans les études précédentes.

Mots-clés : évolution des logiciels, mise à jour des bibliothèques, migration des bibliothèques, exploration des données.

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

Contents

1.1 Software Evolution and Breaking Changes

The important characteristic of modern software development is a high degree of reuse of software artefacts. Software applications depend on multiple external libraries and frameworks, they can also communicate with microservice components [Baldassarre 2005]. For simplicity, in this thesis, we will generally refer to all reusable software as *libraries* and the developers who manage them as *library developers*. We will also refer to the software that depends on a given library as its *client system* that is managed by the *client developers*.

To facilitate the communication between the library and its client systems, the library developers define an *Application Programming Interface* (API) — a set of public classes, methods and fields that are meant to be used by clients. By an unspoken agreement, library developers are expected to keep the API stable and client developers are expected to use only the public API. This is an implicit contract enforced by encapsulation. In practice, however, both parties often violate the agreement.

Every software has to evolve [Lehman 1996, Demeyer 2002, Mens 2004]. This includes reusable libraries. There are two types of changes that can happen during library evolution:

1. *Non-breaking changes* (backward compatible) — do not modify the API and therefore do not affect the clients, unless client developers break the contract and use the private functionality.

2. *Breaking changes* (backward incompatible) — change certain parts of the API, which may break the client code and force client developers to update their systems.

Breaking changes in a library can also propagate from one client to another in what is known as the *ripple effect* [Yau 1978]. Breaking changes in the library may result in breaking changes in their client systems, which in turn will affect all clients of those clients [Robbes 2012a].

1.2 Library Update

The process of changing the client system in response to breaking changes in one of its dependencies is called the *library update*. It happens when a library releases a new version which is no longer compatible with the previous one. In this case, client developers who used the old version have a choice. They can continue using the old version, thus missing out on the new features and risking that, at some point, the support for the outdated library will be dropped. Or they can update their system to use the latest version.

In practice, library update in the presence of breaking changes can be hard and time consuming for the client developers. If functionality was removed, they need to know how to replace it. If the API has changed (*e.g.,* a class was renamed or an argument was added to a method), they need to understand how to use the new API. To ease this process and encourage their clients to update, library developers try to support them through documentation, communication channels such as online forums, or by introducing *deprecations*. Instead of immediately removing the functionality, library developers mark it as deprecated (to be removed) and only remove it in some future release. This gives their clients more time to update.

Both library and client developers could benefit from automated tools that would guide them in the process of library update. The implementation of such tools based on data mining techniques that extract knowledge from source code and commit history is the main focus of this thesis.

1.3 Problems

In this thesis, we study how can we support library and client developers in the process of library update by building tools. Specifically, we address three problems:

1. *Empirical problem.* Understand how library update happens in practice and how it is perceived by both library and client developers. What problems do they face? What support do they need? We address this problem in Chapter 4.

2. *Modeling problem.* What is the language that would allow one to express the information on how to update the dependency and could be used to transform the source code? (Chapter 5).

3. *Automation problem.* How can one automate the process of recommending transformations based on the data that is available: source code, commit history, etc.? (Chapters 6 and 7).

1.4 Contributions

The main contributions of this thesis are:

- A survey of library and client developers from two industrial companies and an open-source community (Chapter 4).

- A first detailed documentation of the Deprewriter approach and tool in Pharo which introduces deprecations that dynamically update client code with transformation rules (Chapter 5).

- A study of how Deprewriter was adopted by the Pharo community through the analysis of source code in Pharo 8 and a developer survey (Chapter 5).

- DepMiner — a novel approach to infer the rules for Deprewriter based on the commit history of a project (Chapter 6).

- A generalization of DepMiner as a new holistic approach to support library developers in the task of library update (Chapter 7).

1.5 Why Pharo?

Most contributions of this thesis are based on Pharo[1] [Black 2009] — an open-source dynamically-typed reflective object-oriented programming language inspired by Smalltalk. It is also an IDE written entirely in itself. That being said, our findings are not limited to Pharo exclusively and can be applicable to other programming languages such as Java, JavaScript, or Python.

We focus on Pharo because:

[1] https://pharo.org

1. Through our team, we have access to the Pharo community, including the core developers of Pharo and many of its most popular libraries.

2. Pharo provides reflective tools for analysing the language from within itself. This makes it easy to query the code, access method calls, extract deprecations, rewrite the code, etc.

For those readers who are not familiar with Pharo, we provide a short introduction to its syntax in Appendix A.

1.6 Structure of the Thesis

The thesis is organised as follows:

- In Chapter 2, we discuss the problem of library update, define the scope for this thesis and the terminology use in it. We also present several motivating examples to show why library update can be difficult and how developers can be supported by the tools.

- In Chapter 3, we discuss the state of the art.

- In Chapter 4, we present the results of our survey of library and client developers from two industrial companies and Pharo open-source comunity.

- In Chapter 5, we present the Deprewriter approach. We also explain how this approach is implemented in Pharo and propose several alternative ways to implement it in other programming languages. We analyse the source code of Pharo 8 and perform a developer survey to understand how Deprewriter was adopted by the community.

- In Chapter 6, we present DepMiner — the data mining approach that automatically infers transformation rules from the commit history and helps library developers deal with breaking changes.

- In Chapter 7, we present a holistic approach to support library developers in the task of library update. This is an improvement and a generalization of DepMiner presented in the previous chapter.

- In Chapter 8, we discuss the challenging scenarios of library update that go beyond simple method-to-method replacements.

- In Chapter 9, we conclude this thesis and discuss the future work.

1.7 List of Publications

Below is the list of papers that were published in the context of this thesis. We group them into three categories: *journal*, *conference*, and *workshop papers*. We also add a *technical reports* category which contains the work that was not peer-reviewed. The papers inside each category are presented in reverse chronological order.

Journal Papers

- Nicolas Anquetil, Julien Delplanque, Stéphane Ducasse, **Oleksandr Zaitsev**, Christopher Fuhrman, and Yann-Gaël Guéhéneuc. *What Do Developers Consider Magic literals? A Smalltalk Perspective*. Information and Software Technology, IST, 2022 [Anquetil 2022].

- Stéphane Ducasse, Guilermo Polito, **Oleksandr Zaitsev**, Markus Denker, and Pablo Tesone. *Deprewriter: On the fly rewriting method deprecations*. Journal of Object Technology, JOT, 2022 [Ducasse 2022].

Conference Papers

- **Oleksandr Zaitsev**, Stéphane Ducasse, Nicolas Anquetil, and Arnaud Thiefaine. *How Libraries Evolve: A Survey of Two Industrial Companies and an Open-Source Community*. 29th Asia-Pacific Software Engineering Conference, APSEC (industrial track), 2022 [Zaitsev 2022b].

- **Oleksandr Zaitsev**, Stéphane Ducasse, Nicolas Anquetil, and Arnaud Thiefaine. *DepMiner: Automatic Recommendation of Transformation Rules for Method Deprecation*. International Conference on Software and Systems Reuse, ICSR, 2022 [Zaitsev 2022a].

- **Oleksandr Zaitsev**, Stéphane Ducasse, Alexandre Bergel, and Mathieu Eveillard. *Suggesting Descriptive Method Names: An Exploratory Study of Two Machine Learning Approaches*. International Conference on the Quality of Information and Communications Technology, QUATIC, 2020 [Zaitsev 2020b].

Workshop Papers

- **Oleksandr Zaitsev**, Sebastian Jordan Montaño, and Stéphane Ducasse. *How Fast is AI in Pharo? Benchmarking Linear Regression*. International Workshop of Smalltalk Technologies, IWST, 2022 [Zaitsev 2022c].

- Julien Delplanque, Stéphane Ducasse, and **Oleksandr Zaitsev**. *Magic Literals in Pharo*. International Workshop of Smalltalk Technologies, IWST, 2019 [Delplanque 2019].

Technical Reports

- **Oleksandr Zaitsev**, Stéphane Ducasse, and Nicolas Anquetil. *Characterizing Pharo Code: A Technical Report*. https://hal.inria.fr/hal-02440055/, 2020 [Zaitsev 2020a].

Chapter 2
Background

Contents

In this chapter, we explain the process of library evolution, define the terminology that we use to refer to all of its objects and actors, and discuss the spectrum of problems that arise in this field. Inside this spectrum, we define the specific scope of problems addressed in this thesis. We finish this chapter with several motivating examples that demonstrate different scenarios of library update, explain why it can be hard, and how automatic tools can be used to support developers in those cases.

2.1 What is Library Update

Modern software depends on many reusable components such as libraries, frameworks, microservices, etc. [Baldassarre 2005]. For simplicity, in this work, we will refer to all reusable components as *libraries* and the software that depends on them as *client systems*. We refer to all developers who maintain a given library as *library developers* and all developers who depend on that library as *client developers* (or simply *clients*). It should be noted that those terms are relative: library developers are also clients of all the libraries that they use.

Like any other software, libraries evolve and release new versions [Demeyer 2002, Lehman 1996, Mens 2004]. Ideally, this process would only involve adding new functionality on top of the existing one and leaving the Application Programming Interface (API) unchanged. In practice however, library developers often introduce *breaking changes* — the changes to the API that break backward compatibility. In response to the breaking changes, the client systems that depended on the old version of a library must be changed if they are to use the new version. This process of changing client software in response to library evolution is called the *library update*. It should not to be confused with *library migration* — the process of

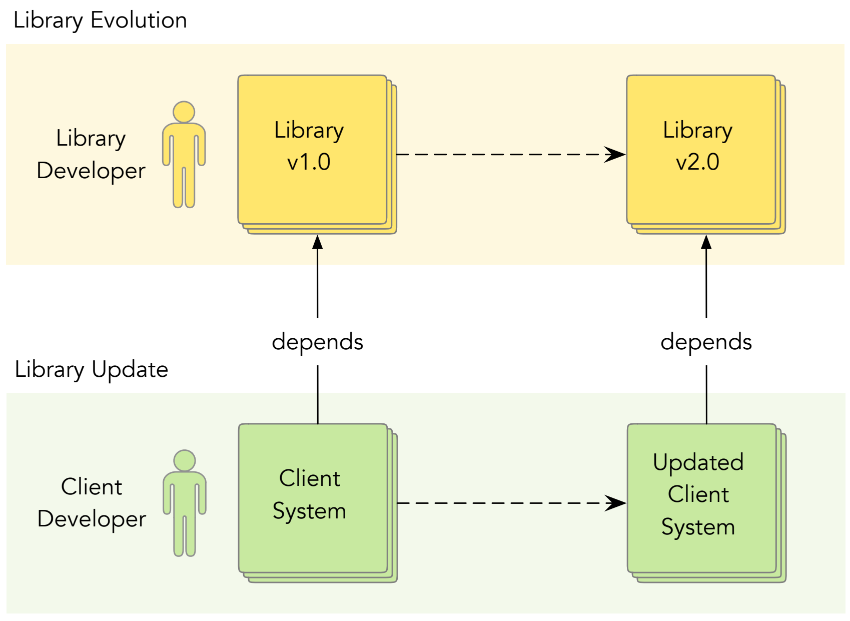


Figure 2.1: Library update is the process of updating the client system in response to library evolution. It has two groups of actors: library developers who release the new version of a library, and client developers who might need to update their code accordingly if they are to use the new version.

changing a dependency from one library to a *different* library. For example, updating the dependency from Struts v1.0.2 to v1.2.9 is library update, but changing the dependency from EasyMock to Mockito is library migration. Unfortunately, there is no universal agreement on terminology: in literature, those two terms are often used interchangeably.

The process of library update is illustrated in Figure 2.1. In this example, a library evolves from version v1.0 to v2.0. If this release contains breaking changes, then the client system must be updated if it is to use the new version. This task can be hard and time consuming for client developers.

To make library update easier for their clients and encourage them to update, library developers adopt different practices such as avoiding breaking changes altogether, deprecating functionality before removing it, or documenting the changes in the new release. We will explore the adoption of those practices in Chapter 4.

2.2 Deprecations

When certain parts of the API from version n must be modified in version $n + 1$, this may break compatibility and cause problems for the clients. For example, if an interface has to be removed, a class must be renamed, or a method must receive additional arguments. Instead of introducing those breaking changes immediately into the new release, library developers can release an intermediate version $n + 1$ in which they label certain elements of the API as *"deprecated"*, thus informing clients that those elements will be changed or removed in the subsequent release $n + k$ [Brito 2018a, Brito 2018b]. Client systems that use a deprecated feature receive a deprecation warning which gives developers time to update their code and adapt to the forthcoming changes.

Besides notifying that the method should not be used, library developers can also inform clients of the alternatives in the new API that can be used instead (we refer to them as *replacements*) using one or multiple practices:

– *Replacement messages.* Developers can supply deprecations with textual messages that suggest a replacement for an obsolete item (*e.g.,* a comment or a warning message *"Method x() is deprecated, use y() instead"*).

– *Annotations with references.* Some programming languages provide annotations that can be added to method definitions and reference the replacement in source code (*e.g.,* in Java, the @Deprecated annotation or @deprecated Javadoc tag, combined with @link or @see tags).

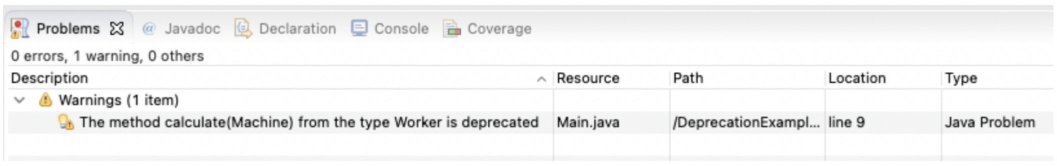
– *Transformation rules.* Several research artifacts allow library developers to write transformation rules that will be used to update client code. In Chapter 5, we will present an approach in Pharo that allows to add transformation rules to method deprecations and apply them when deprecation warning is signalled.

In the listing below[1], we show an example of a method Worker.calculate() being deprecated in Java. In this case, library developers used both the @Deprecated annotation (line 13) and the @deprecated JavaDoc tag (line 4). They also provided a replacement message on line 7, informing clients that Utils.calculatePeriod() should be used instead and referencing the specific method in the system using the @link tag. As can be seen in Figure 2.2, an IDE (in this case, Eclipse) will show a warning whenever the deprecated method is invoked, nonetheless, the client system will continue working.

```
1 public class Worker {
2   /**
3     * Calculate period between versions
4     * @deprecated
5     * This method is no longer acceptable
6     * to compute time between versions.
7     * <p> Use {@link Utils#calculatePeriod(Machine)}
```

[1]Example taken from https://www.baeldung.com/java-deprecated. Accessed: 16-05-2022.

```
     * instead.
     *
     * @param machine instance
     * @return computed time
     */
   @Deprecated
   public int calculate(Machine machine) {
     return machine.exportVersions().size() * 10;
   }
}
```


Problems ⊠ | Javadoc | Declaration | Console | Coverage

0 errors, 1 warning, 0 others

Description	Resource	Path	Location	Type
Warnings (1 item)				
The method calculate(Machine) from the type Worker is deprecated	Main.java	/DeprecationExampl...	line 9	Java Problem

Figure 2.2: Deprecation warning in Eclipse.

In the listing below, we present the same method deprecation written in Pharo. The first line defines the method signature. Lines 2-4 contain the comment which explains that the method is deprecated and should not be used. Line 6 defines the method deprecation with a warning message that contains a replacement message *“Use Utils >> calculatePeriod instead”*. Finally, line 7 contains the body of the method which will be executed even after the warning is signalled.

```
1  Worker >> calculate: machine
2      "Calculate period between versions
3      This method is no longer acceptable
4      to compute time between versions."
5
6      self deprecated: 'Use Utils >> calculatePeriod instead'.
7      ↑ machine exportVersions size * 10
```

As we will see in Chapter 3, in practice, library developers often decide to remove functionality without deprecation. Hora *et al.*, [Hora 2015] report that 59 out of 118 API changes that they analysed are a missed deprecation opportunity. According to Brito *et al.*, [Brito 2019], the most common reason for that is the fear of increasing the maintenance effort (*e.g.*, library developers believe that the breaking change will not affect many clients and deprecations will only add complexity and maintenance issues). Several large-scale studies of popular software projects have revealed that the proportion of deprecations that do not contain a helpful replacement message (in the form of a comment, string, annotation, etc.) is 33% for Java, 22% for C# [Brito 2018b], and 33% for JavaScript [Nascimento 2020]. This suggests that there is a need for automated tools to support library developers in writing deprecations with replacement messages.

2.3 Scope of the Problem

The general problem of library update can be approached from different perspectives. For example, we can support library or client developers, propose changes before the library is released or after that. In this section, we attempt to systematise different aspects of the library update problem and specify a clear scope for this thesis.

Library or client developers? First of all, library update can be approached from the perspective of the developers who are being supported. We can develop tools for client developers who need to update their code and use the new version of a library, or for library developers, who want to make it easy for their clients to update. *In this thesis, we focus both on library and client developers, but we pay more attention to the perspective of library developers because it was less explored in the literature* (see Chapter 3).

Before or after the release? Library update can also be approached from two temporal angles: before the release, when the library and the client system are being developed, or after the release, when all changes have been made and the client system needs to be updated. To approach this problem before the release, one must improve the architecture of a library, provide abstraction layers to reduce the volatility of the API. From the client side, the architecture can also be improved by reducing the coupling with external libraries and making sure that changes in dependencies have small effect on the system. *In this work, we do not focus on the architectural issues. We are only concerned with what happens when the library releases a new version with breaking changes and the client system has to react.*

Type of missing API elements. Breaking changes can affect different parts of library API. Those can be methods, classes, interfaces, fields, etc. *In this thesis, we focus on breaking changes that affect methods.*

There are three specific aspects of the library update problem that we address in this thesis:

1. *Empirical problem.* Understand how library update happens in practice and how it is perceived by both library and client developers. What problems do they face? What support do they need? (Chapter 4).

2. *Modeling problem.* What is the language that would allow one to express the information on how to update the dependency and could be used to transform the source code? (Chapter 5).

3. *Automation problem.* How can one automate the process of recommending transformations based on the data that is available: source code, commit history, etc.? (Chapters 6 and 7).

2.4 Motivating Examples

In this section, we provide several examples to demonstrate the problem of library update from the perspective of library and client developers. We demonstrate why it can be hard for developers to understand changes in the library. We also discuss how developers could be supported with transforming deprecations and the tools that extract knowledge from commit history in the form of rules. In this thesis, we focus primarily on dynamically-typed programming languages. This is why we demonstrate code written in Python in the first two examples, and in the third example, we use Pharo. The consequences of dynamic typing will become evident in Example 2.

2.4.1 Example 1: Perspective of a client developer

Consider a software library ailib that provides different algorithms for artificial intelligence (AI), machine learning and data mining. Developer Alice uses this ailib to train a linear regression model — machine learning algorithm for modelling the relationship between a set of input variables X and an output variable y. Alice uses this algorithm to estimate the salary of employees in a company based on their age and gender.

In the code listing below, we present an example of the script that Alice wrote to train her model using ailib v1.0. First, Alice loaded the data using readData function (line 4) that accepted two arguments: path to the file and type of file to be read, in this case, CSV. Then she instantiated an object of LinearRegression class (line 6) and trained the model using its train method (line 7) which accepted two arguments: the data and the name of the output column y. All other columns of the data table will be used as input variables X. Finally, Alice used the predict method of the model (line 9) to estimate the salary of a new employee.

```
1 from ailib.models import LinearRegression
2 from ailib.data import readData
3
4 data = readData('dataset.csv', type='CSV')
5
```

```
model = LinearRegression()
model.train(data, ycolumn='salary')

salary = model.predict([26, 'female'])
```

When the new version of ailib was released, Alice tried to update her dependency but this broke her code. It turns out that, in ailib v2.0, the following changes were introduced:

1. Rename LinearRegression to AILinearRegression.
2. Replace method readData(file, type) with a more specific method readCsv(file).
3. Replace method train(data, ycolumn) with method fit(x, y) that accepts two arguments: an input matrix x and an output vector y.
4. Change method predict in such a way that instead of accepting only one observation as input, it now accepts a collection of observations and predicts the outputs for all of them.

To use ailib v2.0, Alice is expected to rewrite her code in the following way:

```
from ailib.models import AILinearRegression
from ailib.data import readCsv

trainData = readCsv('dataset.csv')

x = trainData.columns(['age', 'gender'])
y = trainData.column('salary')

newX = [[26, 'female']]

model = AILinearRegression()
model.fit(x, y)

salaries = model.predict(newX)
salary = salaries[0]
```

However, Alice does not know any of that. From her point of view, the code that worked fine on ailib v1.0, suddenly broke on v2.0 with two error messages: 'Error: class LinearRegression was not found' and Error: function readData was not found'. Alice needs support from the library developers that would help her understand (1) what changes were made to ailib in v2.0 and (2) how must she change her code to react to those changes. Such a support can come in the form of documentation, release notes, or deprecations. If support is not available, Alice might have to read the source code of the library or find other clients that have already updated. As we will discuss in Chapter 3, many automated tools have been proposed to help Alice in this difficult task.

2.4.2 Example 2: Perspective of a library developer

Now we will look at the same example from the perpective of the library developer Bob who is preparing the release of ailib v2.0. Bob is aware that the upcoming release will contain multiple breaking changes. He knows that it is important to support the clients and make the process of updating from v1.0 to v2.0 as simple as possible (in Chapter 4, we will discuss the motivation of library developers to support their clients).

Bob himself has renamed the class LinearRegression to AILinearRegression. He documented this change and added a deprecation that would give his clients time to update without breaking their code. Replacing all references to LinearRegression in client code to AILinearRegression is a simple repetitive operation that could be automated. Bob could benefit from a tool that would allow him to "replay" this rename refactoring operation on the code of his clients or a deprecation mechanism that would automatically rewrite client code whenever the client attempts to instantiate a deprecated class (a similar mechanism for deprecated methods will be presented in Chapter 5).

While preparing the release, Bob also notices that the function readData as well as the method fit, that were often referenced in documentation and tutorials, no longer exist in ailib v2.0. Bob was not the one who introduced those changes. It was done by one of the other 100 developers who contributed to ailib. Now Bob needs to search the source code to understand (1) why were those methods removed, and (2) what are the replacements that he could propose to the clients. Those questions are similar to the ones asked by Alice in the previous example. The difference is that Bob has better knowledge of the source code of ailib and access to other developers of this library. He also has the responsibility to document *all* missing methods, and not only those that are used by Alice. In this case, Bob could also benefit from an automated tool that would identify the missing methods in the API and recommend replacements. When possible, those recommendations could be expressed in the form of rules that would be used by Bob to automatically update the client code. We will discuss such tools in Chapters 6 and 7.

The first two breaking changes require simple updates of client code that could be automated. For example, replacing all references to LinearRegression class with AILinearRegression, or replacing all method calls readData(file, type='CSV') with readCsv(file). However, the other two breaking changes require more complex modifications to the client code. To replace the method calls to train(data, ycolumn) with fit(x, y), one must first construct matrix x and vector y from the data object. Method predict was also changed to accept a collection of observations instead of a single observation. In statically-typed languages such as Java, this would result in the change of signature of the method — because the types of arguments have changed. However, Python is a dynamically-typed language, which means that the signature of method predict was not affected by the breaking change. Nonetheless, the clients have to change the logic of using this method: instead of applying it to one observation at a time, they now have to predict the output for the entire dataset. Those kinds of challenging library update scenarios will be discussed in Chapter 8.

2.4.3 Example 3: The case from a real open-source project

The version v8.0 of the Pharo project[2] contained a method insertCompletion of class NECMenuMorph. After a year and a half of adding new features and fixing bugs, the community decided to release a new version — Pharo v9.0. However, in this version the method insertCompletion was no longer present. It was removed without deprecation and this change was not documented.

Core developer *X* who was preparing the release, noticed that insertCompletion was missing and decided to document this breaking change and suggest a replacement to the clients who might be affected by it. However, during one and a half year of development, the Pharo project had more than 100 contributors who have removed 11,862 methods and added 13,277 methods. This made it difficult for *X* to understand why was this particular method removed or what was a good replacement for it.

Using the prototype tool implementing the approach that we present in Chapter 7, developer *X* could find all public methods that were present in Pharo v8.0 but no longer exist in v9.0. *X* was informed that a particular method NECMenuMorph.insertCompletion() was removed by developer *Y* on February 15, 2020 in commit a52462a. Our tool suggested that the removal was caused by a combination of three refactoring operations. First, *Y* moved this method to CompletionEngine class. Then, he renamed the method to replaceCompletionWith(). Finally, on the next day, *Y* renamed this method again to replaceTokenInEditorWith(). Therefore, our tool could suggest *X* that the missing method NECMenuMorph.insertCompletion() was replaced in the new version by CompletionEngine.replaceTokenInEditorWith(). Using this information, developer *X* could document the breaking change and suggest replacement to the clients.

[2]Pharo is a programming language but it is also an IDE written entirely in itself. This can be a source of confusion. In this thesis, we will often refer to Pharo as an open-source project that is implemented in the Pharo programming language, and hosted at https://github.com/pharo-project/pharo.

CHAPTER 3
State of the Art

Contents

In this Chapter, we discuss the state of the art in the field of library update. Based on the three aspects of the library update problem that we defined in Section 2.3, we split this overview into three parts:

1. *Empirical studies of library evolution* — case studies of library evolution, breaking changes and their effect on client systems either by means of code analysis or the surveys of library or client developers.

2. *Code transformation and deprecations* — what is the language that can be used to express code transformations with rules for library update?

3. *Tools to support library update* — in this section, we discuss the automated tools that were proposed to support library and client developers in the task of library update or library migration.

3.1 Empirical Studies of Library Evolution

We start by overviewing the existing empirical studies on library evolution and discussing their shortcomings. In Table 3.1, we systematise the related studies using two criteria:

- *Developer survey or code analysis?* In literature, there are two main techniques for studying library evolution and its effect on clients:

    1. Analysing the source code of the evolving libraries and/or client systems; i.e., *"How do libraries change, and how do those changes propagate?"*

Table 3.1: Empirical studies of library evolution characterized by two criteria: is it based on a developer survey, on source code analysys, or both?; does it explore the client perspective, the library perspective, or both?

Paper	**Dev. survey**	**Code analysis**	**Client persp.**	**Library persp.**
[Robbes 2012a]	no	yes	yes	no
[Jezek 2015]	no	yes	yes	yes
[Hora 2015]	no	yes	yes	no
[Bogart 2016]	yes	no	yes	yes
[Sawant 2016]	no	yes	yes	no
[Xavier 2017a]	no	yes	yes	yes
[Xavier 2017b]	yes	no	no	yes
[Hora 2018]	no	yes	yes	no
[Kula 2018a]	yes	yes	yes	no
[Kula 2018b]	no	yes	no	yes
[Brito 2019]	yes	no	yes	yes
Our study (Chapter 4)	**yes**	**no**	**yes**	**yes**

2. Surveying the developers to explore the human side of library evolution; i.e. *"How do developers perceive this process?"*

- *Client or library perspective?* Library evolution can be explored from the perspective of libraries or from the perspective of their clients. Some studies (like this one) explore both sides.

Studies that analyse source code. There have been multiple studies of the ripple effect caused by breaking changes and how it propagates through client systems. Robbes *et al.,* [Robbes 2012a] studied the reaction of clients in Pharo ecosystem to the deprecation of API elements in their dependencies. They conclude that many clients do not react to library deprecations and when they do react, they often do not apply adaptations to the entire project at once, leaving it inconsistent. Sawant *et al.,* [Sawant 2016] performed a partial replication of this study on Java projects, considering almost 10 times more client systems. They arrive to the same conclusions as the ones derived from Pharo ecosystem. Hora *et al.,* [Hora 2015, Hora 2018] extended the study of Robbes *et al.,* [Robbes 2012a] by exploring the changes in non-deprecated API elements. They report that half of the analysed API changes are missed deprecation opportunities and suggest that recommender tools can be built to help library developers introduce those deprecations. They also observe that most API changes can be implemented as rules and suggest that those rules can be used to help client developers automatically update their code. Jezek *et al.,* [Jezek 2015] studied the evolution of Java libraries both from library and client perspectives. They report that breaking changes are very frequent (80% of version updates break compatibility), however, this causes few actual problems in real client systems. Xavier *et al.,* [Xavier 2017a] performed a large-scale analysis of Java libraries and their clients to understand how frequent are breaking changes, how do they evolve over time, and how do they impact the clients. They discovered that on the median, 15% of all changes in a library are breaking changes and their frequency increases over time. However, they also report that most breaking changes do not have big impact on client systems and explain this with a hypothesis that library developers try not to break highly impactful API elements. Kula *et al.,* [Kula 2018a, Kula 2018b] performed two studies: one on the client side and another one on the library side. In their first study they report that most client systems rarely update their libraries and 81.5% of client systems choose to remain with the older popular version of the library. In their second study, Kula *et al.,* [Kula 2018b] explored the evolution of libraries and found that many breaking changes happen in non client-used API and non API classes, thus the client-used API are less likely to be broken.

Developer surveys. Bogart *et al.,* [Bogart 2016] performed a case study by interviewing developers of three software ecosystems to understand how developers make decisions about changes and document the practices that are used in those communities. In the same study where Kula *et al.,* [Kula 2018a] discovered that most client systems have outdated dependencies, they performed a survey of client developers and found that 69% of them are unaware of their vulnerable dependency. They also report that developers are reluctant to update because they perceive it as extra workload. Xavier *et al.,* [Xavier 2017b] performed a survey of seven core developers from popular Java libraries. Based on this survey, they propose a list of five reasons that motivate breaking changes: library simplification, refactoring, bug fix, dependency changes, and project policy. They report that all surveyed developers are aware of the effect of breaking changes on clients. In their follow-up study, Brito *et al.,* [Brito 2019] conducted a larger survey of 56 library developers through a firehouse interview. According to their study, the most common reason for introducing breaking changes are the need to implement new features (32%), API simplification (29%), and improving maintainability (24%). In the same study, Brito *et al.,* [Brito 2019] also explored the actual impact of breaking changes on clients by analysing the questions published on StackOverflow. They report that 45% of all posts related to breaking changes are client developers asking how to overcome the negative effect of those changes on their code.

Shortcomings of the existing studies. Most surveys that we discussed either targeted specific developers and asked them about breaking changes or vulnerable dependencies that were detected by authors [Brito 2019, Kula 2018a, Xavier 2017b], or analysed the questions related to breaking changes that were published on StackOverflow [Brito 2019]. To the best of our knowledge, Bogart *et al.,* [Bogart 2016] are the only authors that collected diverse population of developers and asked them general questions about the process of library evolution and the practices that they use. However, that study only focused on library developers. Also, no survey has asked client developers about what makes the process of library update hard and what support do they expect from library developers to make this process easier.

3.2 Code Transformation and Deprecations

Robbes *et al.,* [Robbes 2012b] 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 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 2015, Hora 2018] complemented the previous studies by analysing the impact of API evolution on Pharo ecosystem, but focusing only on those changes which are not related to deprecations. They claim that API changes have large impact on the ecosystem and most of the changes that they found can be implemented as rules in static analysis tools. Several authors have also explored the effectiveness of deprecation messages. Large-scale empirical studies of software written in Java and C# [Brito 2016, Brito 2018b] as well as JavaScript [Nascimento 2020] revealed that a large portion of deprecations in those languages (22-33%) is not supported with replacement messages. In their study of Pharo ecosystem, Robbes *et al.,* [Robbes 2012b] also showed that, at that time, almost 50% of deprecation messages did not help identify the correct replacement.

3.3 Tools to Support Library Update

In this section, we provide an overview of the previous approaches that have been proposed to support developers in the problem of library update. We also discuss approaches that deal with library migration because those two problems are closely related, and the same techniques can often be applied to both problems. In Table 3.2, we summarise the related approaches and compare them based on three features:

- the **problem** that they solve: *library update* or *library migration*;

- the **type of input** from which they extract the information: exp(L) — expert knowledge of library developers, 2v(L) — two versions of library code, hist(C) — commit history of a client system that was already migrated, etc.;
- the **technique** that they use to match API entities: *textual similarity* (TS) of method signatures, comments, or documentation, *structural similarity* (SS) of source code, or *call dependency* (CD) — analysis of changes in the call sites of the API entities.

Table 3.2: Related approaches classified by **type of input**: *hist* = commit history, *2v* = 2 versions of source code, *doc* = documentation, *L* = library, *C* = migrated clients, *T* = unit tests of the library, and **technique**: *TS* = textual similarity, *SS* = structural similarity, *CD* = call dependency.

Paper	**Problem**	**Persp.**	**Type of Input**	**Technique**
[Chow 1996]	update	library	exp(L)	—
[Henkel 2005]	update	library	exp(L)	—
[Kim 2007]	update	—	2v(L)	TS
[Xing 2007]	update	client	2v(L)	SS, TS
[Dagenais 2008]	update	client	hist(L)	CD
[Schäfer 2008]	update	client	2v(C,T)	CD
[Wu 2010]	update	client	2v(L)	CD, TS
[Nguyen 2010]	update	client	2v(L,C,T)	CD, TS, SS
[Meng 2012]	update	client	hist(L)	CD
[Teyton 2013]	migration	client	hist(C)	CD
[Hora 2014]	update	client	hist(L)	CD
[Pandita 2015]	migration	client	2v(L)	TS
[Alrubaye 2019]	migration	client	hist(C), doc(L)	CD, TS
[Alrubaye 2020]	migration	client	doc(L)	CD, TS
Our study (Chapter 5)	*update*	*library*	*exp(L)*	—
Our study (Chapter 6)	*update*	*library*	*hist(L), exp(L)*	*CD*
Our study (Chapter 7)	*update*	*library*	*hist(L), exp(L)*	*CD*

Sources of information. The approaches that we discuss in this section are based on knowledge extraction techniques that mine the information about library update or migration from documentation, commit history, source code, the expertise of library developers, etc. To systematise those approaches we identify eight sources of information that can be used as input for proposed mining techniques. The short symbolic notation will be used in Table 3.1 to compare the different state of the art approaches based on their type of input.

exp(L) — expert knowledge provided by library developers. It can take the form of a library update script (a set of manually written library update rules), recorded and documented changes that were made to library's API and can be "replayed" on the client code, etc. [Chow 1996, Henkel 2005]

doc(L) — documentation of a library. Can be used to find a mapping between classes, methods, or fields by comparing their descriptions between two versions of the documentation [Alrubaye 2019, Alrubaye 2020].

hist(L) — commit history of a library. Commits are a source of very granular and detailed information that completely describes library evolution. It can be used either to mine changes that affected the API and apply them to client code or to detect the locations in source code where library uses its own API and observe how the invocations in these locations changed as API evolved [Dagenais 2008, Meng 2012, Hora 2014].

2v(L) — two versions of library code. By comparing the source and target versions of library code, one can map similar entities (classes, methods, etc.) and either generate replacement rules or use this mapping to support other approaches [Kim 2007, Xing 2007, Wu 2010, Nguyen 2010, Pandita 2015].

hist(C) — commit history of a client system that has already been updated. It describes the changes that were performed on client systems as they were updated to the new version of a library [Teyton 2013, Alrubaye 2019].

2v(C) — two versions of code of a client system that has already been updated. Although it is less granular than the commit history, by comparing two versions of client code before and after the library update, one can build the diff and infer the changes that were made [Schäfer 2008, Nguyen 2010].

hist(T) — commit history of the library's unit tests. Inside its unit tests, the library invokes its own API attempting to mimic its most common use cases. As API evolves, tests must be updated and provided that the library has good test coverage, by learning from test evolution, one can generate the library update rules that will cover a wide range of API use cases. Often the commit history of a library includes the commit history of unit tests: $hist(T) \subset hist(L)$, but this is not always the case.

2v(T) — two versions of the library's unit tests. As an alternative to analysing the evolution of tests throughout the entire commit history, one can compare only the tests in the source and target versions of the library. Once again, it is often the case that $2v(T) \subset 2v(L)$ [Schäfer 2008, Nguyen 2010].

Expert-guided library update. The first studies of how to support client developers in updating their systems to the new versions of evolving libraries were based on collecting the expertise of library developers in the form of transformation rules and applying them to client code in a semi-automatic manner. Chow and Notkin [Chow 1996] proposed library developers to annotate changed functions with transformation rules that can be applied to client code. They designed a language for expressing code transformations and implemented a semi-automatic tool that applies transformations to AST and generates the modified source code. Henkel et al. [Henkel 2005] decided to reduce the added cost for library developers. They proposed to support the API evolution by recording the refactorings as they are performed on the library and "replaying" the recorded changes on the client codebase.

Deprecation messages are among the most common ways for library developers to recommend replacements for the removed functionality. Modern programming languages provide powerful support for annotating deprecated elements of API with references to the possible replacements. In their large-scale empirical study of deprecation messages in Java and C# projects, Brito et al. [Brito 2018b] reported that 66.7% of Java deprecations and 77.8% C# deprecations contain replacement messages.

Automatic library update and migration. In recent years, many approaches have been proposed that do not require the direct involvement of library developers. Those approaches extract all necessary information from the source code, commit history, or code documentation. Kim et al. [Kim 2007] proposed to find matches between the two versions of API by calculating textual similarities of method signatures collected from the source code of the two versions of the library. They defined a set of low-level API transformations (e.g., package replacement, argument deletion, etc.) and performed a rule-based matching to find a mapping between the two versions of library API. Xing et al. [Xing 2007] proposed a Diff-CatchUp tool that compares two versions of the library's source code, detects changes to the API, and proposes transformation rules together with working usage examples. Unlike Kim et al., they calculated the structural similarity of source code and not only the textual similarity of method signatures. Unlike previous automatic approaches to library update, which compared two versions of library's source code, the novel SemDiff tool proposed by Dagenais and Robillard [Dagenais 2011] extracted the necessary information on a more granular level from the commit history of a library. It recommended changes to client systems based on how the library reacted to its own evolution. SemDiff could recognise changes that were more complex than simple refactorings, for example, method additions and deletions, and recommended multiple replacements for methods that no longer existed in API, supplying each one with a confidence score.

Schäfer et al. [Schäfer 2008] were the first authors who proposed mining library update rules from already updated client systems. They used the library's unit tests as an additional source of information. Unit tests describe the use cases of the library's API and therefore, can be treated as one of the clients that must react to API changes. Schäfer et al. generated rules for the library changes that were caused not only by refactorings but also by conceptual changes (changed or replaced concepts, altering the responsibilities of the building blocks, removing certain behaviour, etc.). They used the A-Priori algorithm to mine the transformation rules from two versions of client code; however, their approach was only suitable for generating one-to-one rules. Wu et al. [Wu 2010] proposed a hybrid approach called AURA (AUtomatic change Rule Assistant) that combined call dependency and text similarity analyses. They compared their solution to three previous approaches by Dagenais et al. [Dagenais 2008], Kim et al. [Kim 2007], and Schäfer et al. [Schäfer 2008] and reported 58.07% higher recall and a similar precision. Nguyen et al. [Nguyen 2010] proposed a tool for library update (LibSync) that first uses the textual and structural similarities to find the mapping of the library functions and then extracts the usage graphs from already updated client systems and mines the transformations that need to be made to the client code. Meng et al. [Meng 2012] proposed a history-based matching approach HiMa which compares consecutive revisions of a library obtained from its commit history and supplies this information with the analysis of commit messages to generate transformation rules for client systems.

Teyton et al. [Teyton 2013] turned to the problem of library migration — replacing client dependency on a third-party library in favour of a competing library. They adopted and improved the approach of Schäfer et al. [Schäfer 2008], but extracted method call changes from a commit history of clients that were already migrated. Hora et al. [Hora 2014] proposed a similar approach to find method mappings between different releases of the same library. They analysed the commit history of a library to detect frequent method call replacements. This way, they mined the transformation rules by learning from how the library adapts to the changes of its own API.

The most recent research has focused on the problem of library migration. Pandita et al. [Pandita 2015] computed the textual similarity of documentation of the API entities from different external libraries and recommended the entities that were most similar to one another as possible replacements. They built a tool called TMAP (Text Mining based approach to discover likely API mappings) and used it to discover mappings from Java to C# API and from Java ME to Android API. Alrubaye et al. [Alrubaye 2019] mined the commit history of client systems that were already migrated from one third-party library to a different one and generated mappings for method replacements. They improved their results by calculating the textual similarity of method descriptions taken from library documentation. In their next study, Alrubaye et al. [Alrubaye 2020] proposed a novel machine learning approach, RAPIM, for the task of library migration. They extracted features such as the similarity of method signatures and documentation, represented them as numerical vectors, and trained a machine learning classifier to label method mappings as “valid” or “invalid”.

Shortcomings of existing studies. The first studies on library update were focused on the perspective of library developers. However, they only allowed them to record the changes or express rules, without proposing automated tools to guide them through this process by extracting knowledge from source code or the commit history. The following studies proposed such knowledge extraction tools for client developers, considering only the case when library developers are not available and their expertise cannot be utilised. In our work, we propose to build the tools for library developers that would help them understand the changes that were made to the library, document those changes, and propose replacements to the client. Another shortcoming is that most existing studies focus on statically-typed programming languages such as Java. This means that when they mine the rules for method replacements, they often have more information about the type of arguments, return type, and the type of object from which the method is invoked. To the best of our knowledge, the work that we present in Chapters 6 and 7 is the first attempt to target specifically the dynamically-typed programming languages such as Pharo, Python, Ruby, etc.

3.4 Chapter Conclusion

In this chapter, we overviewed the literature in the field of library update. This included the discussion of empirical studies of library evolution from the perspective of library an client developers, the studies on code transformation and deprecations that are related to our study on transforming deprecations (see Chapter B), and the studies that propose the tools to guide library or client developers in the process of library update. We identified several shortcomings of existing studies: (1) most existing studies focus only on the problems of client developers; (2) they do not consider the context of dynamically-typed programming languages; (3) most studies only consider simple method-to-method transformations. In the rest of this thesis, we will address those shortcomings and try to fill this gap in the literature.

CHAPTER 4

How Libraries Evolve: Developer Survey of Library Update

Contents

4.1 Introduction

The evolution of software libraries is a process that requires a joint effort of two groups of developers: the effort of *client developers* to update their dependencies to the latest versions and the effort of *library developers* to make the update process simple and clear.

To build better tools for library update and efficiently automate parts of this process, it is important to understand the behavior of both library and client developers, the practices that they adopt, the problems that they encounter. As we discussed in Chapter 3, several empirical studies were performed, based either on the analysis of source code [Hora 2015, Hora 2018, Jezek 2015, Kula 2018a, Kula 2018b, Robbes 2012a, Sawant 2016, Xavier 2017a] or surveying the developers [Kula 2018a, Bogart 2016, Brito 2019, Xavier 2017b]. However, those studies either focus only on one group of developers (library or client) or ask questions about specific breaking changes found in the code. Also, to the best of our knowledge, no previous study has explored the type of help that library developers can provide to their clients.

In this chapter, we present the results of two surveys: one of library developers and another one of client developers. Participants came from different backgrounds (open-source or industry) and different projects, the questions were general and not related to specific libraries or issues. Our study involved 18 library developers and 37 client developers. Although this population might be too small to claim statistical significance, it is comparable to other surveys in this field (*e.g.,* Bogart *et al.,* [Bogart 2016] surveyed 28 developers, Xavier *et al.,* [Xavier 2017b] — 7 dev., Kula *et al.,* [Kula 2018b] — 16 dev., and Brito *et al.,* [Brito 2019] — 102 dev.). The focus of our study is the perception of the impact of library evolution from different perspectives and the type of support that library developers can provide to help their clients. The results have been published at the technical track of the Asia-Pacific Software Engineering Conference (APSEC 2022) [Zaitsev 2022b].

The main contributions presented in this chapter are:

1. We performed two surveys: of library and client developers from two industrial companies and the Pharo open-source community.

2. We were the first ones to ask client developers about the type of support that they need from library developers.

3. We confirmed the results of the previous studies [Xavier 2017a, Brito 2019, Xavier 2017b] on the motivation of library developers to introduce breaking changes and their perception of impact.

The rest of this chapter is structured in the following way: In Section 4.2, we discuss the methodology of this study and list the research questions. In Section 4.3, we describe the population of developers who were selected for this survey. In the following two sections, we present the results from both of our surveys: first the library developer survey in Section 4.4 and then the client developer survey in Section 4.5. In Section 4.6, we discuss the threats to validity. Finally, in Section 4.7 we present the conclusions.

4.2 Survey Design

To address the shortcomings of the previous studies, we propose to conduct a survey of two groups: library and client developers. Each group should include developers from different projects and different backgrounds (*e.g.,* open-source and industry). The survey must ask general questions about the experiences and practices of developers and not be restricted to specific cases of breaking changes.

In our study, we consider the following research questions:

Library Developer Survey.

- RQ.1 How do library developers perceive the impact of library evolution on their clients?
- RQ.2 Why do library developers introduce breaking changes?
- RQ.3 How motivated are library developers to support their clients?
- RQ.4 How do library developers help their clients to update?

Client Developer Survey.

- RQ.5 How do client developers perceive impact of library evolution on their systems?
- RQ.6 What makes library update easy and what makes it hard?

Considering the relatively small population size, we did not perform statistical tests and derived conclusions from simple descriptive statistics. These conclusions lay out the paths for the follow-up investigations.

4.3 Describing the Population

We conducted a survey of developers from two industrial companies (Arolla and Berger-Levrault) and the Pharo open-source community. Both Arolla and Berger-Levrault are medium size software companies. The key difference is that Arolla is a consulting company where each developer works on a different project and Berger-Levrault is an international software company with multiple teams where developers work on their projects together. We selected those companies because of convenience: our research lab is part of the Pharo Consortium and many of our colleagues are members of the Pharo community. We also have common research projects with Berger-Levrault, and Arolla is the company that sponsored my PhD.

Developers selection. For Pharo and Arolla, we sent emails to the mailing lists and messages to the official online forums of each community. For Berger-Levrault, we contacted the director of research and asked him to spread the survey among his developers. In all three cases, participation was optional and all participants were informed that the survey would be anonymous. We asked every software developer (no matter what type of programming they do) to fill the client developer survey. As for the library developer survey, we explicitly asked it to be filled only by those developers who work on libraries, frameworks, microservices, or any tools that have API, several versions, and some users. Developers were free to answer both surveys. In general the libraries that client developers in our study use are not the same as the ones that library developer participants maintain.

Library Developers. We received the answers of 18 library developers, 11 of them belong to the Pharo open-source community, four are from Arolla, and one from Berger-Levrault. Two participants decided not to specify the community to which they belong. It should be noted that getting library developers in companies is rarer than library users. This proportion is different in the context of open-source because many open-source projects are proposing libraries to other developers (whose code may often be closed source).

We asked developers to evaluate their level of expertise as either *Absolute Novice* (0 dev.), *Beginner* (0 dev.), *Intermediate* (3 dev., 18%), *Advanced* (9 dev., 53%), or *Expert* (5 dev., 29%). Twelve developers work on open-source projects, five developers work on closed-source projects, and one developer works on both. We asked developers to specify the kind of software that they develop. Their answers can be seen in Figure 4.1: fifteen paticipants develop libraries, ten developers work on frameworks, four work on SDK, and four on microservices. Three developers selected *"Other"* as their type of project.

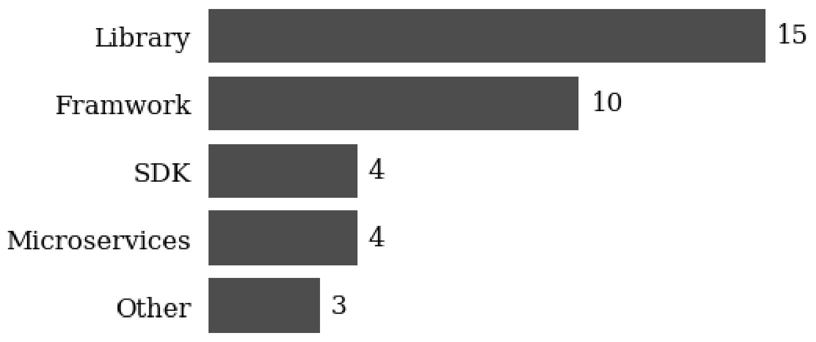


Figure 4.1: The types of software developed by the library developers who took part in our study.

We also asked library developers to specify approximately how many client developers (users) they have. The estimates scattered among the following options: 1 to 10 clients (6 dev., 33%), 11 to 50 clients (5 dev., 28%), 51 to 100 clients (3 dev., 7%), 101 to 500 clients (4 dev., 22%). No library developer in our survey claimed to have more than 500 clients.

As will be discussed in Section 4.6, we acknowledge that the relatively small size of libraries developed by the participants of our survey may pose a threat to

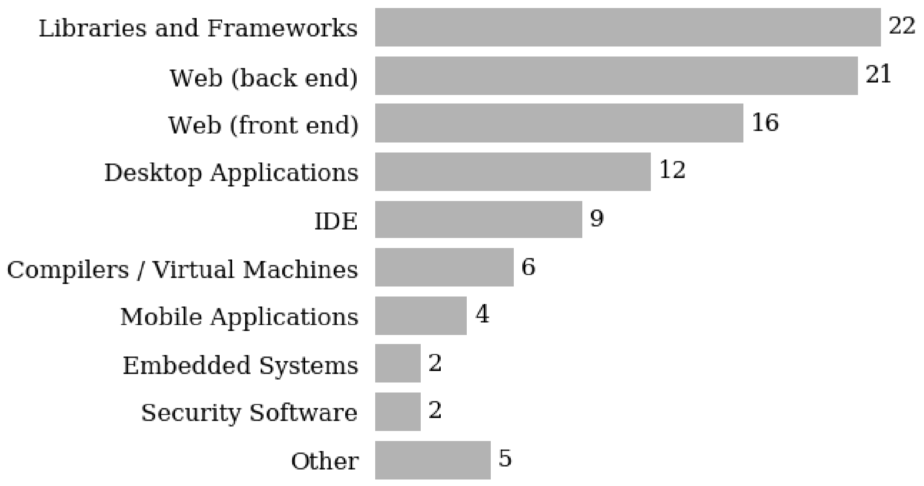


Figure 4.2: The types of software developed by the client developers who took part in our study.

external validity. But we also consider this to be a novelty because the surveys that only focus on large or popular open-source libraries are also biased by their size.

Client Developers. We received 37 answers from client developers, 22 of which were from the Pharo community, five from Arolla, and four from Berger-Levrault. Also, four developers preferred not to specify their affiliation, and other two developers listed other open-source communities that they belong to. As can be seen in Figure 4.2, our selection covers a variety of domains by the types of projects that client developers work on. They range from library, frameworks, web, desktop applications but also tool development and even compiler ones. As we mentioned before, the developers in our study were free to participate in both surveys. This and the fact that most respondents come from the open-source community, may explain that 22 out of 37 client developers also develop libraries. We further discuss this in Section 4.6. We asked client developers to evaluate their level of expertise as either *Absolute Novice* (0 dev.), *Beginner* (2 dev., 6%), *Intermediate* (6 dev., 18%), *Advanced* (18 dev., 53%), or *Expert* (8 dev., 24%). We also asked client developers to specify approximately how many dependencies they have. The most popular answers were 1 to 10 (15 dev., 42%) and 11 to 50 (16 dev., 44%). Four developers said that they have between 51 and 100 dependencies and one developer claimed to have more than 100 dependencies.

4.4 Results of the Library Developer Survey

Now we present the results of the survey following the research questions mentioned before. To answer research questions that are rather general, we had to ask specific questions in our questionaires. That is why, each *research question* in our study is associated with one or more *survey questions*. In this section, we discuss the results of a library developer survey and in Section 4.5, we will present the resuts of a survey of client developers.

4.4.1 RQ.1 - How do library developers perceive the impact of library evolution on their clients?

To answer this research question, we asked two questions in our survey: one to assess their perception of the impact of breaking changes on client systems and one to understand how they estimate the time that clients need to update. The answers are presented in Figures 4.3 and 4.4.

As can be seen in the figures, both questions were answered by all 18 library developers who took part in this survey. Nine of them believe that the impact of breaking changes on their clients is *moderate*, six (33%) that it is *big*, and two (11%) that it is *very big*. One developer (6%) assessed the impact as *very small*.

As for the time that is needed to update the client systems to the new versions of their libraries, opinions of library developers are different. When presented with a five-option scale between *less than an hour* and *a week or more*, the estimates scattered among them with *less than an hour* being the most popular option, selected by seven developers (39%), and *a week or more* being the least popular one, selected by one developer (6%). We hypothesise that different answers to this question are caused by different complexity of changes in library releases. Some library updates are simple and can be done in less than an hour. Others are hard and may require a week or more.

The answers to the second question about the time to update could have been influenced by *Deprewriter* [Ducasse 2022] — the automatic deprecation update mechanism that is used by the Pharo community. It allows library developers to annotate method deprecations with transformation rules that will be used to automatically rewrite the call-sites in the client code. This mechanism considerably eases the update of deprecated methods. Since many of the library developers answering the survey were members of the Pharo community, this is a factor to take into account. Deprewriter will be presented and discussed in Chapter 5.

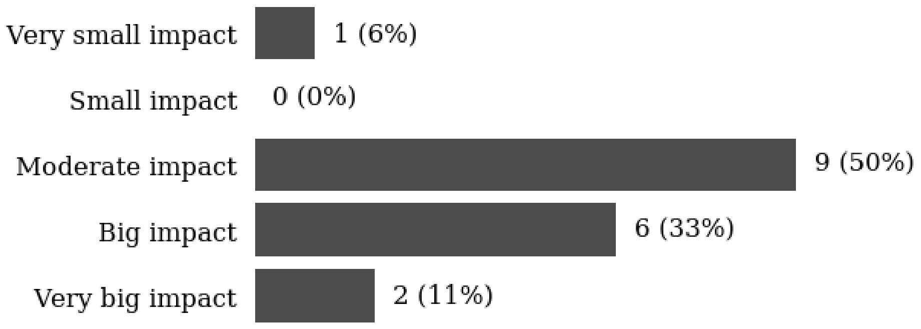


Figure 4.3: Do you think that breaking changes in your releases have big impact on clients?

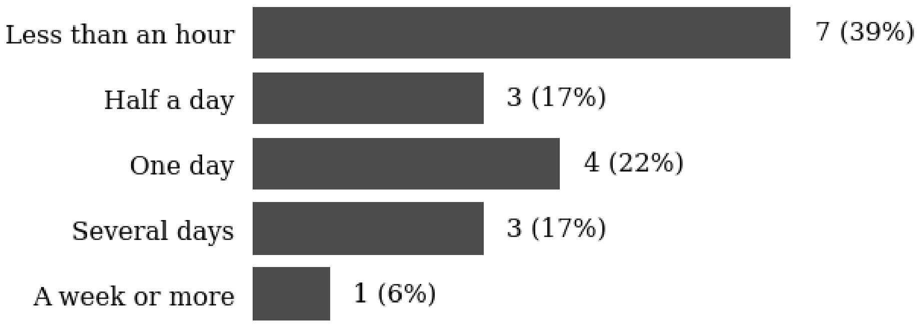


Figure 4.4: How much time do you think client developers need to update to the new version of your library?

Summary:

- Most library developers in our survey agree that the impact of breaking changes on their clients is not small: 50% believe that the impact is moderate, 44% say that the impact is big or very big.
- Library developers had different opinions on how long it takes their clients to update. The most common estimate is less than an hour (39%) but some developers claim that it can take multiple days.

4.4.2 RQ.2 - Why do library developers introduce breaking changes?

This question has previously been answered by Brito *et al.,* [Brito 2019] using the firehouse interview (sending an email to the developer as soon as the breaking change was detected). They reported that the most common reasons for introducing breaking changes are the need to implement new features (32%), API simplification (29%), and improving maintainability (24%). To verify those results, we asked a similar question in our survey of library developers. This was a checkbox question and the list of options was the same as the factors identified by Brito *et al.,* [Brito 2019]. We also added an *"Other"* option and an optional open text field to identify additional factors. All 18 library developers answered this question. As can be seen in Figure 4.5, 12 developers identified *implementing a new feature* as a primary reason for introducing breaking changes; 11 developers identified *API simplification* and 11 developers selected *improving maintainability*; six developers selected *bug fixing*. This confirms the results that were previously reported by Brito *et al.,* [Brito 2019]. Two developers also provided additional reasons: *research transfer* and *performance*.

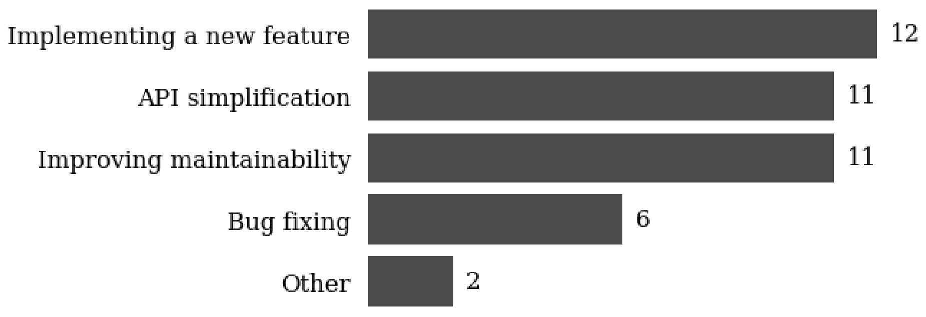


Figure 4.5: What are your primary reasons for introducing breaking changes?

We also asked developers an open question: *"Were there specific scenarios when you had to sacrifice backward compatibility to introduce important changes?*

Can you describe them briefly?”. We received 15 answers to this question. In addition to the four reasons listed above, three developers wrote that breaking changes are caused by *system redesign and architectural changes*, two developers mentioned *refactorings*. Also, three developers wrote that breaking changes in their APIs were caused by changes in their dependencies — a process that is known as the *ripple effect*, when a change in one library can impact other libraries that depend on it and propagate through the ecosystem from client to client. Two developers mentioned security issues related to authentication and one developer wrote that they break the API when they need to improve names or remove features that are not used.

Summary: The most common reasons for introducing breaking changes are the *implementation of new features* (12 dev. out of 18), *API simplification* (11 dev.), and *improving maintainability* (11 dev.), *bug fixing* was selected by six developers. This confirms the results of the previous study conducted by Brito *et al.,* [Brito 2019]. Among other reasons mentioned by developers are *architectural changes*, *refactorings*, *security issues*, and *changes in other libraries*.

4.4.3 RQ.3 - How motivated are library developers to support their clients?

As we discussed in Chapter 3, recently there were many studies that proposed automated tools to support developers in the process of library update. Some of those approaches are designed for client developers, while others are for library developers. For the second group of approaches, it is important to understand what is the motivation of library developers to support their clients.

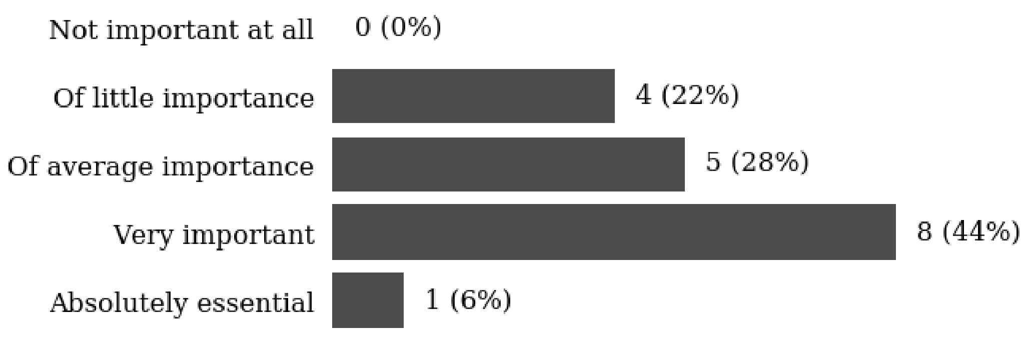


Figure 4.6: How important is it for you to maintain backward compatibility?

To answer this question, we asked library developers two questions: *“How* *important is it for you to maintain backward compatibility?"* and *"How important is it for you to encourage clients to update to the latest version?"*. Those questions are different, because, as we seen in Section 4.4.2, library developers have many good reasons to introduce breaking changes (and thus break compatibility) even if they want their clients to update easily.

In Figure 4.6, we present the answers to the first question. Four developers (22%) specified that it is of little importance. For five developers (28%), backward compatibility is of average importance. Eight of the surveyed developers (44%) stated that it is very important for them, and for one developer it is absolutely essential to be backward compatible.

In Figure 4.7, we present the answers to the second question. Only one library developer specified that encouraging clients to update is of little importance. For five developers (28%) it is of average importance, for eight developers (44%) it is very important, and for four developers (22%) it is absolutely essential.

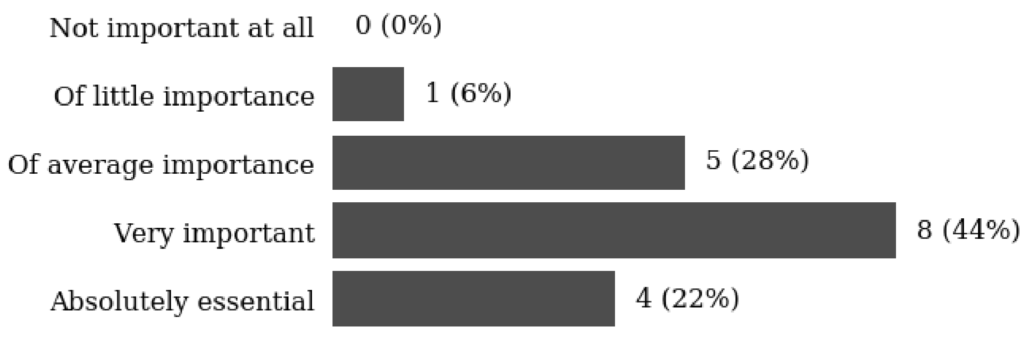


Figure 4.7: How important is it for you to encourage clients to update to the latest version?

Summary:

- Maintaining backward compatibility is very important or absolutely essential for 50% of surveyed library developers.
- Encouraging clients to update is very important or absolutely essential for 66% of surveyed library developers.

4.4.4 RQ.4 - How do library developers help their clients to update?

To understand what library developers do to help their clients, we asked two questions in our survey. First, a checkbox question (multiple selection) intended to understand the software development practices adopted *before* the release: *“What software development practices do you use to improve the stability of your API?”*. Second, an open question about the practices *after* the release: *“When you are forced to break backward compatibility, what do you do to reduce the negative impact on users?”*.

The first question was answered by all 18 developers. We gave them three options: *weak coupling* (selected by 15 dev.), *abstraction layer* (13 dev.), and *microservice architecture* (5 dev.). In the *“Other”* section of this question, developers mentioned three more practices: *“design patterns”*, *“automatic transformation of deprecated methods”*, and *“independent software that checks the stability of the API and the continuity of its expected behaviour”*.

The open question was answered by 16 out of 18 surveyed developers. We analysed their responses and identified four practices that library developers use to reduce the negative effect of breaking changes on their clients:

- **Documentation** (8 dev.), including *migration guide* (4 dev.), *release notes* (1 dev.), and *deprecation comments* (1 dev.).
- **Deprecation** (4 dev.).
- **Automation** (4 dev.), including *rewriting deprecations* (2 dev.).
- **Communication** (3 dev.): including *communication before making the change* (1 dev.) and *live workshops to help clients update* (1 dev.).

Two developers explicitly said that they do nothing to help client developers. One developer, who mentioned that he/she uses the tools for automatic adaptation of source code, has also expressed discontent with such automation techniques: *“... usually, the devs prefers to see exactly how the code will be modified in the context of the application. When the change requires actions on many parts of the software, manually going through all the modications by hand help to put again "in context" the impact of the modification”*.

Summary:

- Among library developers who answered our survey, the most popular practices to improve the stability of API are *weak coupling* (18 dev.) and *abstraction layer* (15 dev.). Developers also mentioned the *microservice architecture*, *design patterns*, and *automation tools* such as rewriting deprecations and API stability checks.
- The most common practices to support clients after the introduction of breaking changes are: *documentation* (8 dev.), *deprecation* (4 dev.), *automation* (4 dev.), and *communication* (3 dev.).

4.5 Results of the Client Developer Survey

In this section we present and discuss the results of the survey of client developers. As before, each research question is associated with one or more survey questions.

4.5.1 RQ.5 - How do client developers perceive impact of library evolution on their systems?

We asked the client developers two questions that mirror the questions that were answered by library developers and discussed in Section 4.4.1. Each question was answered by 36 out of 37 client developers that took part in our survey.

First, we asked client developers to estimate, how much they are affected by the evolution of their dependencies. Notice that this question is slightly different from the library developer question in which we asked about the impact of breaking changes. As can be seen in Figure 4.8, the answers are almost equally distributed around the midpoint *somewhat affected* (14 dev.): 11 developers answered that they are *affected a little*, 10 developers — *significantly affected*, one developer claims to be *affected to a great extent*.

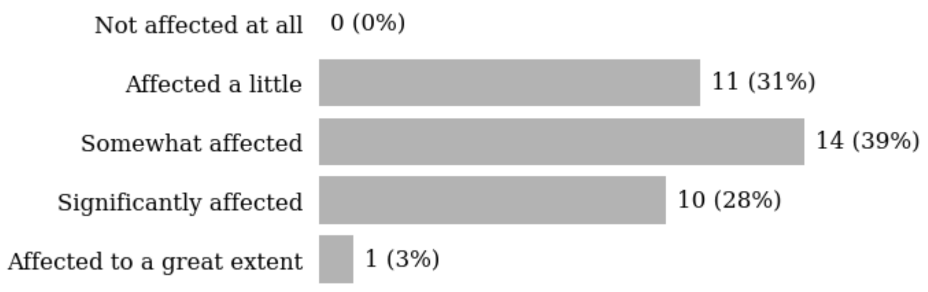


Figure 4.8: How much are you affected by the evolution of your dependencies? (e.g., when one of your dependencies releases a new version or drops support for the old one)

Then we asked client developers to estimate how long it usually takes them to update a dependency. As can be seen in Figure 4.9, the answers are scattered between *"less than an hour"* and *"several days"*. Only one developer answered *"one week or more"*. This trend is similar to the one we observed in Section 4.4.1. Again, we hypothesise that the different answers are caused by different complexity of changes in libraries on which clients depend.

Finally, we asked client developers to estimate how often they have to deal with the task of updating dependencies. As can be seen in Figure 4.10, out of

36 client developers who answered this question, 17 (47%) have to update their dependencies three times a year or more often, 10 (28%) have to do it twice a year.

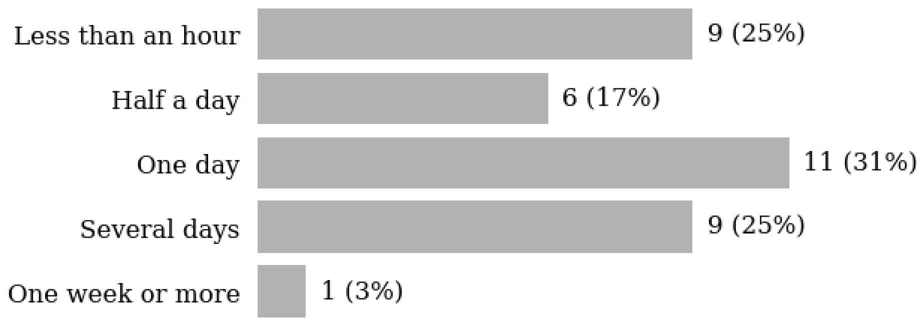


Figure 4.9: How much time does it usually take you to update your dependencies?

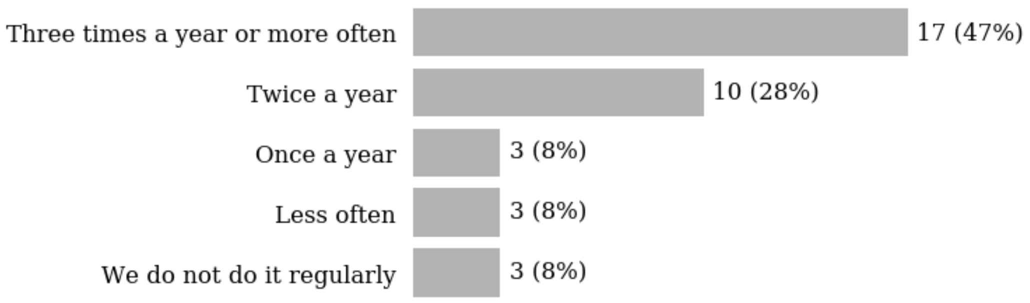


Figure 4.10: Try to estimate how often do you have to deal with the task of updating dependencies

The other two options: once a year and less often each have three developers who selected them. Also, three developers stated that they do not update their dependencies regularly.

Summary:

- Most client developers in our study do not think that they are greatly affected by library evolution.
- The time required to update a library dependency can be different: from less than an hour to several days. Only one developer claimed that it may take a week or more.
- In our study, 27 out of 36 client developers have to update their dependencies at least twice a year; 17 developers do it three times a year or more often.

4.5.2 RQ.6 - What makes library update easy and what makes it hard?

To answer this research question, we asked client developers two open questions in the survey: *"When updating a dependency is easy, what makes it easy?"* and *"When updating a dependency is hard, what makes it hard?"*. Each question was answered by 34 out of 37 client developers who took part in the survey. We analysed their answers and summarise them in Tables 4.1 and 4.2.

Table 4.1: When updating a dependency is easy, what makes it easy? Second column is the number of developers who mentioned this factor in an open question.

Factor	**dev.**
Documentation	15
Absence of breaking changes	11
Test coverage	6
Tool support	5
Deprecations	4
Simple breaking changes	4
Community support	3

According to our respondents, the main factors that make library update easy are good *documentation* (mentioned by 15 developers, 44%) and the *absence of breaking changes* (11 dev., 32%). Other factors include *test coverage* of client code, *tool support* such as dependency managers and automated code rewriting, *deprecations* that are introduced before removing functionality, *simple breaking changes* such as method renaming, and *community support*. The most commonly mentioned factors that make library update hard are *breaking changes* (mentioned by 11 developers, 32%), *absent or bad documentation* (10 dev., 29%), *indirect dependencies* to other libraries that can result in version conflicts (7 dev., 21%), and *big changes to the API* that force clients to change the logic of how the library is used (7 dev., 21%). Other factors that were mentioned 2-4 times are *poor test coverage* of client code, *removed functionality*, *changed hooks and abstract classes*, *absence of community support*, and *behavioral changes*. There are also factors that were mentioned only once: strong coupling, expertise required, security issues, naming collisions with other libraries, absence of deprecations, bugs and compilation errors. We did not include those factors in Table 4.2.

Table 4.2: When updating a dependency is hard, what makes it hard? (only factors that were mentioned by at least 2 developers).

Factor	**dev.**
Breaking changes	11
Absent or bad documentation	10
Indirect dependencies	7
Big changes to the API	7
Poor tests coverage	4
Removed functionality	3
Changed hooks or abs. classes	3
No community support	2
Behavioral changes	2

Summary: The two most important factors affecting the complexity of library update are breaking changes and documentation.

- 11 out of 37 client developers in our study mentioned that breaking changes make updating hard, and also 11 developers mentioned that their absence makes it easy;
- 15 developers mentioned that good documentation makes updating easy, and 10 developers mentioned that missing or misleading documentation makes it hard.

4.6 Threats to Validity

We discuss the four types of validity threats that were presented by Wohlin *et al.,* [Wohlin 2000]: internal, external, construct, and conclusion validity.

Internal Validity

- We did not ask the participants to focus on a single project but rather to answer the questions based on their general experience. This makes it hard to further analyse the contexts that led to certain situations/decisions.
- Some questions in our survey are hard to answer generally. The answers can vary depending on the project that developer has in mind or specific situations on the client side. For example, when we asked how much time it takes for clients to update, the answer could depend on how much of the API was used by a particular client.
- The transforming deprecation mechanism of Pharo [Ducasse 2022] may have affected how Pharo developers answered this survey.
- Inside the three mentioned communities, the surveys were public and participation was optional. This means that many library developers could also have answered the client developer survey. We can not verify this because the surveys were fully anonymous and we did not want to restrict participation to only one survey because each library developer might also be a client developer for other libraries.

External Validity

- For convenience, in this study we surveyed developers from an open-source community that we (authors of the study) are part of, and two industrial companies that collaborate with our research group. The participation was optional, which means that the developers who responded to our call might be the ones who know us personally and have interest in our research. This might have introduced a bias.
- The library developers in our survey are responsible for libraries with no more than 500 clients. Those libraries can be considered small compared to the top-1000 libraries in NPM and Maven. This means that our study may not be representative of the large libraries and frameworks, but more focused on the smaller libraries that are also more common. Although, we list it as a threat, we also believe this to be a novelty of our study because most related work focuses only on large libraries, which introduces a bias on their side.

Construct Validity

- Some research questions in this survey can not be fully expressed with specific survey questions. For example, to measure the impact of library evolution on clients, we ask questions about the perception of this impact (too general) and another question about the time it takes to update (too specific).

- To save the time of our participants, most questions in our survey were not open but contained a list of options to choose from. This poses a threat to construct validity because developers were limited and biased by those options. To address this threat, we tried to provide a diverse list of options for each question, taking inspiration from the literature (*e.g.,* the options for RQ2 were the same as the ones identified by Brito *et al.,* [Brito 2019]) or by discussing them with our colleague developers.

- The perception of the impact of breaking changes may tell us about the way client uses the library rather than the actual impact.

Conclusion Validity

- The population size is relatively small. Our study involved 18 library developers and 37 client developers. Although it is comparable to other surveys in this field (*e.g.,* Bogart *et al.,* [Bogart 2016] surveyed 28 developers, Xavier *et al.,* [Xavier 2017b] — 7 dev., Kula *et al.,* [Kula 2018b] — 16 dev., and Brito *et al.,* [Brito 2019] — 102 dev.), this population might be too small to claim statistical significance.

- Considering the relatively small population size, we did not perform statistical tests and derived conclusions from simple descriptive statistics.

4.7 Chapter Conclusion

In this chapter, we presented the results of a first general survey of the perception of library evolution by both client developers (who need to update their software to new versions of libraries they use) and library developers (who produce new versions of libraries). The survey involved software developers from two industrial companies: Arolla and Berger-Levrault, and one open-source community: Pharo. We have confirmed the results of a previous study performed by Brito *et al.,* [Brito 2019] who reported that the most common reasons for introducing breaking changes are the addition of new features, API simplification, and maintainability improvement. We have also identified what makes library update easy and what makes it hard for clients, and the kind of support library developers can provide them. The survey results presented in this chapter can help better understand the process of library update from the perspective of library and client developers.

CHAPTER 5

Deprewriter: Transforming Deprecations

Contents

In this chapter, we discuss the powerful mechanism of transforming deprecations that is supported by Pharo. It allows library developers to annotate method deprecations with transformation rules which can be automatically applied to client code. The technique of transforming deprecations existed in Pharo for several years. However, it was not well documented. Besides, only part of this functionality was used by the community and only in a small circle of people. To that end, we have published a paper in the Journal of Object Technologies [Ducasse 2022] in which we document the transforming deprecations in Pharo, study how it is used in practice, and identify the limitations that can be addressed in the future versions. In this chapter, we introduce *Deprewriter* and present our findings.

5.1 Introduction

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 (*e.g.,* in the form of annotations); thus, compilers and IDEs will report them to the client developer [Brito 2018a]. However, it is up to the developers of client applications to manually transform and update their code to the new APIs [Kim 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 chapter, we discuss Deprewriter (which stands for DEPrecation REWRITER): 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 calls from the invocations of other method with the same name. Since 2016, Deprewriter is used in production in the Pharo distributions: Pharo 6, 7, 8, and 9 alpha.

Using Deprewriter:

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.

When a deprecated method is executed, Deprewriter walks the execution stack to find the caller method and the call site from which the deprecated method is invoked. 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 can 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.

The implementation of Deprewriter in Pharo is based on call-stack reification and navigation [Rivard 1996], dynamic program update, and tree matching rewriting [Roberts 1996, Roberts 1997], however the approach is generic enough to be implemented using different mechanisms as explained in Section 5.6.

As we have discussed in Chapter 3, in recent years, many studies have shown how the library update rules can be extracted from source code [Kim 2007, Xing 2007, Schäfer 2008, Wu 2010, Nguyen 2010, Pandita 2015] and the commit history [Dagenais 2008, Meng 2012, Teyton 2013, Hora 2014, Alrubaye 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 rules. The Deprewriter approach, which we present in this chapter, 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 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.

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 developers (*e.g.,* the Pharo consortium) and the second category is about the perception by the users of rewriting deprecations (both library and client developers). 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 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 library developers who write the deprecation rules for Deprewriter and client developers who use the libraries with deprecated source code. Of all 46 persons, 28 (61%) report that rewriting deprecations helped them while 10 (22%) state the inverse and 8 (17%) are uncertain. Additionally, by analysing the answers to the open questions, we conclude that most developers who took part in our survey consider Deprewriter to be beneficial in their work. They also acknowledge the advantages of using unit tests to exercise automatic deprecations, concluding that having a good test coverage helps with library update.

This chapter is based on a paper that we published in the Journal of Object Technologies [Ducasse 2022]. Its key contributions are:

- First thorough description of the run-time deprecation rewriting 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 who use Deprewriter either as client, or library developers, or both.

The rest of this chapter is structured in the following way: In Section 5.2, we discuss the challenges of automatically rewriting code in dynamically-typed languages. In Section 5.3, we present Deprewriter from the perspective of the library and client developers. Section 5.4 describes the architecture and a 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. In Section 5.5, we discuss the key aspects of the current implementation. Since our approach is not bound to Pharo, Section 5.6 presents sketches of implementation using, for example, AOP. As validation of the approach, Section 5.7 presents an analysis of the rewriting and non-rewriting deprecations in Pharo 8. Section 5.8 presents the second part of the evaluation: we report the results of a survey that was completed by 46 Pharo developers. In Section 5.9, we discuss the limitations of our approach and the future work.

5.2 Problem: Replacing Deprecated Method Calls

5.2.1 The Difficulties of Dealing With Deprecated Methods

Deprecations are a powerful tool for reducing the negative effect of breaking changes on client systems. They inform clients that the feature will be removed and give them time to react. Nonetheless, it is not always evident to client developers what kind of reaction is expected of them. For example, consider a client system that calls a method `name()` to access the name of a product. At a certain point, developer of this system receives a deprecation warning informing him or her that method `name()` is obsolete and will be removed in the upcoming release of the library. Now it is not clear to the client developer, (1) why will this method be removed and (2) what is the new way of accessing the name of a product. If method was renamed then what is the new name that should be used instead? Perhaps, the name of a product should now be accessed as a property of a more complex object or through an HTTP request. Or maybe the products in the new version of a library do not have names.

Library developers who introduce the deprecation usually know the answers to those questions (the case when library developers do not know the answers will be discussed in Chapter 6). They can share this information with client developers in a form of deprecation messages, comments, or documentation. We identify three different scenarios:

1. Deprecated method can be replaced with one or more other methods and this replacement can be automated.
2. Replacement exists but it is complex and cannot be applied automatically.

For example, it has to be adapted to each specific case.

3. The deprecated method should not be used and there is no replacement.

The most common type of breaking changes are small structural changes such as method renamings [Dig 2006b]. Therefore, in most cases, replacements exist and they are automatable, for example: *"Method a() is deprecated, use b() instead."* In those cases, client developers could benefit from transformation rules in the form `a` → `b` that could be used to automatically rewrite their code. In this chapter, we will present such a technique.

5.2.2 Static Analysis for Dynamically-Typed Languages

Static analyzers are useful to identify and transform the 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 by static analyzers. For example, consider Figure 5.1 showing the `log:` methods coming from two different libraries: a logging library and a math library, and finally the client class `User` that calls both methods. 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 warning should be signalled 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 5.1: The limitations of static analysis. An analyzer cannot 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:`. To fix the client code, not only does one need 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 method `name` is called 3109 times and implemented in 346 classes, `isEmpty` has 1595 callers and 103 implementors. In addition, inheritance and polymorphism across different hierarchies produce code where a single method call can invoke multiple implementations during execution.

The research community has proposed to use type inference for dynamically-typed languages [Suzuki 1981, Furr 2009, Ren 2016, Spoon 2004, Pluquet 2009, Passerini 2014] or to use dynamic type information collected by the virtual machine to get concrete types [Milojković 2016]. Such type inferencers often do not cover the whole language [Suzuki 1981] or are not applicable to large code-bases [Spoon 2004]. Type speculation combined with runtime statistics is used to implement speculative just in time compilers and speculative optimizers [Hölzle 1991]. However, when such information is not available, polymorphic methods force developers to manually identify and replace the deprecated calls, leading to the introduction of bugs.

Deprewriter, that will be presented in the following sections, uses runtime information to identify the correct call sites to rewrite. It is based on call-stack navigation and program transformation at runtime.

5.3 Example: Rewriting Deprecations In Action

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 developer.

5.3.1 Library Developer Perspective

In this section, we explain what a library developer should do to deprecate a method. Library developers can mark the given method as deprecated by calling the special method `deprecated:transformWith:` from anywhere inside the method body. This method has two arguments. The first one is a string explaining the deprecation. It will be displayed to the client of the library when the deprecation warning is signalled or logged if the deprecation is applied silently. The second argument is a transformation rule that will be used to automatically rewrite the call sites of the deprecated method.

In the following code listing, we demonstrate the deprecation of the method `log:` of `Logger` class 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
```

- Line 1 defines the signature of the deprecated method.
- Lines 4 to 7 define the deprecation and its companion transformation rule
  - Line 5 defines the text that will be used 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 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 consequent expression with the matched variables expanded. In this case, any expression *e.g.,* `expr1 log: expr2` will be rewritten to `expr1 log: expr2 level: #info`.
- Finally, after the deprecation definition, Line 10 calls the method from the new API. It can also contain the body of the old method. It 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 the domain specific language (DSL) of an embedded parse-tree rewriter, which is part of the Refactoring Engine developed by J. Brant and D. Roberts [Roberts 1996, Roberts 1997, Roberts 1999, Renggli 2010a]. This DSL and its syntax are not part of the contributions of our article nor of this thesis. We only documented it and used it in our study. The language for expressing transformation rules will be discussed in more detail in Section 5.4.2.

5.3.2 Client Developer Perspective

From the perspective of a client developer who uses the deprecated API, 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.

Although library developers are free to place the deprecation in any part of the method body, the most common practice is to place deprecation at the very top of the method, before the rest of its body. In this case, at runtime, when the deprecated method is executed, the following steps occur:

1. Deprewriter is called before the rest of the method's body is executed.
2. It accesses the caller method and the exact call site from where the deprecated method was invoked — Pharo implementation introspects the call stack, but alternate approaches such as AOP are possible (see Section 5.6).
3. It then triggers 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. Another good practice is to use this to call the new non-deprecated code.

In the `Logger` example (see the code listing in Section 5.3.1), 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 can 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.

5.4 Deprewriter

The approach of Deprewriter has the responsibility to perform three main tasks.

- Detect the call sites of the deprecated methods (Section 5.5.1).
- Provide a way for the library developers to express the transformation rules and then apply these rules to the identified call sites (Section 5.4.2).
- Provide a way for the client developers to select 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 5.4.3).

5.4.1 General Architecture in a Nutshell

Our approach works as follows (See Figure 5.2):

1. Library developer deprecates a method and provides a transformation rule, as shown by (Step 1) in the figure.
2. During the execution of a client system, the deprecated method is invoked and it raises an exception (Step 2).
3. Deprewriter captures the exception (Step 3).
4. When configured to rewrite on the fly, Deprewriter identifies the call site in the caller method and rewrites it using the transformation rule (Step 4).
5. The execution of the deprecated (halted by exception) method is resumed. The body of the deprecated method is executed (Step 5): one common practice is to simply call 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.

We further discuss the key implementation points in Section 5.5.

5.4.2 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 [Roberts 1996, Roberts 1997, Roberts 1999].

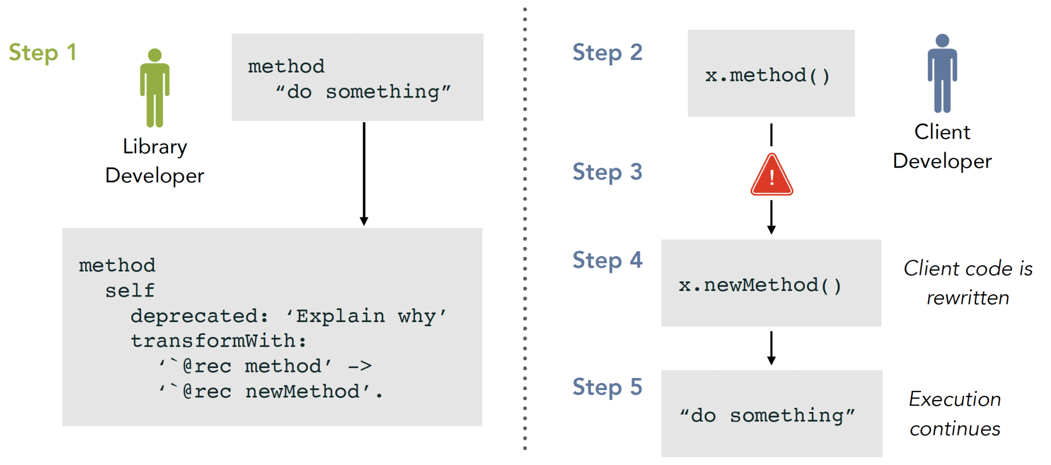


Figure 5.2: Two stages of rewriting deprecation: declaration and execution. At execution, deprecated method callers are rewritten and execution continues.

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` = `(x 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 8 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 10 to 13 check that the rewriter can effectively transform the code and reformat the dynamic array: here {(1 @ 255). (Color lightMagenta). 3}

Finally lines 15 to 20 verify that the result is correct.

```
1  testRewriteDynamicArray
2    | newSource |
3    rewriter := RBParseTreeRewriter new
4      replace: '{`@first. `@second. `@third}'
5      with: 'Array
6        with: `@first
7        with: `@second
8        with: `@third'.
9
10   newSource := (rewriter executeTree:
11     (self parseRewriteExpression:
12       ' {(1 @ 255). (Color lightMagenta). 3}'))
13     ifTrue: [ rewriter tree formattedCode ].
14
15   self
16     assert: newSource
17     equals: 'Array
18       with: 1 @ 255
19       with: Color lightMagenta
20       with: 3'.
```

One particular aspect of ParseTreeRewriter is that it extends the syntax of the language. Programmers can simply take their code, 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 2015]. Alternate extensible solutions have been proposed to support programmers during the definition of transformation patterns [Rizun 2016], but they require the use of a tool to help generating the rules.

5.4.3 Configuring the rewriting process

The user has the possibility to configure the rewriting process:

- *Automatic rewriting* — all the rewriting deprecations will automatically rewrite their callers.

- *Warning* — developers are warned interactively that their code is invoking deprecated methods, but the caller will not be automatically rewritten.
- *Doing nothing* — with this configuration, the deprecated methods are executed normally, with no warning nor rewriting.
- *Logging* — developers can log the deprecated calls (and transformations done). This option can be combined with other three options.

5.4.4 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 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.

5.4.5 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 5.7) and partially inspired by the lists of refactoring operations that were analysed by Murphy-Hill *et al.,* [Murphy-Hill 2011] and Dig *et al.,* [Dig 2006b].

Rename method — method renaming is the most common refactoring operation [Murphy-Hill 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` $\rightarrow$ `‘@receiver newMethodName: ‘@arguments`.

Remove argument(-s) — similar to renaming[1], when one or multiple arguments are removed from a method, the old method name can be deprecated. This scenario can always be supported with a transformation rule in the form `‘@receiver oldMethodName: ‘@oldArguments` $\rightarrow$ `‘@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` $\rightarrow$ `‘@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 2009]. For example,

[1]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.

`‘@receiver evaluate: ‘@argument` $\rightarrow$ `‘@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 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.

5.5 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 5.6, we also present some sketches of possible implementations using exception or AOP [Kiczales 2001, Colyer 2005, Chern 2007].

5.5.1 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 ("parent") stack frame. This gives us access to the whole chain of stack frames. Using this feature, we access the stack frame of the caller of the deprecated method. Also, the reified stack frame has the responsibility to resolve its activating 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 5.3, the execution of a deprecated method will invoke the method `deprecated: transformWith:` with two arguments: `anExplanationString` and `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.

```
Object >> deprecated: anExplanationString transformWith: aRule
        Deprecation new
                context: thisContext sender;
                explanation: anExplanationString;
                rule: aRule;
                transform
```

Figure 5.3: The method `deprecated: transformWith:`

Rewriting steps. The `transform` method does the following (Listing 5.4):

- Lines 3-4: check if the deprecation 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 6: the caller method is identified. Given the method that invoked the `deprecated:transformWith:`, the method `contextOfSender` will find the stack element representing the caller.

- Line 7: ignore code snippets because during the execution they are mapped to anonymous methods. Deprewriter does not rewrite snippets.

- Line 9: get the program node of the AST representing the caller of the deprecated method.

- Lines 11-12: the transformation rule (called a rewrite rule here) is created based on the specifications of the deprecation.

- Lines 14-15: check if the rule can be applied to the node by calling the method `executeTree:`. When the node cannot be rewritten, the default exception is raised.

- Lines 17-21: if 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. The implementation is relatively simple. It assembles existing functionality available in the language: exception mechanism, call-stack reification, and a parse tree rewriting engine.

5.6 Sketches of Possible Alternative Implementations

The approach presented in this chapter 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 the caller method to use the newer versions of deprecated methods.

To show the generality of our proposed solution, in this section, we present 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.

5.6.1 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.

```
Deprecation >> transform
  | node rewriteRule aMethod |
  self shouldTransform
    ifFalse: [ ↑ self signal ].

  aMethod := self contextOfSender method.
  aMethod isDoIt ifTrue: [ ↑ self ].

  node := self contextOfSender sourceNodeExecuted.

  rewriteRule := self rewriterClass new
      replace: rule key with: rule value.

  (rewriteRule executeTree: node)
       ifFalse: [ ↑ self signal ].

  node replaceWith: rewriteRule tree.

  aMethod origin
      compile: aMethod ast formattedCode
      classified: aMethod protocol.
```

Figure 5.4: Method `transform:` — the core of the rewriting behavior.

Call-Stack Reification. Our proposed solution is implemented based on the call-stack reification [Rivard 1996] mechanism that is present in Pharo. This mechanism allows the inspection and manipulation of the current execution 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. The Aspect-Oriented Programming (AOP) [Kiczales 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 their source code. This information is expressed in metadata that is read by the AOP framework. The methods are modified by the AOP framework during the weaving process, without modifying the original source code. Also, AOP frameworks allow 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 can replace our inspection of the current call-stack.

Exceptions with Stack Trace Information. As we mentioned earlier, having high-level support for inspecting the call-stack is not needed for implementing our approach. It is possible to identify the caller method using exception handling and logging. The signalled exception will capture the current stack trace information, which would allow to extract the caller method. Listing 5.5 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).

5.6.2 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 uses tree matching rewriting [Roberts 1996, Roberts 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.

```
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 5.5: Java pseudo-code showing how to identify caller method by only using existing exception support.

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).

Dynamic Software Support in Tools. In the scenario we are using a language that does not natively support DSU. However, we are still 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 2012], Ginseng [Neamtiu 2006]) to object-oriented languages running in a VM like Java (*e.g.,* JRebel [ZeroTurnAround 2012], Rubah [Pina 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.

5.7 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,[2] the latest stable version of the Pharo Project.[3]

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?

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

[3]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.

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.

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 (for example, `deprecatedMethod1`, `deprecatedMethod2`). We also removed 2 deprecations that contained an invalid transformation rule, based on the validity criteria discussed in Section 5.4.4. 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 compute 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.

Table 5.1: Six deprecation selectors available in Pharo 8 together with number of senders.

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

In Pharo, there are six selectors that can be used to deprecate a method. In Table 5.1, 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 5.1, 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 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.

RQ2. What are the different types of deprecation scenarios that can be supported by Deprewriter? To answer this question, 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* $\rightarrow$ *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 5.4.5. The summary of this classification is presented in Table 5.2. 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. Out of 218 rewriting deprecations 179 (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

Table 5.2: 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.

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

rules that we have collected from Pharo 8 can be found in Table B.1 in Appendix B.

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 the fourth column of Table 5.2, we present the number of non-rewriting deprecations that we found in Pharo 8, classified by the deprecation scenarios presented in Section 5.4.5. 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 the Pharo Project. In Section 5.4.5, we claimed that *Method rename* and *Remove argument(-s)* scenarios can always be supported by rules. This can be seen in Table 5.2. 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 5.2, for 2 out of 43 non-rewriting deprecations that represent the complex replacement we introduced a transformation rule thus turning them into 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 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, six methods have already been supplied with the transformation rules (same as the ones we proposed), and one deprecation had been reverted (method that was marked as deprecated will not be removed after all). The other 16 transformation rules that we submitted as pull requests were merged into Pharo 9.[4]

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 callers. Those deprecations were introduced by 8 out of 15 developers who deprecated methods in Pharo 8. This demonstrates that the technology that we discuss in this chapter has already been adopted and used by the community.

- **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. The most interesting case is the lack of support for object-oriented rewriting (overriding abstract methods, introducing subclasses, adding methods, etc.). We will further discuss the challenging scenarion of library update in Chapter 8. The limitations of the language of code transformations can be addressed in the future work.

[4]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.

5.8 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. We distributed the survey through the mailing list of the Pharo open-source community (developers programming in Pharo); participation was optional and anonymous. We did not select the participants based on their practices or use of Deprewriter. From that perspective, they could either know or not about the existence of Deprewriter, and be either *client* or *library* developers.

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?

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 5.3 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 than the developers of client applications.

Table 5.3: Survey: *"What kind of software do you maintain?"*

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%)

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 5.4 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.

Table 5.4: Survey: *"How often do you migrate[5] your software to newer versions of its dependencies?"*

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

RQ4. How is Deprewriter used by client developers who are affected by deprecation rewriting? We asked several questions 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 5.5 shows that 28 (60%) participants of 46 acknowledged that the rewriting deprecations helped them, while 10 mentioned otherwise and 8 are uncertain.

Table 5.5: Survey: *"Do you know Pharo's support for automatic deprecation rewritings?"* and *"Did automatic deprecation rewritings help you in a migration?"*

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

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

[5]At the time of performing this survey, we did not yeat adopt the terminology that was presented in Chapter 2. In the survey questions, we referred to library update as "migration".

in the comment 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*[6]*. 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.

[6]In Pharo, a package is called "dirty" when it has been modified and has not been committed yet to a version control system such as Git.

- 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.

Table 5.6: Survey: *"Did you write your own automatic deprecation transformation rules to help migrate your users?"*

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

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 5.6). 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 our journal article [Ducasse 2022].
- To the question *'How easy was it to write a rewrite rule?'*, two participants found the expressions very easy, one easy, nine medium, and one difficult, 27 did not answer (see Table 5.7). The other three 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?'*, four developers indicated that they need support for rewriting on the level of class deprecations, four 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.), two developers requested conditional rewriting. Those answers demonstrate the limitations of Deprewriter that can be targeted in the future work (see Section 5.9).

Table 5.7: Survey: *"How easy it was to write a rewrite rule?"*

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

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 5.8).
- 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 5.8).
- To the question *What is your most frequent setup?* Developers use a mix of configurations. No clear choice jumped out.

Table 5.8: Survey: What are the configurations of Deprewriter preferred by the developers?

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

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 better documentation that would inform them on how to write better transformation rules.[7] 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 5.9).

5.9 Limitations and Discussion

5.9.1 Limitations in Presence of Extensibility

The approach presented in this chapter, 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 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 5.6 is only called by the method `template()` and should be overridden in subclasses.

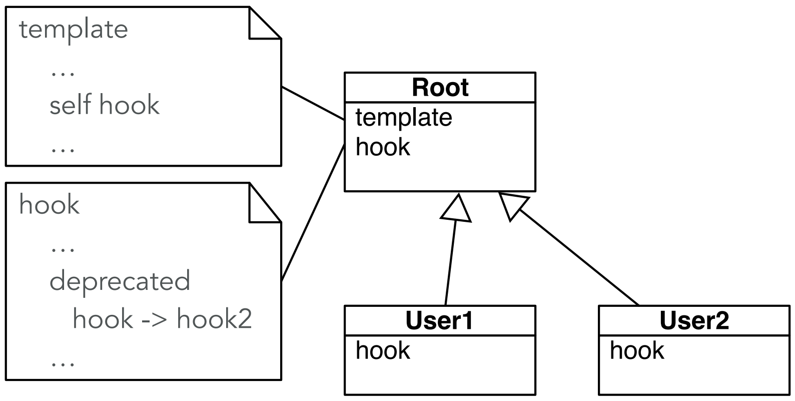


Figure 5.6: A simple hook and template situation where the default `hook` method is deprecated.

[7]At the time we performed the study presented in this chapter, no accurate documentation was available, only some unit tests.

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 5.6, 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 5.6, 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 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, as we will discuss in Section 8.2, in Commander,[8] a command design pattern framework, the following replacements were necessary: `basicName` should be renamed as `name`, and `name` as `dynamicName`. However, `basicName` should not be replaced with `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`.

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

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 5.2.2. 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.

5.9.2 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.

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 5.6, our approach is also implementable using AOP frameworks: it is feasible to instrument the deprecated methods through weaving.

5.9.3 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. The effects of Deprewriter 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.

5.10 Chapter Conclusion

In this chapter, we 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 library developer annotates the changed method with a transformation rule. During client program execution, Deprewriter uses the rule to find the call site of the deprecated method, then it rewrites and recompiles on the fly the method that called the deprecated method. Client developers can then assess the produced changes and version their updated 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 used it to rewrite their code, others wrote rewriting deprecations, and finally, some developers were not aware of Deprewriter. 28 out of 46 (60%) developers 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 and instance variables, as well as offer solutions for the limitations of the Deprewriter approach which we discussed in Section 5.9.

CHAPTER 6

DepMiner: Helping Library Developers to Deal with Breaking Changes

Contents

6.1 Introduction

In Chapter 5, we have discussed *Deprewriter* [Ducasse 2022] — a powerful deprecation engine provided by Pharo. It allows library developers to add transformation rules to their method deprecations specifying the replacements. When a deprecated method is invoked, Deprewriter identifies the call-site at run-time and uses the rule to update the client code without interrupting its execution [Roberts 1996, Roberts 1997, Renggli 2010b].

However, developers of real projects do not always follow good deprecation practices. They tend to introduce breaking changes to the APIs by renaming or removing certain classes, methods, or fields without deprecating them first [Xavier 2017a, Xavier 2017b, Brito 2020]. Also, as we have mentioned in Chapter 3, several large-scale studies of popular software projects have revealed that the proportion of deprecations that contain a helpful replacement message (in a form of comment, string, annotation, etc.) is only 66.7% for Java, 77.8% for C# [Brito 2018b], and 67% for JavaScript [Nascimento 2020].

In this chapter, we will be dealing with one aspect of the library update problem: automatically infering the replacements for deprecated methods, based on the commit history. As we have discussed in Chapter 3, multiple approaches have been proposed to support client developers in this task. Dig *et al.,* [Dig 2006a] proposed to detect refactorings between the two versions of the library based of the textual similarity of source code and the similarity of references. Schaffer et al. [Schäfer 2008], Dagenais *et al.,* [Dagenais 2008], and Hora et al. [Hora 2014] mined frequent method call replacements in the commit history of a library to learn how it adapted to its own changes. Pandita *et al.,* [Pandita 2015] and Alrubaye *et al.,* [Alrubaye 2019] used a similar technique to help client developers replace dependencies to one library with dependencies to another one. Teyton et al. [Teyton 2013] and Brito *et al.,* [Brito 2018b] recommend replacements by learning from client systems that have already updated their code.

We look at the problem from the perspective of library developers. We propose an approach and a tool called *DepMiner* to help them identify breaking changes before the release, understand when and by whom they were introduced, and find the potential replacements that could be suggested to the clients. We propose recommendations in the form of transformation rules that can be used by Pharo's Deprewriter. Inspired by the existing approaches that were proposed to support the client developers [Schäfer 2008, Dagenais 2008, Hora 2014], our approach is based on the frequent method call analysis. The main differences are: (a) we recommend replacements *before* the release which makes it impossible to rely on the clients that were already updated; (b) Pharo is a dynamically-typed language, which means that we can not rely on type information when analyzing method call replacements; (c) Pharo has no explicit method visibility (*i.e.,* `public` or `private` specifiers), which makes it hard to define the API. DepMiner can be extended to work with other increasingly popular dynamically-typed languages such as JavaScript, Python, Ruby, etc.

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

The contributions of the work presented in this chapter are the following:

1. We proposed an approach to recommend transformation rules for *Deprewriter* based on the commit history.

2. We implemented our approach as an open-source tool for Pharo.

3. We performed a first evaluation of our approach with developers of five open-source project.

The results presented in this chapter were published at the International Conference on Software Reuse (ICSR 2022) [Zaitsev 2022a].

The rest of this chapter is structured as follows. In Section 6.2, we discuss the problem of supporting library developers and the challenges that arise when dealing with this problem in Pharo. In Section 6.3, we describe our proposed approach and explain the underlying data mining algorithm. In Section 6.4, we evaluate our approach by comparing the generated transformation rules to the ones that are already present in the source code and by performing a developer study. Finally, in Section 6.5, we explain the limitations of our approach.

6.2 Why Support Library Developers?

In Chapter 5, we discovered that out of 367 valid deprecations in Pharo 8, 149 deprecations (41%) do not contain transformation rules. Out of those 149 non-transforming deprecations, 33 (22%) can have a simple transformation rule that can be generated automatically; 47 deprecations (32%) require developers with project expertise to provide extra information (additional argument, default value, etc.) and write a rule manually; the other 69 deprecations (46%) do not have a replacement and can not be expressed with a transformation rule. This indicates that developers do not always write transformation rules for their deprecations even when it is possible. Similar trends can be observed in other programming languages. As we mentioned earlier, according to large-scale studies of software systems, the proportion of deprecations that do not contain a helpful replacement message (in a form of comment, string, annotation, etc.) is 33% for Java, 22% for C# [Brito 2018b], and 33% for JavaScript [Nascimento 2020].

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

Challenge 1: Absence of method visibility. Languages like Java and C++ have public, private, and protected keywords that can help identify methods that are meant to be used by clients and can be considered as part of the API. However, in languages like Python or Pharo all methods are public [Schärli 2004]. Sometimes Python developers use underscores at the beginning of method names to mark them as “private” but it is more of a good practice than a strict requirement and this practice is not always followed. Although Pharo developers often adopt different practices to mark methods as private, none of those practices are universally adopted by the Pharo community.

Challenge 2: Absence of static type information. Pharo is a dynamically-typed programming language [Suzuki 1981, Milner 1978]. The absence of static type information complicates the task of identifying correct method mappings between the old and the new version because it is not easy to map method calls in the source code to the actual method implementations. We also do not know the argument types. This has an important implication that we can get a combinatorial explosion when analysing a sequence of messages. The research community has proposed type inference for dynamically-typed languages [Suzuki 1981, Furr 2009, Ren 2016, Spoon 2004, Pluquet 2009, Passerini 2014] or use dynamic type information collected by the Virtual Machine to get concrete types [Milojković 2016]. But such type inferencers often do not cover the full language [Suzuki 1981] or are not applicable to large code bases [Spoon 2004]. In this study, we do not perform type inference and consider that the type information is missing, which constitutes a challenge for the data mining algorithm.

6.3 DepMiner Approach

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

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

Identifying Methods of the Old and the New API. As we have discussed in Section 6.2 (Challenge 1), all methods in Pharo are public in the sense that clients can access them, however not all of those methods are meant to be used. To deal with this challenge, we define several categories of methods in Pharo that can be considered private: (a) *initialize methods* — they act like constructors in Pharo; (b) *unit test methods*, including setUp, tearDown, and methods of mock classes; (c) *example methods*; (d) *baseline methods* — define project structure and dependencies; (e) *help methods* — a form of documentation; (f) *methods in "private" protocols*[1]

[1]Methods in Pharo are organised into protocols — named categories that facilitate navigation through source code.

— any protocol that includes the word "private". We implemented heuristics to infer the visibility of methods in Pharo and released them in a public repository[2]. With that information, we define two sets of methods: API_{old} — "public" methods in the old version, and API_{new} — "public" methods in the new version.

Collecting Method Changes from the Commit History. Given the history of commits between the old version and the new version, we extract method changes from every commit. A *method change* describes how one specific method was changed by a given commit. For each method change, we parse the source code of a method before and after it was changed and extract a set of method calls from each version. As a difference between those two sets, we get the sets of deleted and added method calls for every method change. We remove all deleted method calls that were not part of API_{old} and all added method calls that are not part of API_{new}. Because Pharo is a dynamically-typed language, we do not know which implementation of a method will be executed (see Section 6.2, Challenge 2). As a result, many method calls in our dataset are false positives, because they call the method with the same name as the one in API_{old} or API_{new}, but in reality that method is called from a different library (*e.g.,* methods such as add() or asString() can be implemented by different classes). To deal with this problem and reduce the noise in our data, we choose a threshold K and remove all method changes that have more than K added or more than K deleted method calls (by experimenting with different values of K, for this project we selected $K = 3$). We also removed all calls to highly polymorphic methods such as = and printOn:. Finally, we removed all method changes for which either the set of deleted or the the set of added calls was empty.

Mining Frequent Method Call Replacements. After collecting the dataset of method changes from the commit history, we apply a data mining algorithm for market basket analysis to find all frequent subsets of method call replacements. This technique was inspired by the work of Schäfer *et al.,* [Schäfer 2008], Hora *et al.,* [Hora 2014], and Dagenais *et al.,* [Dagenais 2011] who proposed similar history-based approaches to support the clients of Java libraries. In terms of market basket analysis, each method change can be represented as a transaction or an itemset. To do that, we merge the sets of added and deleted calls in a method into a single set. For example, {deleted(isEmpty), deleted(not), deleted(add), added(new), added(isNotEmpty) }. By selecting a minimum support threshold min_{sup}, we can use a data mining algorithm such as A-Priori, Eclat, or FP-Growth to find all combinations of method calls that appear in different method changes at least min_{sup} times (frequent itemsets). In this study, we decided to use the A-Priori algorithm

[2] https://github.com/olekscode/VisibilityDeductor

because it is well-known and easy to implement. Then we construct association rules by putting all deleted method calls into the *antecedent* (left hand side) and all added method calls into the *consequent* (right hand side). We remove the rules with empty antecedent or empty consequent. For each association rule $I \rightarrow J$, we calculate its confidence — the probability that a set of deleted calls I appear jointly with added calls J and not with something else:

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

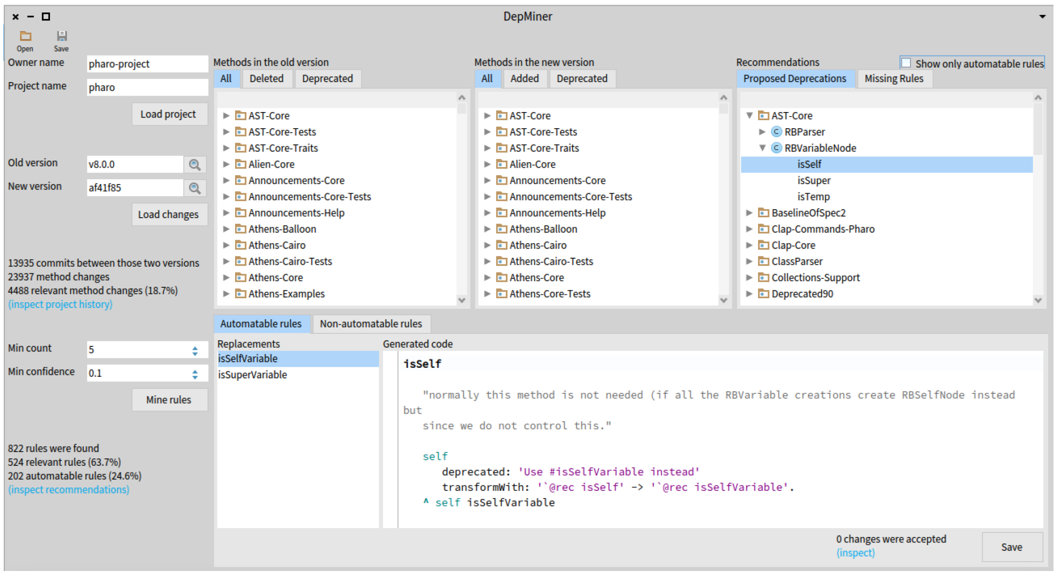
We select a confidence threshold min_{conf} and filter out all association rules that do not reach this threshold. The current implementation of Deprewriter supports only one-to-one (one antecedent, one consequent) and one-to-many rules (one antecedent, several consequents) — the ones that define the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

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

1. **Proposed deprecations** — we find all methods of the old API that were deleted without being deprecated (every method m such that $m \in API_{old}$ and $m \notin API_{new}$). If we can find at least one association rule in $Assoc$ that defines the replacement for a given method m, then we recommend to reintroduce m into the new version of a project with deprecation and a transformation rule if it can be generated.

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

Transformation rules of the form ‘@rec selector1: ‘@arg $\rightarrow$ ‘@rec selector2: ‘@arg can be generated automatically from the association rule such as {selector1:} $\rightarrow$ {selector2:} only if:



Owner name: pharo-project
Project name: pharo
Old version: v8.0.0
New version: af41f85

13935 commits between those two versions
23937 method changes
4488 relevant method changes (18.7%)
(inspect project history)

Min count: 5
Min confidence: 0.1

822 rules were found
524 relevant rules (63.7%)
202 automatable rules (24.6%)
(inspect recommendations)

Methods in the old version: AST-Core, AST-Core-Tests, AST-Core-Traits, Alien-Core, Announcements-Core, Announcements-Core-Tests, Announcements-Help, Athens-Balloon, Athens-Cairo, Athens-Cairo-Tests, Athens-Core, Athens-Examples

Methods in the new version: AST-Core, AST-Core-Tests, AST-Core-Traits, Alien-Core, Announcements-Core, Announcements-Core-Tests, Announcements-Help, Athens-Balloon, Athens-Cairo, Athens-Cairo-Tests, Athens-Core, Athens-Core-Tests

Recommendations: AST-Core (RBParser, RBVariableNode: isSelf, isSuper, isTemp), BaselineOfSpec2, Clap-Commands-Pharo, Clap-Core, ClassParser, Collections-Support, Deprecated90

Replacements: isSelfVariable, isSuperVariable

```
isSelf

   "normally this method is not needed (if all the RBVariable creations create RBSelfNode instead
but
   since we do not control this."

   self
      deprecated: 'Use #isSelfVariable instead'
      transformWith: '`@rec isSelf' -> '`@rec isSelfVariable'.
   ^ self isSelfVariable
```

0 changes were accepted
(inspect)

Figure 6.1: Screenshot of the DepMiner tool.

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

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

6.4 Evaluation

We have implemented our approach in a prototype tool for Pharo called *DepMiner*.[3] We have applied *DepMiner* to several open-source projects and asked core developers of those projects to review the recommendations produced by DepMiner.

6.4.1 Evaluation Setup

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

[3] https://github.com/olekscode/DepMiner

- **Pharo**[4] — a large and mature system with more than 150 contributors, containing the language core, the IDE, and various libraries.
- **Moose Core**[5] — Moose is a large platform for data and source code analysis. It consists of multiple repositories. In this study, we focus only on the core repository of Moose.
- **Famix**[6] — generic library that provides an abstract representation of source code in multiple programming languages. Famix is part of the Moose project.
- **Pillar**[7] — a markup syntax and tool-suite to generate documentation, books, websites and slides.
- **DataFrame**[8] — a specialized collection for data analysis that implements a rich API for querying and transforming datasets.

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

We classify the projects selected for this study into three types (see Table 6.1):

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

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

Table 6.1: Selected software projects

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

Two versions of each project. To mine the repetitive changes and propose deprecations, we must first select two versions of each project: the *new version* for which we will propose the deprecations and the *old version* to which we compare the new version of the project. All patterns will then be mined from the slice of the commit history between those two versions. Table 6.1 lists the two versions of each project that we have loaded as well as the number of commits between those two versions.

Mining frequent method call replacements. We used DepMiner to mine frequent method call replacements from the histories of those projects and recommend deprecations with transformation rules. In Table 6.2, we report the minimum support and minimum confidence thresholds that were used to initialize the A-Priori algorithm. The minimum support threshold for each project was selected experimentally. We started with a large support threshold = 15 (meaning that we are only interested in replacements that happened at least 15 times) and decreased it until the number of generated recommendations seemed sufficiently large. The confidence threshold was selected based on the number of method changes and the number of rules that *DepMiner* generated for a selected support value. For Pharo and Famix we can expect rules with confidence of at least 0.4. For other projects, we limit confidence to 0.1. In the last two columns of Table 6.2, we present the number of association rules (frequent method call replacements) that were found by *DepMiner* given the settings discussed above, and the number of rules that can automatically generate the transformation rules of the form '@rec deletedSelector: '@arg $\rightarrow$ '@rec addedSelector: '@arg (only one-to-one rules where deleted and added selectors have the same number of arguments).

6.4.2 Evaluation by Project Developers

We have performed a first developer study of our tool involving the core developers from each project listed in Section 6.4.1. We asked 4 developers with different areas of expertise to validate the recommendations generated for Pharo and one developer for each of the other 4 projects (two developers had expertise in two

Table 6.2: Association rules mined from the commit history

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

projects each so in total, our study involved 6 developers).

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

Proposed deprecations. Table 6.3 reports the numbers of deprecations that were recommended to developers for each project (column *Recommended*), the number of those recommendations that were accepted (column *Accepted*), and the number of those accepted recommendations that contain an automatically generated transformation rule (column *Transforming*). Each recommended deprecation is a method that was deleted from the project without being deprecated first and which we propose to re-introduce with the recommended replacement.

Table 6.3: Number of recommended deprecations accepted by developers

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

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

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

Missing rules. The other type of recommendations that we showed to developers were transformation rules for existing non-transforming deprecations. Table 6.4 reports the number of existing deprecations that are missing a transformation rule, the number of recommendations that DepMiner managed to generate for those deprecations, and finally the number of recommendations that were accepted by developers.

Table 6.4: Number of missing rules accepted by developers

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

Deprecations that were introduced without a transformation rule represent either complicated cases for which the transformation rule can not be provided (e.g. method was deleted without replacement) or simple cases for which developers forgot to write a rule. As we could see in Chapter 5, for 22% of non-transforming deprecations the transformation rule could be generated automatically (assuming that we know the correct replacement), the other 78% of non-transforming deprecations require a complex rule that must be written manually. *DepMiner* proposed 6 transformation rules for existing non-transforming deprecations in Pharo (2 of which were accepted) as well as 2 transformation rules for Famix (both were accepted).

Depminer was able to find only 2 transformation rules for 189 non-transforming deprecations in Pharo 9. By contrast, as we reported in Chapter 5, human developers found 33 rules out of 149 non-transforming deprecations in Pharo 8. In the next chapter, we will propose an improvement to this approach.

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

6.5 Limitations of Our Approach

Unused/untested methods. Our approach is based on library's usage of its own API. This means that we can not infer anything for methods that are not used by the library itself but only intended for clients. Test cases play the role of clients of the library's API, so for the methods that are well tested, we can have enough input to identify the correct replacement for them. But if a method is not used by the library and not covered by test, then its deletion or renaming will not be reflected anywhere else in the source code. *To overcome this limitation, one can combine the DepMiner approach for mining frequent method call replacements with the detection of refactorings* (see Chapter 7).

Searching the entire commit history. As all previous studies of library update through mining frequent method call replacements that we discussed in Chapter 3, DepMiner approach mines the entire commit history to find replacements for the method that was removed. *One can overcome this limitation by identifying the commit in which the given method was removed and then only searches for method call replacements in the smaller subset of neighboring commits* (see Chapter 7).

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

Reflective operations. Modern programming languages offer reflective operations [Richards 2011, Callau 2011]. They allow developers to invoke methods programmatically and create generic and powerful tools. However, since some methods can be invoked reflectively for example passing the name of the method to be invoked in a variable, when a different argument is passed to a reflective call, our tool cannot identify such change. Most static analysers ignore such case [Bodden 2011]. *Overcoming this limitation is beyond the scope of this thesis.*

6.6 Chapter Conclusion

Method deprecation is a powerful technique for supporting the evolution of software libraries and informing client developers about the upcoming breaking changes to the API. In this chapter, we presented an approach to mine the frequent method call replacements from the commit history of a library and use them to recommend method deprecations and transformation rules. We implemented our approach for the Pharo IDE in a tool called *DepMiner*. We applied our tool to five open-source projects and asked 6 core developers from those projects to accept or reject the recommended changes. In total, 134 proposed deprecations were accepted by developers as well as 4 transformation rules for the existing deprecations. 61 new deprecations and 2 transformations rules for existing deprecations were integrated into the Pharo project.

CHAPTER 7

First Steps Towards a Holistic Approach

Contents

7.1 Introduction

Library developers can reduce the negative effect of breaking changes with deprecations [Robbes 2012a, Sawant 2016] and clear documentation that explains all API inconsistencies and suggests replacements. However, changes are not always documented at the same time when they are introduced. Different developers at different stages of library evolution can rename methods, move them to a different class, merge or split classes, reorganize packages, etc. Those changes affect the API but they are often documented at the end of the library release cycle by the person who cleans up the code and writes the release notes. At that point, it can be hard to remember all the changes, understand them, and find replacements. Especially in large open-source libraries that change very fast and have many contributors.

As we have discussed in Chapter 3, various approaches have been proposed to help *client developers* identify the replacements for public methods that are no longer present in the new version of a software library [Kim 2007, Xing 2007, Schäfer 2008, Wu 2010, Dagenais 2011, Meng 2012, Teyton 2013, Hora 2014]. In our work, we focus on how we can help the *library developers* to support their clients and reduce the negative effect of breaking changes *before* releasing the new version of a library. The DepMiner tool that we presented in Chapter 6, recommended transformation rules and generated method deprecations that library developers could insert into their code.

Our approach was based on the assumption that library developers are always willing to deprecate methods instead of directly breaking backward compatibility. However, as we have discovered while conducting the DepMiner experiment and discussing with library developers from different open-source projects of Pharo ecosystem, this is not always the case. There are situations when a transformation rule exists but library developers prefer not to deprecate the method but remove it directly. In Section 6.5, we have also identified several shortcomings of the DepMiner approach, including: (1) inability of DepMiner to find replacements for methods that are not used internally and not tested; (2) the inefficiency of searching the entire commit history for method call replacements that were caused by the removal of a specific method.

In this chapter, we discuss the first steps towards developing a more general approach to help library developers support their clients. An approach that takes into account different types of actions that library developers can take depending on the specific scenario. We also propose the ways to overcome the two limitations of DepMiner that were mentioned above. Implementation and validation of the holistic approach will be the focus of our future work.

The main contributions presented in this chapter are:

1. Identification of the six responses that library developers can have to breaking changes.

2. Discussion of the possible adaptations of the DepMiner approach that would overcome some of its limitations and improve its efficiency in detecting replacements for removed methods.

The rest of this chapter is structured as follows. In Section 7.2, we discuss the different strategies of library developers to reduce the negative effect of breaking changes and two families of approaches that have been proposed to support developers in the task of library update. In Section 7.3, we discuss the shortcomings of existing approaches. In Section 7.4, we present our first contribution — the breakdown of different scenarios and different actions that can be taken by library developers to reduce the effect of breaking changes. In Section 7.5, we present our second contribution — the adaptations of the DepMiner approach to mine replacements for the API-breaking methods. In Section 7.6, we describe the holistic approach to help library developers deal with breaking changes.

7.2 Different Types of Support

In this section, we discuss the strategies that library developers use to reduce the negative effect of breaking changes on clients. Then we review several approaches that were used to help client developers update their software to the new version of a library that was released with breaking changes.

7.2.1 Strategies of Library Developers to Reduce the Negative Effect of Breaking Changes

Based on the analysis of Java libraries, Xavier *et al.,* [Xavier 2017a] report that on median 14.78% of all API changes break compatibility with clients. As indicated by the study of the StackOverflow questions reported by Brito *et al.,* [Brito 2019], breaking changes have an important impact on the API clients as 45% of the analysed questions related to breaking changes are from client developers trying to overcome their negative effects. The process of updating the client application to the new version of a library can be error-prone and time consuming. To help client developers in this task, library developers use several strategies to reduce the negative impact of breaking changes:

Strategy 1: Deprecations. Instead of removing a method in release n, it is marked as *deprecated* (*"to be removed"*) and only actually removed in a future release $n+k$. Client systems that use the deprecated method receive a deprecation warning which gives developers time to update their code. Besides notifying that the method should not be used, library developers can also inform clients of the alternatives in the new API that can be used instead (we refer to them as "replacements") using one or multiple practices:

- *Replacement messages.* Developers can supply deprecations with messages that suggest a replacement for an obsolete item (*e.g.,* a comment or a warning message *"Method x() is deprecated, use y() instead"*).
- *Annotations with references.* Some programming languages provide annotations that can be added to deprecations and reference the replacement in source code (*e.g.,* in Java, the @Deprecated annotation or @deprecated Javadoc tag, combined with @link or @see tags).
- *Transformation rules.* The Deprewriter approach, that we discussed in Chapter 5, allows library developers to add transformation rules to method deprecations that will be used to automatically fix the client code.

Adoption: In practice, library developers often decide to remove functionality without deprecation. Hora *et al.,* [Hora 2015] report that 59 out of 118 API changes that they analysed are a missed deprecation opportunity. According to Brito *et al.,* [Brito 2019], the most common reason for that is the fear of increasing the maintenance effort (*e.g.,* developers believe that the breaking change will not affect many clients and deprecations will only add complexity and maintenance issues). Several large-scale studies of popular software projects have revealed that the proportion of deprecations that do not contain a helpful replacement message (in the form of a comment, string, annotation, etc.) is 33% for Java, 22% for C# [Brito 2018b], and 33% for JavaScript [Nascimento 2020].

Strategy 2. Documenting the breaking changes. It is a good practice to document the breaking change in the release notes, explain why the functionality was removed, and suggest a replacement or workaround that can be applied by client developers to fix their code.

Adoption: In our survey of library evolution (see Chapter 4), library developers identified documentation as the most common practice that they use to support client developers. Client developers in our survey identified documentation as the most important factor that makes library update easy and the absence of documentation as the main factor that makes library update hard.

7.2.2 Automated Tools to Support Developers

As we have discussed in Chapter 3, in recent years many approaches have been proposed to support client developers in the process of library update by analysing source code and mining the commit history. In this thesis, we claim that similar techniques can be proposed to help library developers better support their clients. We take a closer look at two families of approaches:

1. *Detecting the refactorings.* According to Dig *et al.,* [Dig 2006b], more than 80% of breaking changes are refactorings. Many of those refactorings, if detected correctly, can suggest a specific replacement for the removed functionality on the client side. Dig *et al.,* [Dig 2006a] proposed an algorithm to detect refactorings between the two versions of the library based of the textual similarity of source code and the similarity of references.

2. *Analysing frequent method call replacements.* There are several approaches to deal with the remaining 20% of breaking changes that are more complex than those caused by the common refactoring operations. Dagenais *et al.,* [Dagenais 2011] and Hora *et al.,* [Hora 2014] mine the commit history

to analyse how the self-usage of library changed when a method was removed. Other approaches extract information from already updated client systems [Schäfer 2008, Teyton 2013].

The DepMiner approach, that we presented in Chapter 6, is based only on the analysis of frequent method call replacements. This leads to the limitations of our approach that were mentioned above and will be discussed in more detail in the following section. In this chapter, we propose to improve the DepMiner approach by combining the analysis of frequent method call replacements with the detection of refactorings. Such a combination was inpired by the work of Dagenais *et al.,* [Dagenais 2011].

7.3 The Shortcomings of Existing Approaches

There are several aspects that are often overlooked by the studies that focus on recommending replacements. In this section, we first mention two shortcomings of DepMiner that will be addressed in the rest of this chapter. Then we discuss the shortcomings of the rule-based approaches to support developers in the problem of library update.

Limitations of the DepMiner approach. The approach that we presented in Chapter 6, suffers from several limitations that we discussed in Section 6.5. Some of those limitations, such as *reflective operations* or *the order of method calls* are outside the scope of this thesis. The other two limitations will be addressed in this chapter:

1. *Unused/untested method.* The idea of mining frequent method call replacements is based on the assumption that after renaming or removing a method, the library reacts to this change by updating all references to that method in its own code. While this assumption holds for the methods that are used internally or covered by unit tests, there are also methods that are only meant for clients and never used by the library itself. The replacements for such methods can not be identified by the analysis of frequent method call replacement. *To overcome this limitation, we propose to extend the DepMiner approach with the detection of refactoring operations.*

2. *Searching the entire commit history.* Like many similar approaches, DepMiner searches for frequent method call replacements through the entire commit history. However, if we assume that those replacements are caused by a specific breaking change that was introduced in commit C, then it is

more likely that the effect of this change will occur in the neighbouring commits (the ones that come just before or after commit C). *We propose to improve the precision of DepMiner by searching a smaller local subset of commits.*

Shortcomings of the rule-based studies. Most studies that we discussed in Chapter 3, as well as the DepMiner approach, are based on mining rules that can be applied to client code and replace one or multiple method calls with other method calls. Such rule-based approaches suffer from the following shortcomings:

1. *Not every breaking change has a replacement.* Although the majority of changes in software libraries are due to refactorings [Dig 2006b], there are situations when libraries stop performing certain operations either because they are delegated to other external libraries or because the functional responsibilities of the library are reduced or changed. For example, DataFrame v1.0 provided methods for visualizing data frames: barplot, scatterplotMatrix, etc. In version v2.0, developers decided to drop support for visualizations. And now, even though the same visualizations can be created using an external library, there are no replacements for the removed methods in the DataFrame project.

2. *Developers may not want to deprecate or document.* There are situations when library developers do not want to deprecate a public method that was removed. We witnessed multiple such cases while performing the DepMiner experiment described in Chapter 6: even though the transformation rule proposed by our tool was correct, developers did not want to introduce the deprecation. They believed that the removed method is not supposed to be used by clients and that adding a deprecation would only increase the complexity of the code.

3. *Some replacements can be hard to express with rules.* Although many breaking changes can be expressed in the form of transformation rules such as the ones described in Chapter 5, there are also changes during library evolution that require more complex changes to the client code. For example, if a method is moved to a different class, then not only the method calls must be replaced, but also the code that instantiates the receiver. Creating an instance of a class is not always an easy task. If an extra argument was added to the method call, client developers must provide a value for that argument. There are also breaking changes that are not caused by refactorings but by a complex combination of architectural and functional changes in the source code of the library. In Chapter 8, we will report several challenging cases of library update and discuss them in more detail.

The shortcomings listed above make it difficult to apply rule-based tools for library update to certain scenarios of library update. In those cases, library developers can rely on different strategies to support their clients, for example, by documenting those changes, writing examples, and tutorials to help clients to update. They can still benefit from the automated tools that extract knowledge from the commit history, but the recommendations produced by those tools should be more generic than method-to-method replacement rules.

7.4 Understanding the Needs of Library Developers

In the work that was described in Chapter 6, we proposed an approach to generate deprecations with transformation rules. We conducted a study involving library developers of five open source projects from the Pharo ecosystem. We have noticed that multiple recommendations generated by our tool, although correct, were rejected by library developers because they did not want to deprecate the given method. Thus, by observing the behaviour of developers in this study and discussing with them the reasons for accepting or rejecting certain recommendations, we have identified 6 possible scenarios that suggest the possible responses of library developers to breaking changes related to public methods.

We present those scenarios in Table 7.1. They are based on two factors:

1. *Does the library developer want to deprecate a method?* In some cases, developers can mark the method as deprecated before removing it. This deprecated method may then be supplied with a comment or a transformation rule that would help clients to update their code. In other cases, library developers prefer not to deprecate the affected method. When this happens, the support must be placed elsewhere, for example in project documentation or release notes.

2. *What kind of replacement exists for the affected method?* Some public methods that are affected by library evolution can be replaced with other methods in the new version (for example, if a method was renamed or split in two). It also happens that methods are removed without replacement (for example, if the functionality is dropped). Some replacements can be expressed with rules that can be used to automatically update client code — we call those replacements *automatable*. Other replacements are more complex and can not be expressed with rules — we call them *non-automatable* replacements.

Depending on the scenario, we suggest one of the following six responses to breaking changes:

Table 7.1: Different scenarios that occur when dealing with a breaking change. The columns differentiate cases when a removed method has or does not have a replacement as well as the automatability of the replacement. The rows describe the willingness of the library developer to deprecate a method.

	Method has an automatable replacement	**Method has a non-automatable replacement**	**Method has no replacement**
Developer wants to deprecate	Deprecation with a transformation rule	Deprecation with a replacement message	Deprecation without a replacement message
Developer does not want to deprecate	Library update script with transformation rules	Documentation with replacement instructions	Documentation: list of breaking changes

- *Add a deprecation with transformation rule.* If developers agree to deprecate the public method that needs to be removed, and a replacement on the client code is automatable (can be expressed as a rule), then the most effective way to support the clients would be to write a transforming deprecation that would automatically rewrite client code when a deprecation warning is signalled (see Chapter 5).

- *Provide a library update script with transformation rules.* If an automatable replacement exists but library developers prefer to remove the method directly without deprecating it first, then the transformation rule can be sent to clients in the form of a library update script — a piece of code that could be executed by clients and update their system using the rules.

- *Deprecate the method with a replacement message.* If a replacement for a method exists but it is not automatable, library developers can add a deprecation message in the form of a comment or a string, suggesting the replacement to the clients and referencing relevant examples.

- *Add replacement instructions to documentation.* If library developers decide not to deprecate, they can place the instructions on how to update into the documentation of a library or its release notes.

- *Deprecate the method without a replacement message.* In case there is no replacement for the obsolete method, library developers can deprecate it and explain in the comment or deprecation string the reasons why it was removed.

- *Add the list of breaking changes to the documentation.* In case the method was removed without a replacement and library developers refuse to deprecate it, they can still help their clients by clearly listing this breaking change in the documentation.

7.5 Overcoming the Limitations of DepMiner

In this section, we propose a way to overcome two limitations of DepMiner that were discussed in Section 7.3.

7.5.1 Finding a Local Subset of Relevant Commits

The first limitation of DepMiner is that it searches the entire commit history to find the frequent method call replacements caused by a breaking change, instead of searching a local subset of commits in which this change happened. This is also the limitation of the related approaches that were discussed in Chapter 3 [Schäfer 2008, Teyton 2013, Hora 2014].

To overcome this limitation, we propose to conduct a more local search by considering only the commit c_i in which the method was removed, as well as k previous and k next commits (see Figure 7.1).

$$C = \{c_{i-k}, \dots, c_i, \dots, c_{i+k}\}$$

Parameter k, which defines the size of a subset of commits, can be adjusted by the library developers to control the number of false positive and false negative recommendations for every given project.

We hypothesise that, when a breaking change takes place, it is performed either in a single commit or in a group of consecutive commits. This allows us to reduce the number of potential candidates and the search complexity. If a method was removed by a refactoring that was performed using a refactoring tool provided by the IDE, the replacement will typically be added at the same time as the method is removed. All references to the old method should also be replaced by the refactoring tool at the same time. It is also possible that the method is removed and committed, then the replacement is introduced and the references are updated in the following commits. Finally, developers may be following a good practice of introducing the replacement first before removing the old method. To capture situations like that, we also search for candidate replacements in k previous commits.

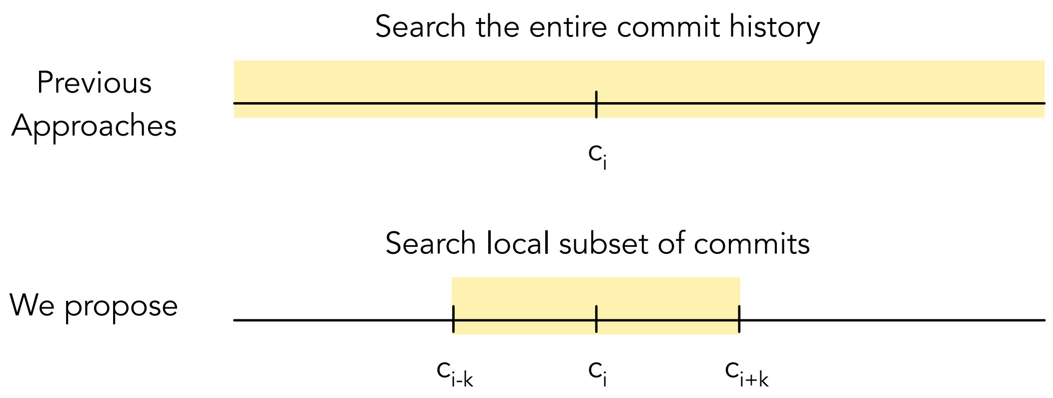

Search the entire commit history
Previous Approaches
c_i
Search local subset of commits
We propose
c_i-k
c_i
c_i+k

Figure 7.1: Mining the local subset of commits around the commit c_i that introduced a breaking change.

7.5.2 Detecting Refactorings that Removed a Method

The second limitation of DepMiner is its inability to recommend replacements for methods that are not used internally by the library and not covered by unit tests. When those methods are removed, there are no method calls in the library that need to be updated, and therefore the mining of frequent method call replacements will produce no results. To overcome this challenge, we propose to combine the DepMiner approach with the technique to detect refactoring operations in the commit history that is similar to the one proposed by Dig *et al.*, [Dig 2006a]. This combination is inspired by the work of Dagenais *et al.*, [Dagenais 2011].

Most breaking changes are caused by refactoring operations such as renaming, adding an argument, moving method to a different class, etc. [Dig 2006b]. Those changes can be detected in the commit history. For example, if a method a() was renamed to b(), there will be a commit that removed method a() and added method b() with same (or almost same) source code.

For every API-breaking method $m()$, one must find the commit c_i that removed it and a surrounding subset of commits C (see Section 7.5.1), then do the following:

1. *Find candidate replacements.* Find all methods that were added by a commit in C. Compute textual similarity between the removed method $m()$ and each of the added methods. Select added methods that pass the similarity threshold T as candidate replacements for $m()$.

2. *Detect the possible refactorings.* For every candidate replacement $m'()$, find a refactoring R that explains the $m \to m'$ replacement. In Table 7.2, we present the list of considered refactorings.

In Table 7.2, we present the list of refactoring operations that we propose to consider together with the conditions that can be used to detect them. Conditions

Table 7.2: Refactoring operations that we consider in our analysis and the conditions to detect them. In the examples, method $A.m()$ is replaced with method $B.m'()$. The first two operations are *Name* refactorings, because they affect the name of a method or its signature, and the last three are *Location* refactorings, because they move the method to a different class or package.

Category	Refactoring	Condition	Automatable?
Name	Rename Method (RM)	m and m' have different names but same return type, number of arguments, and argument types	Yes
	Change Signature (CS)	m and m' have different signatures (return types, number of arguments, or type of arguments)	No
Location	Rename Class (RC)	if method $A.m()$ was removed by commit c_i and $B.m'()$ was added by commit c_j, then c_i must remove the class A and c_j must add the class B[1]	Yes
	Push Down (PD)	B is the subclass of A	No
	Move Method (MM)	A and B are not connected by class hierarchy	No

must be checked in the same order in which they appear in the table. Those are the same refactorings that were analysed by Dig *et al.*, [Dig 2006a] except the Pull Up refactoring which does not break backward compatibility and therefore should not be considered in the context of library update. In this case, for a given API-breaking method $A.m()$ and every candidate replacement method $B.m'()$, we try to find the refactoring(-s) that could have caused the $A.m \to B.m'$ replacement.

The first two refactorings belong to the *Name* category because they affect the name of a method or its signature. The other three refactorings belong to the *Location* category because they move a method to a different class or package. Two refactorings from different categories can be combined in the same commit. For example, a method can be moved to a different class and then renamed (MM+RM) or pulled down into a subclass and have one of its arguments removed (PD+CS). On the other hand, two refactorings from the same category cannot be detected in a single commit because then they are indistinguishable from a single refactoring. For example, if method a is renamed to b and b is renamed to c in the same commit, then we will only see the $a \to c$ renaming. If method a is moved from class A

to class B and then pushed down into a subclass C in the same commit, then we will only see the Move Method($A.a \rightarrow C.a$) refactoring. So in addition to the 5 refactoring operations listed above, we also consider the 6 valid combinations of those refactorings: RC+RM, RC+CS, PD+RM, PD+CS, MM+RM, and MM+CS.

Our approach is similar to the one proposed by Dig *et al.,* [Dig 2006a]. However, unlike authors who compare two versions of source code, we propose to analyse more granular changes by comparing every pair of subsequent commits in the history of a project. This would allow not only to propose the replacement for a removed method, but also to explain why and when each change was made, providing exact references from the commit history that can help the library developers to make an informed decision.

7.6 Holistic Approach to Deal with Breaking Changes

In this Section, we propose a roadmap for a holistic approach to support library developers dealing with breaking changes (see Figure 7.2). It is based on the DepMiner modified with the two improvements proposed in Section 7.5 and used to guide library developers respond to breaking changes in one of the six ways that we proposed in Section 7.4.

Below are the steps of our approach:

1. *Collect the data.* Collect the history of changes between those versions and the list of public methods in each version.

2. *Detect breaking changes.* Compare two versions of the API to find the methods that were part of the old version but are no longer present in the new version of the library.

For every missing method $A.m()$:

3. *Find replacements.*

   – In the history of changes, find the last commit c_i that removed method $A.m()$. Select c_i together with k previous and k next commits into a subset of relevant commits: $C = \{c_{i-k}, \dots, c_i, \dots, c_{i+k}\}$.

   – Analyse C to find possible replacements for $A.m()$ using the combination of two strategies: detecting the refactorings that removed $A.m()$ and mining frequent method call replacements to understand how the library itself reacted to the removal of $A.m()$.

   – For every replacement method $B.m'()$, check if it is part of the new API. If yes — recommend $B.m'()$ as a valid replacement for $A.m()$. If

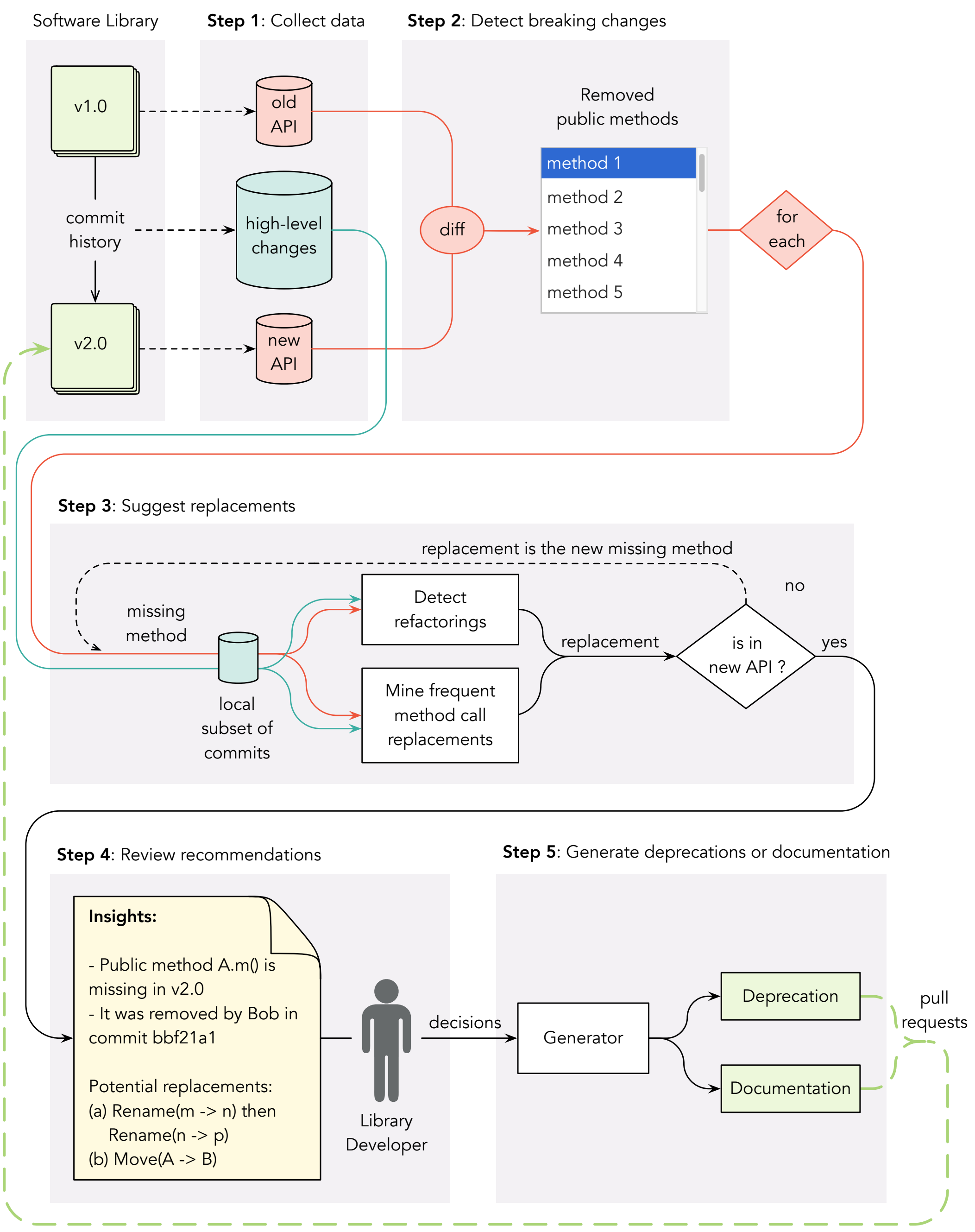


Figure 7.2: Holistic approach to help library developers identify breaking changes before the release and reduce their negative effect on client systems.

no — add $B.m'()$ to the sequence of replacements and repeat this step, now considering $B.m'()$ as a missing method.

4. *Review recommendations.* Present the breaking change and the proposed replacements to library developers. Ask them to validate the recommendations and select the appropriate response to handle the breaking change, based on Table 7.1.

5. *Generate deprecation or documentation.* If developer wants to deprecate the method, generate the deprecation, otherwise add this method to the list of breaking changes in the release notes. If replacement exists, generate the replacement message. If replacement is automatable, generate the transformation rule.

The development and validation of such an approach will be the focus of the future work.

7.7 Chapter Conclusion

In this chapter, we discussed the first steps towards developing a holistic approach to help library developers support their clients. We have discussed the needs of library developers and identified six responses that they can have to breaking changes in their systems. We have also proposed the adaptations to the DepMiner approach that would allow to overcome some of its limitations. Finally, we have proposed the first sketch of a holistic approach to support library developers. Implementation and validation of such an approach will be the focus of our future work.

CHAPTER 8

Case Studies of Challenging Library Update Problems

Contents

8.1 Introduction

In addition to the survey that we presented in the previous chapter, we have discussed with the developers from Pharo community to collect the case studies of challenging library update scenario. Those are the situations that are hard to handle with current deprecation mechanisms or hard to automate with rules that could be applied to client code (such as the Deprewriter rules that we presented in Chapter 5).

What do we consider a "challenging" scenario? Most studies of library update that were discussed in Chapter 3, as well as our studies that will be presented in Chapters 6 and 7, focus on the simple case of method-to-method mapping. In this model, two versions of a library can be represented as a collection of public methods and the main question of the library update problem is how to find the mapping between a public method in the old version and public methods in the new version. The basic assumption is that by replacing one method call with a new method call, we can overcome the negative effect of a breaking change and fix the client code. Although, in many cases, this is true, there are also scenario that go beyond method call replacements. In this chapter, we try to document some of those scenario for the purpose of future research.

How did we collect the scenario? We started by documenting several cases of library update problem that were encountered by our colleagues. Then we wrote an email to the mailing list of Pharo open-source community, asking developers to report their experience with library update. We also added an open question about challenging library update scenario to the survey that we discussed in Chapter 4.

8.2 Case 1: Reassigning the Existing Name

Description. This situation occurs when the name that is already used for one software artifact has to be reassigned to another one. For example, *"Rename a to b"* and then in parallel *"Rename b to c"* (see Figure 8.1). In this case, we assume that $a \neq c$. The more challenging scenario when $a = c$ and thus two names are being swapped will be discussed in Section 8.3.

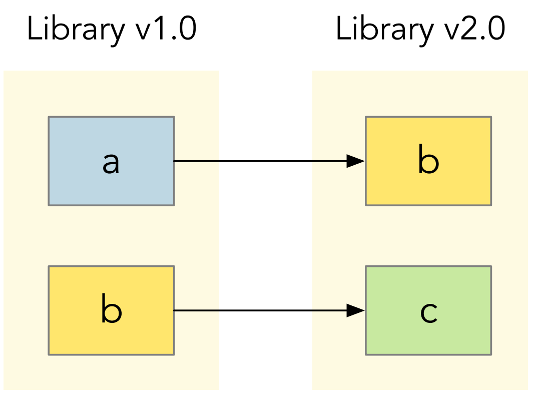


Figure 8.1: Reassigning the existing name b to a different software artifact. In this example, a is renamed to b and in parallel b is renamed to c.

Example. The example of such renaming was found in Commander2[1] — a second iteration of a Pharo library that models application actions as first class objects following the Command design pattern. In the old version of this library, class CmCommand had two methods for getting the name of command:

- basicName — accessor for the name of command stored as a string. For example, the command 'Switch' can be used to turn on the lights if they are off and turn off the lights if they are on.

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

- name — the hook that is called to generate the name dynamically. By overriding this method, developers can extend the command name with contextual information. The 'Switch' command from the previous example can have a dynamic name 'Switch(on)' that specifies if the lights will be turned on or off.

On August 14, 2019, method name was renamed to dynamicName and method basicName was renamed to name. This was a breaking change that could affect many clients who used the above-mentioned methods.

Why is it challenging? Below are the reasons why the case of reassigning the existing name can be challenging in the contect of library update, both for developers and for the automated tools:

- Most automated tools that are proposed in the related studies, apply transformation rules in an arbitrary order. This situation presents a scenario when the order of applying rewriting rules matters. All references to b must be replaced with c before a is replaced with b.
- In the context of dynamic rewriting, this is especially challenging because we do not have access to all references at once. For example, consider a situation when at runtime a tool encounters a reference to b. Is it the old reference that should be replaced with c? Or was this piece of code already rewritten by the rule a $\rightarrow$ b?
- It is also hard because deprecation warning will never be signalled for b.

How can it be fixed?

- *Static rewriting.* If it is possible to access all references to a and b in the client system, they can be rewritten statically. In this case, the tool can first apply the b $\rightarrow$ c rule thus rewriting all references to b, and then apply the a $\rightarrow$ b rule.
- *Logging rewritten locations.* If it is not possible to rewrite all calls statically, then the tool can log all references that were already rewritten.

8.3 Case 2: Circular Renaming

Description. This is a special case of reassigning the existing name when the names of two software artifacts are being swapped: *"Rename a to b"* and *"Rename b to a"* (see Figure 8.2). Although this case is rarer than the previous one, it is particularly challenging for automated tools and therefore deserves to be discussed.

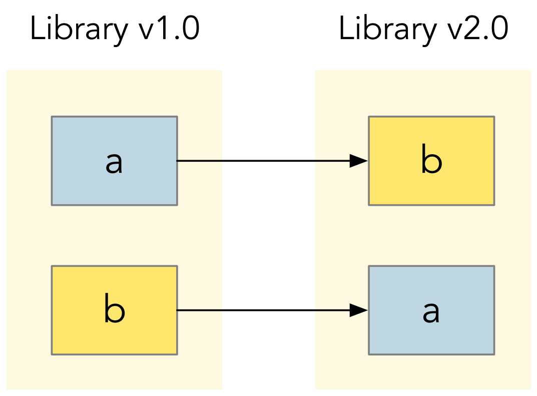


Figure 8.2: Circular renaming is a particularly challenging case when the names of two software artifacts are being swapped: *a* is renamed to *b* and at the same time *b* is renamed to *a*.

Why is it challenging? Most refactoring tools, as well as manual rewriting performed by developes, apply transformations one rule at a time. In case of circular renaming, there are two rules that cancel each other. They can be applied over and over to the same place in source code, first replacing a with b, then replacing that b with a, and so on. The tool must remember which locations have already been rewritten, or apply all rules simultaneously.

How can it be fixed?

- If it is possible to access all references to a and b in the project they can be rewritten simultaneously.
- Otherwise, the tool can introduce a temporary name temp and then perform renaming in three steps: (1) a $\rightarrow$ temp, (2) b $\rightarrow$ a, (3) temp $\rightarrow$ b. However, this also requires that *all* references to a are replaced with temp before the first refence to b is replaced with a.
- If it is not possible to access all references at once (for example, in the context of dynamically-typed languages), the tool can keep a log of locations in source code that have already been rewritten.

8.4 Case 3: Modifying Abstract Hooks

Description. Most approaches to library update view the library as a collection of methods that are invoked by clients. Hovewer, some libraries also provide hooks — abstract methods that should be overriden by clients, as in the *Template Method* design pattern [Gamma 1995]. In this case, it is not the client who invokes the library, but the library that calls a method provided by the client system. This situation is related to the *Reuse Contract* [Steyaert 1996]. We illustrate this scenario in Figure 8.3.

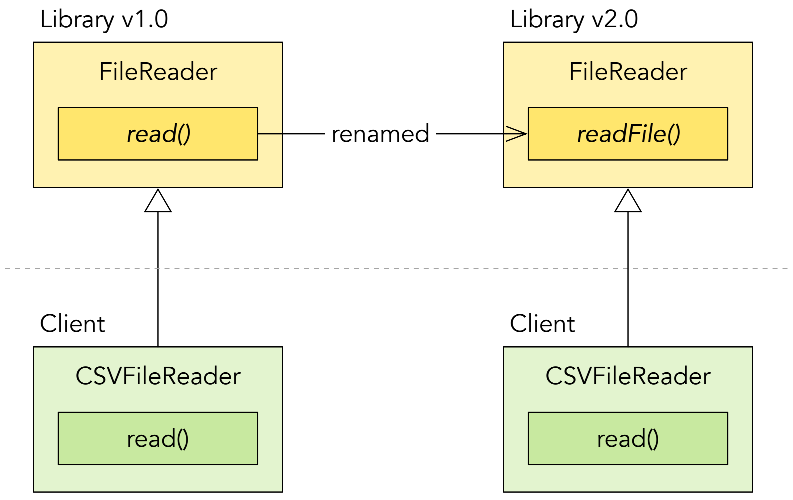


Figure 8.3: In v1.0, library used to provide an abstract hook read() that was meant to be implemented by clients. In v2.0, this hook was renamed to readFile(). But the client system still provides the implementation for the old method read(). Now the client implementation will never be called.

Example. In older versions of Pharo, the OSPlatform class provided an abstract method ffiModuleName: that was implemented by concrete classes MacOSPlatform, UnixPlatform, and WinPlatform to specify the name of a dynamic library used for FFI calls on the given platform. For example, 'libgit2.dll', 'libgit2.so', and 'libgit2.dylib' are used to make calls to LGit library on Win32, Unix, and MacOS platform respectively, and 'libc.so.6' is used to make calls to the LibC library on Unix platform. Each one of those implementations of ffiModuleName was a template method [Alpert 1998] that called a corresponding platform-specific method — macModuleName, unixModuleName, or win32ModuleName — that should be implemented by all subclasses of the FFILibrary. In Figure 8.4, yellow classes represent library code while the classes of the client system are green: LibC and LGitLibrary are two implementations of FFILibrary that are provided by client systems.

In Pharo 8, all those ...ModuleName methods were renamed to ...LibraryName. For example, OSPlatform.ffiModuleName: was renamed to OSPlatform.ffiLibraryName:, FFILibrary.macModuleName was renamed to FFILibrary.macLibraryName, etc.

As a result, template methods of MacPlatform, UnixPlatform, and WinPlatform call the new methods macLibraryName, unixLibraryName, and winLibraryName instead of the old ones. The clients are expected to rename ...ModuleName to ...LibraryName in

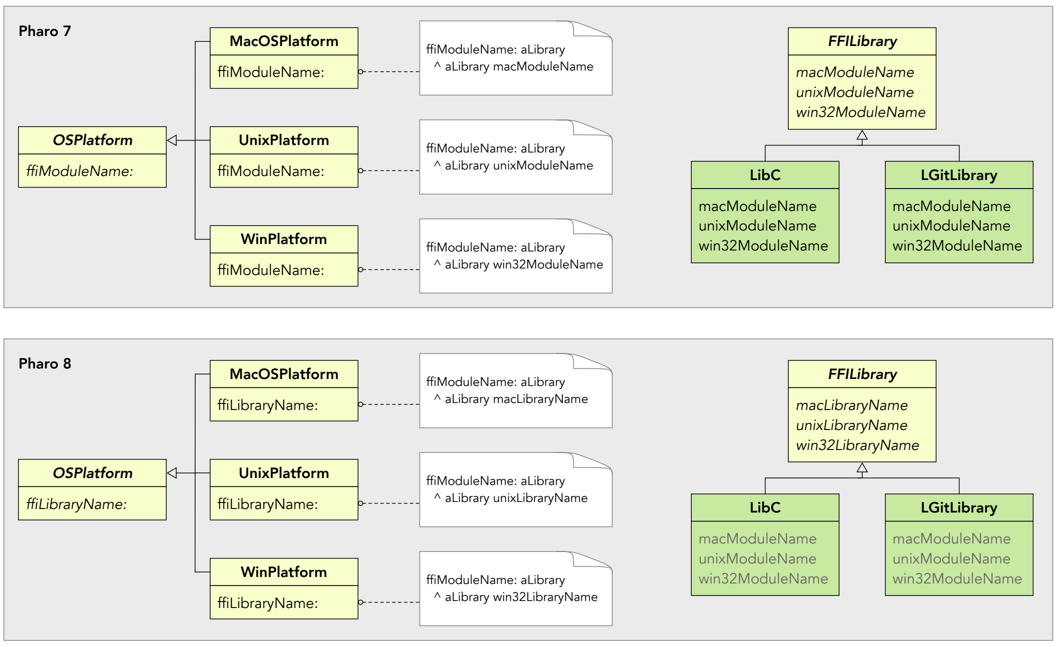


Figure 8.4: Yellow classes belong to the library and green classes are clients. This system follows the Template Method design pattern — client systems are expected to provide implementations for the methods that are called by the library. When those abstract hooks methods in the library get renamed, the clients must also rename their implementations. However, the current deprecation mechanism does not allow one to deprecate the hooks and notify the client, because in this case it is a library that makes calls to client methods, and not a client who calls the methods of a library.

their subclasses of FFILibrary, such as LibC and LGitLibrary. But it is not possible to deprecate these methods and warn clients who did not perform the renaming.

Why is it challenging? The standard deprecation mechanism works in such way that the old methods of the library are marked as deprecated and clients receives a deprecation warning when they call those methods. In this case, however, clients do not call any methods of the library. It is the library that calls certain methods of the client. Two client classes in our example — LibC and LGitLibrary — are completely unaware that the hook methods from their superclass, which they override, were renamed. As a result, the clients will only notice that renaming took place when library calls the abstract methods macLibraryName, unixLibraryName, and winLibraryName and raise an exception.

How can it be fixed? One way of overcoming this challenge would be to install a smart deprecation into the abstract hook that needs to be implemented. This deprecation could check if the caller provides the method with old name, and if yes, the deprecation warning will appear suggesting to rename that method. This approach could also be used to automatically rewrite the client code, with a technique similar to the one that will be discussed in Chapter 5.

8.5 Case 4: Cleaning Up Spurious Objects

Description. Some methods accept input through complex objects that contain multiple values as attributes. For example, a method can access a Point object that contains two values, x and y. The problem arises when such a method needs to be refactored to accept the values directly and not through an object. For example, a method distance(point1, point2) can be replaced with distance(x1, y1, x2, y2). In this case, to update the client code one needs to change not only the method call but also the place where Point objects are created.

Example. We discovered this scenario in the deprecation of the fraction:offset: group of methods from Morphic [Ducasse 2017] graphics library.

LayoutFrame is a basic data structure for Morphic graphics that defines a transformation frame relative to some rectangle. It can be used to define an area of a parent widget occupied by the child widget. This area is defined by two sets of numbers: *fractions* — the fractional distance (between 0 and 1) to place the morph in its owner's bounds, and *offsets* — the fixed pixel offset to apply after fractional positioning (for example, "10 pixel right of the center of the owner").

The old version of Morphic provided a flexible API that allowed clients to initialize a LayoutFrame in two different ways. One way required 4 arguments: bottomOffset, rightOffset, bottomFraction, and rightFraction. The other way involved representing fractions and offsets as two rectangle objects.

In the code listing below, we demonstrate, how the method bottomFraction: rightFraction: bottomOffset: rigthOffset: can be used to position a submorph inside a parent morph (think of it as positioning a child widget inside a parent widget). The parent morph in this example is 300 pixels wide and 200 pixels high. The child morph has 40% of its parent's width and 25% of its height. The center of a child morph is positioned 46 pixels to the bottom and 71 pixels to the left (negative right-offset of -71 pixels) from the center of a parent morph. The result can be seen in Figure 8.5.

```
frame := LayoutFrame identity
  bottomFraction: 0.25;
  rightFraction: 0.4;
  bottomOffset: 46;
  rigthOffset: -71;
  yourself.
```

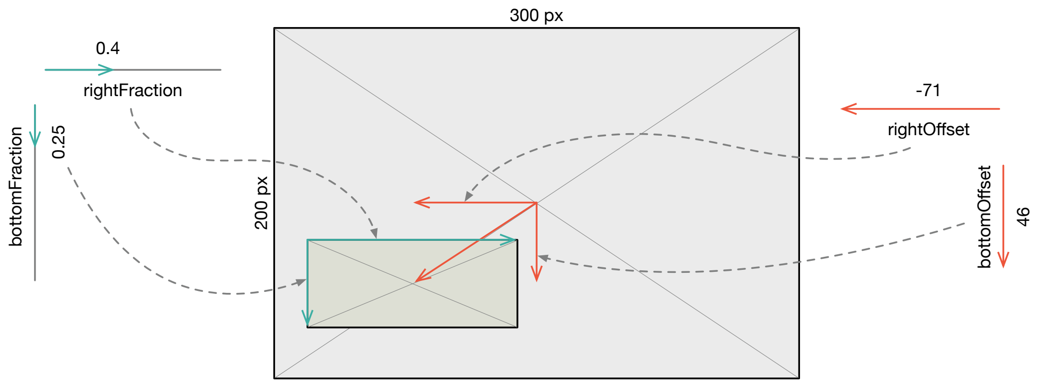


Figure 8.5: To position a submorph inside a parent morph, client provides four parameters: bottomFraction, rightFraction, bottomOffset, and rightOffset. All four are numbers. They can represent percentages (in the case of fractions), or be negative (in the case of offsets).

In the next listing, we demonstrate the other method fractions: offsets: that did the same thing but accepted two arguments: the fraction and the offset rectangles. The sides of those rectangles corresponded to the values of fractions and offsets. The visualization of this can be seen in Figure 8.6.

```
fractionRectangle := 0@0 extent: 0.4@0.25.
offsetRectangle := 0@0 extent: 46@(-71).

frame := LayoutFrame
```

```
    fractions: fractionRectangle
    offsets: offsetRectange.
```

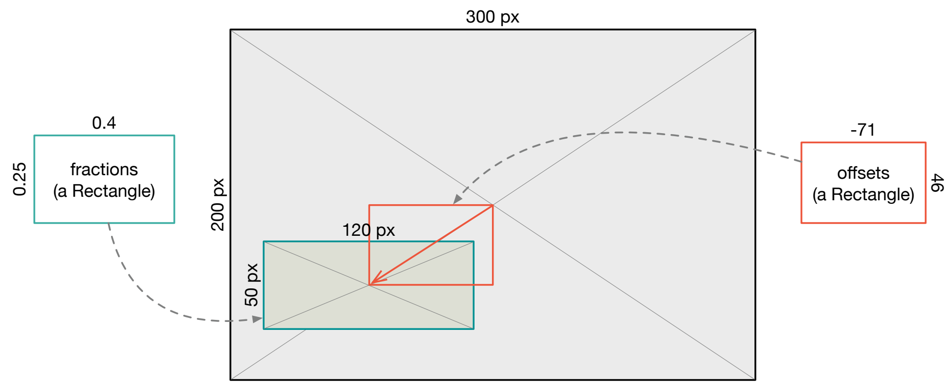


Figure 8.6: This way of positioning a submorph requires only two arguments: fractions and offsets. Both of them are rectangles, which introduces two problems: (1) the offsets rectangle can have negative sides, (2) the sides of a fractions rectangle are percentages. Both issues violate the nature of a rectangle and turn it from a geometrical object into a simple data structure that is used for storing and passing numbers around.

This is a misuse of Rectangle class which was intended to represent a geometrical object but now it is used as a value holder. As a result, in the example above, user creates a rectangle with negative width, which can create all sorts of problems.

Those were the considerations that lead to the removal of method fractions: offsets: in the recent version of Morphic. However, as we will see, automatically rewriting the call-sites of this method is not an easy task.

Why is it challenging?

- It can be difficult to find the part of client code where objects are created. They may be initialized just before the method call, or passed by argument from a different method.

- Those objects can also be used elsewhere in the client system. So before deleting them, one must make sure to check all the references.

How can it be fixed? In the following listing, we show how the fractions:offsets: method can be deprecated informing client developers that they should use the method with four arguments instead.

```
LayoutFrame >> fractions: fractionsOrNil offsets: offsetsOrNil

  | fractions offsets |
  self deprecated: 'Do not use this method.
  It forces to create spurious objects (rectangle or points)
  for nothing. Use bottomFraction: rightFraction:
     bottomOffset:
  rigthOffset: instead'.

  fractions := fractionsOrNil ifNil: [0@0 extent: 0@0].
  offsets := offsetsOrNil ifNil: [0@0 extent: 0@0].

  ↑ self
    topFraction: fractions top offset: offsets top;
    leftFraction: fractions left offset: offsets left;
    bottomFraction: fractions bottom offset: offsets bottom;
    rightFraction: fractions right offset: offsets right
```

In this case, rewriting client code is a challenging task that goes beyond the scope of this thesis. This problem could be addressed in the future work.

8.6 Case 5: When String Literals are Used as Identifiers

Description. In this situation, instead of rewriting the identifier name, library developers need to rewrite string literals that are used by clients. This typically occurs when library provides access to certain objects by a string identifier: icons, files, columns of a dataset, etc. The language of transformation rules must be flexible enough to allow for pattern matching inside string literals.

Example. In this example, clients of the library can access icons using the iconNamed: method which accepts one argument — the name of the icon:

```
ClyTestResultProperty >> createIcon

  allCount = 0 ifTrue: [ ↑ self iconNamed: #testNotRunIcon ].
```

In the old version all icon names ended with word 'Icon'. This was redundant, so in the new version, all icon names were changed to drop the word 'Icon' from the end:.

```
ClyTestResultProperty >> createIcon

  allCount = 0 ifTrue: [ ↑ self iconNamed: #testNotRun ].
```

Now all matches of '‘@rec iconNamed: #*Icon’ must be rewritten as '‘@rec icon-Named: #*’.

Why is it challenging? This goes beyond the pattern matching scenarios that are performed by tools such as Deprewriter (see Chapter 5). Those tools and the language that they use must be extended to support the pattern matching and transformation inside string literals.

How can it be fixed? Although Deprewriter can be extended to handle string transformations, the implementation and validation of such an extension goes beyond the scope of this thesis. It could be the focus of the future work.

8.7 Chapter Conclusion

In this chapter, we presented several case studies of challenging scenarios of library update. This is not an exhaustive list, we have only documented several challenging case studies that were reported to us by the Pharo community. Those scenarios can serve as examples of how to extend the language and tools for automatic code transformation. They can also be addressed in the future work on automatic library update.

CHAPTER 9
Conclusion

Contents

9.1 Summary

Modern software often depends on multiple external libraries and frameworks, which in turn can also depend on other libraries and frameworks. Like any other software, libraries evolve. They release new versions, often incompatible with the previous ones. This forces client applications that depend on those libraries to update their code in response to library evolution — a process that is known as *library update*.

Updating the dependencies can be a difficult and time consuming task for client developers. It involves repetitive operations and requires knowledge about the changes that were made to the library. In recent years, multiple approaches have been proposed to mine the data or apply machine learning techniques and extract knowledge about library update in the form of rules. However, most of those approaches focus on client developers and do not consider the expertise of library developers. They consider only simple method-to-method replacements and are only designed for statically-typed programming languages.

In this thesis, we address these shortcomings in the literature by exploring the problem of library update from the perspective of library developers and in the context of dynamically-typed programming languages. Below, we summarize the work that we presented in each chapter.

Chapter 2 presents the background that is necessary for understanding the research presented in this thesis. We explained the process of library evolution, defined the terminology that we use to refer to all of its objects and actors, and discussed the spectrum of problems that arise in this field. Inside that spectrum, we defined the specific scope of problems that are addressed in this thesis. We finished the chapter with several motivating examples that demonstrate different scenarios of library update, explain why it can be hard, and how automatic tools can be used to support library and client developers in those cases.

Chapter 3 contains an overview of the literature in the field of library update. It includes the discussion of empirical studies of library evolution from the perspective of library and client developers, the studies on code transformation and deprecations that are related to our work on transforming deprecations, and the studies that propose the tools to guide library or client developers in the process of library update. We identified several shortcomings of existing studies that are addressed in the following chapters of this thesis.

Chapter 4 presents the results of two surveys: one of library developers and another one of client developers. The surveys involved software developers from two industrial companies, Arolla and Berger-Levrault, and one open-source community, Pharo. The questions were general and not related to specific libraries or issues. The focus of the study, presented in this chapter, is the perception of the impact of library evolution from different perspectives and the type of support that library developers can provide to help their clients.

Chapter 5 contains the discussion of *Deprewriter* — a mechanism of transforming deprecations that is supported by Pharo. It allows library developers to annotate method deprecations with transformation rules which can be automatically applied to client code. After describing the approach, we answered two categories of research questions: the first category is about the use and flexibility of the rewriting deprecations as used by library developers and the second category is about the perception of external users of the rewriting deprecations. We then presented two validations: an analysis of the Deprewriter rules used in the deprecations of Pharo 8 and a user study.

Chapter 6 presents an approach and a tool called *DepMiner* that helps library developers identify breaking changes before the release, understand when and by whom they were introduced, and find the potential replacements that could be suggested to the clients. The tool generates recommendations in the form of transformation rules that can be used by Pharo's Deprewriter. Inspired by the existing approaches that were proposed to support the client developers, our approach is based on the frequent method call analysis. We implemented our approach as a tool for Pharo IDE, applied it to five open-source projects, and asked 6 core developers from those projects to accept or reject the recommended changes. In total, 134 proposed deprecations were accepted by developers as well as 4 transformation rules for the existing deprecations. 61 new deprecations and 2 transformations rules for existing deprecations were integrated into the Pharo project.

Chapter 7 presents the first steps towards developing a holistic approach to help library developers support their clients. We have discussed the needs of library developers and identified six ways how they can respond to breaking changes in their systems. We have also proposed adaptations to the DepMiner approach that would allow to overcome some of its limitations. Finally, we have proposed the first sketch of a holistic approach to support library developers. Implementation and validation of such an approach can be the focus of future work.

Chapter 8 presents several case studies of challenging scenarios of library update that were reported to us by the Pharo community. Those are the situations that are hard to handle with current deprecation mechanisms or hard to automate with rules that could be applied to client code. Those scenarios can serve as examples of how to extend the language and tools for automatic code transformation. They should also be addressed in future work on automatic library update.

9.2 Contributions

The main contributions of this thesis are:

- A survey of library and client developers from two industrial companies and an open-source community;
- First detailed documentation of the Deprewriter approach and tool in Pharo which introduces deprecations that dynamically update client code with transformation rules;
- A study of how Deprewriter was adopted by the Pharo community through the analysis of source code in Pharo 8 and a developer survey;
- DepMiner — a novel approach to infer the rules for Deprewriter based on the commit history of a project;
- A generalization of DepMiner as a new holistic approach to support library developers in the task of library update.

9.3 Future Work

In this section, we present the open issues that were not addressed in this thesis. These open issues provide opportunities to continue our research concerning the data mining-based tools to support library update.

Overcoming the limitations of DepMiner. In Section 6.5, we discussed several shortcomings of the DepMiner approach and tool that supports library developers by mining frequent method call replacements from the commit history and generating transformation rules for method deprecations. In Section 7.5, we proposed a way to overcome two of those shortcomings:

- *Unused/untested methods.* DepMiner is based on detecting the changes in how the library uses its own API. Therefore, it is ineffective for the methods that are not used internally by the library and not covered by unit tests. To overcome this limitation, we propose to combine the mining of frequent method call replacements with an approach to detect refactoring operations in the commit history (see Section 7.5.1 for more details).
- *Searching the entire commit history.* To detect the response of library to its own changes, DepMiner searches the entire commit history. We propose to improve its performance by identifying the specific commit in which the breaking change was introduced and only analysing the neighboring commits to detect how library reacts to that change (see Section 7.5.2 for more details).

Implementing a holistic approach to support library developers. By discussing with library developers and observing their reaction to recommendations proposed by DepMiner, we concluded that recommending the correct replacement for removed functionality is not always enough to help library developers support their clients. We identified six responses that library developers can have to breaking changes, based on whether or not they want to deprecate and whether or not the automatable replacement can be found (see Section 7.4). Based on those six scenarios, a more general approach can be developed to support the developers of evolving libraries: documentation, deprecations, library update script, etc. In Section 7.6, we proposed a blueprint for such a holistic approach.

Dealing with circular renaming. In Chapter 8, we have presented multiple case studies of the challenging library update problem that were reported to us by the Pharo community. The first of those scenarios involved reassigning the existing name to a different entity $a \to b$, $b \to c$ (see Section 8.2) and a more challenging case when both names are being exchanged $a \to b$, $b \to a$ (see Section 8.3).

Such transformations can cause collisions. They are especially challenging in dynamically-typed programming languages because it is difficult to find all references to a given method statically. We proposed two ways to overcome this challenge: using a temporary name or logging all locations in source code where transformations have already been applied.

Rewriting abstract hooks. Another challenging scenario arises when library developers need to introduce a breaking change into an abstract hook (for example, rename it). An abstract hook is a method that needs to be implemented by clients. In this case, clients do not call the library, but instead the library calls the implementation of a client. The traditional deprecation mechanisms do not cover this case. It is also overlooked in the related works that propose to support developers through automatic rewriting of client code. We discussed this scenario in Section 8.4 and proposed to overcome it by installing a transforming deprecation into the abstract hook itself. When the client method is not present, the hook would be invoked on the library side. This could trigger the rewriting methanism.

Removing spurious objects. In Section 8.5, we discussed the situation when methods that accept complex objects as arguments need to be refactored in such a way that those complex arguments are decomposed or removed. Rewriting client code in this case can leave spurious objects. Removing such objects is challenging because it can be hard to find the part of client code where the object is created and hard to make sure that the object is not used elsewhere (see Section 8.5 for a detailed example). Such code transformations go beyond the scope of this thesis.

Rewriting literals. The final challenging scenario that we reported in Section 8.6 involves rewriting string literals based on a given pattern. In this thesis, we focused on rewriting method calls. The Deprewriter tool that we presented in Chapter 5 allows library developers to dynamically rewrite the calls to deprecated methods. However, the language of code transformations that is used by Deprewriter can be extended to find and rewrite string literals in client code based on a given pattern.

APPENDIX A
Pharo syntax in a nutshell

To help readers follow the code snippets we present briefly the syntax of Pharo.[1] 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]For a more detailed introduction into the syntax of Pharo, please visit https://pharo.org/documentation.html

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.

```
Integer >> slowFactorial [
    "Answer the factorial of the receiver."
    self = 0 ifTrue: [↑ 1].
    self > 0 ifTrue:
            [↑ self * (self - 1) slowFactorial ]
]
```

Appendix B

Transformation Rules of Deprewriter Extracted from Pharo 8

In this appendix, we present examples of transformation rules that we found in Pharo 8. We classify those rules into 6 categories according to the scenario that led to their introduction. Those are four refactoring scenarios (*Rename Method*, *Remove Argument*, *Add Argument*, *Split Method*), one scenario that involved *changing the receiver* of the method, and one scenario that describes a complex replacement that does not fit any of the above mentioned categories. The examples can be found on the following page in Table B.1.

Table B.1: Examples of the transformation rules extracted from Pharo 8.

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	‘@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 arg.	‘@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]

Bibliography

[Alpert 1998] Sherman R. Alpert, Kyle Brown and Bobby Woolf. The design patterns Smalltalk companion. Addison Wesley, Boston, MA, USA, 1998. 75, 113

[Alrubaye 2019] Hussein Alrubaye and Ali Mkaouer Mohamed Wiemand Ouni. *On the Use of Information Retrieval to Automate the Detection of Third-Party Java Library Migration at the Method Level.* In ICPC'19, 2019. 21, 22, 24, 46, 82

[Alrubaye 2020] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni and Jason Mcgoff. *Learning to recommend third-party library migration opportunities at the API level.* Journal of Applied Software Computing, pages 106–140, 2020. 21, 22, 25

[Anquetil 2022] Nicolas Anquetil, Julien Delplanque, Stéphane Ducasse, Oleksandr Zaitsev, Christopher Furhman and Yann-Gael Guéhéneuc. *What do developers consider magic literals? A smalltalk perspective*. Information and Software Technology, 2022. 5

[Baldassarre 2005] Maria Teresa Baldassarre, Alessandro Bianchi, Danilo Caivano and Giuseppe Visaggio. *An industrial case study on reuse oriented development.* In 21st IEEE International Conference on Software Maintenance (ICSM'05), pages 283–292. IEEE, 2005. 1, 7

[Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Pharo by example. Square Bracket Associates, Kehrsatz, Switzerland, 2009. 3, 57

[Bodden 2011] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati and Mira Mezini. *Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders*. In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM. 93

[Bogart 2016] Christopher Bogart, Christian Kästner, James Herbsleb and Ferdian Thung. *How to break an API: cost negotiation and community values in three software ecosystems.* In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 109–120, 2016. 18, 19, 20, 27, 28, 43

[Brito 2016] Gleison Brito, Andre Hora, Marco Tulio Valente and Romain Robbes. *Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems*. In International Conference on Software Analysis, Evolution, and Reengineering (SANER), pages 360–369. IEEE, 2016. 20

[Brito 2018a] Aline Brito, Laerte Xavier, André C. Hora and Marco Tulio Valente. *APIDiff: Detecting API breaking changes*. International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 507–511, 2018. 9, 46

[Brito 2018b] Gleison Brito, Andre Hora, Marco Tulio Valente and Romain Robbes. *On the use of replacement messages in API deprecation: An empirical study*. Journal of Systems and Software, vol. 137, pages 306–321, 2018. 9, 10, 20, 23, 81, 82, 83, 98

[Brito 2019] Aline Brito, Marco Tulio Valente, Laerte Xavier and Andre Hora. *You broke my code: understanding the motivations for breaking changes in APIs*. Empirical Software Engineering, pages 1–35, 2019. 10, 18, 19, 20, 27, 28, 34, 35, 43, 97, 98

[Brito 2020] Aline Brito, Marco Tulio Valente, Laerte Xavier and Andre Hora. *You broke my code: understanding the motivations for breaking changes in APIs*. Empirical Software Engineering, vol. 25, no. 2, pages 1458–1492, 2020. 81

[Callau 2011] Oscar Callau, Romain Robbes, David Rothlisberger and Eric Tanter. *How developers use the dynamic features of programming languages: the case of Smalltalk*. In Mining Software Repositories International Conference (MSR'11), 2011. 93

[Chern 2007] Rick Chern and Kris De Volder. *Debugging with Control-flow Breakpoints*. In International Conference on Aspect-Oriented Software Development (AOSD'07), pages 96–106, New York, NY, USA, 2007. ACM. 58

[Chow 1996] Kingsum Chow and David Notkin. *Semi-automatic update of applications in response to library changes*. In International Conference on Software Maintenance (ICSM), volume 96, page 359, 1996. 21, 22, 23, 47

[Colyer 2005] A. Colyer and Clement A. *Aspect-oriented programming with AspectJ*. IBM Systems Journal, vol. 44, no. 2, pages 301–308, 2005. 58

[Dagenais 2008] Barthélémy Dagenais and Martin P. Robillard. *Recommending adaptive changes for framework evolution*. In International Conference on Software Engineering (ICSE'08), pages 481–490, New York, NY, USA, 2008. ACM. 21, 22, 24, 46, 82

[Dagenais 2011] Barthélémy Dagenais and Martin P Robillard. *Recommending adaptive changes for framework evolution*. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 20, no. 4, pages 1–35, 2011. 23, 85, 95, 98, 99, 104

[Delplanque 2019] Julien Delplanque, Stéphane Ducasse and Oleksandr Zaitsev. *Magic Literals in Pharo*. In International workshop of Smalltalk Technologies, 2019. 6

[Demeyer 2002] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz. Object-oriented reengineering patterns. Morgan Kaufmann, 2002. 1, 7

[Dig 2006a] Danny Dig, Can Comertoglu, Darko Marinov and Ralph Johnson. *Automated Detection of Refactorings in Evolving Components*. In ECOOP, pages 404–428, 2006. 82, 98, 104, 105, 106

[Dig 2006b] Danny Dig and Ralph Johnson. *How do APIs evolve? A story of refactoring*. Journal of Software Maintenance and Evolution: Research and Practice (JSME), vol. 18, no. 2, pages 83–107, April 2006. 49, 57, 98, 100, 104

[Ducasse 2017] Stéphane Ducasse, Dmitri Zagidulin, Nicolai Hess, Dimitris Chloupis Originally written by A. Black, S. Ducasse, O. Nierstrasz, D. Pollet with D. Cassou and M. Denker. Pharo by example 5. Square Bracket Associates, 2017. 115

[Ducasse 2022] Stéphane Ducasse, Guillermo Polito, Oleksandr Zaitsev, Marcus Denker and Pablo Tesone. *Deprewriter: On the fly rewriting method deprecations*. Journal of Object Technologies (JOT), vol. 21, no. 1, 2022. 5, 32, 42, 45, 47, 73, 81

[Furr 2009] Michael Furr, Jong hoon (David) An, Jeffrey S. Foster and Michael Hicks. *Static Type Inference for Ruby*. In Symposium on Applied Computing (SAC'09), 2009. 50, 84

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design patterns: Elements of reusable object-oriented software. Addison-Wesley, 1995. 112

[Hayden 2012] Christopher M Hayden, Edward K Smith, Michail Denchev, Michael Hicks and Jeffrey S Foster. *Kitsune: Efficient, general-purpose dynamic software updating for C*. In International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'12), pages 249–264, 2012. 64

[Henkel 2005] Johannes Henkel and Amer Diwan. *CatchUp!: capturing and replaying refactorings to support API evolution*. In Proceedings International Conference on Software Engineering (ICSE 2005), pages 274–283, 2005. 21, 22, 23

[Hölzle 1991] Urs Hölzle, Craig Chambers and David Ungar. *Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches*. In P. America, editeur, Proceedings ECOOP '91, volume 512 of *LNCS*, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag. 50

[Hora 2014] Andre Hora, Anne Etien, Nicolas Anquetil, Stéphane Ducasse and Marco Túlio Valente. *APIEvolutionMiner: Keeping API Evolution under Control*. In Proceedings of the Software Evolution Week (CSMR-WCRE'14), 2014. 21, 22, 24, 46, 82, 85, 95, 98, 103

[Hora 2015] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse and Marco Túlio Valente. *How Do Developers React to API Evolution? The Pharo Ecosystem Case*. In International Conference on Software Maintenance (ICSM'15), pages 251–260, 2015. 10, 18, 20, 27, 98

[Hora 2018] André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien and Stéphane Ducasse. *How do Developers React to API Evolution? a Large-Scale Empirical Study*. Software Quality Journal, vol. 26, pages 161–191, March 2018. 18, 20, 27

[Jezek 2015] Kamil Jezek, Jens Dietrich and Premek Brada. *How Java APIs break–an empirical study*. Information and Software Technology, vol. 65, pages 129–146, 2015. 18, 19, 27

[Kiczales 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. *Aspect-Oriented Programming*. In Mehmet Aksit and Satoshi Matsuoka, editeurs, European Conference on Object-Oriented Programming (ECOOP'97), pages 220–242. Springer-Verlag, June 1997. 62

[Kiczales 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold. *An overview of AspectJ*. In European

Conference on Object-Oriented Programming (ECOOP' 01), numéro 2072 de LNCS, pages 327–353. Springer Verlag, 2001. 58

[Kim 2007] Miryung Kim, David Notkin and Dan Grossman. *Automatic inference of structural changes for matching across program versions*. In International Conference on Software Engineering (ICSE'07), pages 333–343. IEEE, 2007. 21, 22, 23, 24, 46, 95

[Kula 2018a] Raula Gaikovina Kula, Daniel M German and Ali Ouni andTakashi Ishio andKatsuro Inoue. *Do developers update their library dependencies?* Empirical Software Engineering, vol. 23, pages 384–417, 2018. 18, 19, 20, 27

[Kula 2018b] Raula Gaikovina Kula, Ali Ouni, Daniel M German and Katsuro Inoue. *An empirical study on the impact of refactoring activities on evolving client-used APIs*. Information and Software Technology, vol. 93, pages 186–199, 2018. 18, 19, 27, 28, 43

[Lehman 1996] Manny Lehman. *Laws of Software Evolution Revisited*. In European Workshop on Software Process Technology, pages 108–124, Berlin, 1996. Springer. 1, 7

[Meng 2012] Sichen Meng, Xiaoyin Wang, Lu Zhang and Hong Mei. *A history-based matching approach to identification of framework evolution*. In International Conference on Software Engineering (ICSE), pages 353–363. IEEE, 2012. 21, 22, 24, 46, 95

[Mens 2004] Tom Mens, Juan F. Ramil and Michael W. Godfrey. *Analyzing the Evolution of Large-Scale Software: Issue Overview*. Journal of Software Maintenance and Evolution: Research and Practice, vol. 16, no. 6, pages 363–365, November 2004. 1, 7

[Milner 1978] Robin Milner. *A Theory of Type Polymorphism in Programming*. Journal of Computer and System Sciences, vol. 17, pages 348–375, 1978. 49, 84

[Milojković 2016] Nevena Milojković, Clément Béra, Mohammad Ghafari and Oscar Nierstrasz. *Inferring Types by Mining Class Usage Frequency from Inline Caches*. In International Workshop on Smalltalk Technologies IWST'16, Prague, Czech Republic, August 2016. 50, 84

[Murphy-Hill 2009] Emerson Murphy-Hill, Chris Parnin and Andrew P. Black. *How We Refactor, and How We Know It*. In International Conference on Software Engineering (ICSE), pages 287–297, 2009. 57

[Murphy-Hill 2011] Emerson Murphy-Hill, Chris Parnin and Andrew P Black. *How we refactor, and how we know it*. IEEE Transactions on Software Engineering, vol. 38, no. 1, pages 5–18, 2011. 57

[Nascimento 2020] Romulo Nascimento, Aline Brito, Andre Hora and Eduardo Figueiredo. *JavaScript API deprecation in the wild: A first assessment*. In International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 567–571. IEEE, 2020. 10, 20, 81, 83, 98

[Neamtiu 2006] Iulian Neamtiu, Michael W. Hicks, Gareth Stoyle and Manuel Oriol. *Practical dynamic software updating for C*. In Programming Language Design and Implementation (PLDI), pages 72–83, 2006. 64

[Nguyen 2010] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim and Tien N Nguyen. *A graph-based approach to API usage adaptation*. In Conference on Object-Oriented Programming, Systems and Applications (OOPSLA'10), pages 302 – 321, 2010. 21, 22, 24, 46

[Pandita 2015] Rahul Pandita, Raoul Praful Jetley, Sithu D Sudarsan and Laurie Williams. *Discovering likely mappings between APIs using text mining*. In International Working Conference on Source Code Analysis and Manipulation (SCAM), pages 231–240. IEEE, 2015. 21, 22, 24, 46, 82

[Passerini 2014] Nicolás Passerini, Pablo Tesone and Stéphane Ducasse. *An extensible constraint-based type inference algorithm for object-oriented dynamic languages supporting blocks and generic types*. In International Workshop on Smalltalk Technologies (IWST'14), August 2014. 50, 84

[Pina 2013] Luis Pina and Michael Hicks. *Rubah: Efficient, General-purpose Dynamic Software Updating for Java*. In International Workshop on Hot Topics in Software Upgrades (HotSWUp), 2013. 64

[Pluquet 2009] Frédéric Pluquet, Antoine Marot and Roel Wuyts. *Fast type reconstruction for dynamically typed programming languages*. In Dynamic Languages Symposium (DLS), pages 69–78, New York, NY, USA, 2009. ACM. 50, 84

[Ren 2016] Brianna M. Ren and Jeffrey S. Foster. *Just-in-time Static Type Checking for Dynamic Languages*. In Conference on Programming Language Design and Implementation (PLDI), 2016. 50, 84

[Renggli 2010a] Lukas Renggli. *Dynamic Language Embedding With Homogeneous Tool Support*. PhD thesis, University of Bern, October 2010. 51

[Renggli 2010b] Lukas Renggli, Tudor Gîrba and Oscar Nierstrasz. *Embedding Languages Without Breaking Tools*. In Theo D'Hondt, editeur, Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10), volume 6183 of *LNCS*, pages 380–404. Springer-Verlag, 2010. 81

[Richards 2011] Gregor Richards, Christian Hammer, Brian Burg and Jan Vitek. *The Eval that Men Do: A Large-scale Study of the Use of Eval in JavaScript Applications*. In Proceedings of Ecoop 2011, 2011. 93

[Rivard 1996] Fred Rivard. *Smalltalk: a Reflective Language*. In Proceedings of REFLECTION'96, pages 21–38, April 1996. 46, 62

[Rizun 2015] Markiyan Rizun, Jean-Christophe Bach and Stéphane Ducasse. *Code Transformation by Direct Transformation of ASTs*. In International Workshop on Smalltalk Technologies (IWST), 2015. 55

[Rizun 2016] Markiyan Rizun, Gustavo Santos, Stéphane Ducasse and Camille Teruel. *Phorms: Pattern Combinator Library for Pharo*. In International Workshop on Smalltalk Technologies IWST'16, Prague, Czech Republic, August 2016. 55

[Robbes 2012a] Romain Robbes, Mircea Lungu and David Röthlisberger. *How Do Developers React to API Deprecation?: The Case of a Smalltalk Ecosystem*. In International Symposium on the Foundations of Software Engineering (FSE), pages 56:1–56:11, New York, NY, USA, 2012. ACM. 2, 18, 27, 95

[Robbes 2012b] Romain Robbes, David Röthlisberger and Éric Tanter. *Extensions during software evolution: do objects meet their promise?* In European Conference on Object-Oriented Programming (ECOOP), pages 28–52, Berlin, Heidelberg, 2012. Springer-Verlag. 20

[Roberts 1996] Don Roberts, John Brant, Ralph E. Johnson and Bill Opdyke. *An Automated Refactoring Tool*. In Proceedings of ICAST '96, April 1996. 46, 51, 53, 62, 81

[Roberts 1997] Don Roberts, John Brant and Ralph E. Johnson. *A Refactoring Tool for Smalltalk*. Theory and Practice of Object Systems (TAPOS), vol. 3, no. 4, pages 253–263, 1997. 46, 51, 53, 62, 81

[Roberts 1999] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999. 51, 53

[Sandewall 1978] Erik Sandewall. *Programming in an Interactive Environment: The "Lisp" Experience*. ACM Comput. Surv., vol. 10, no. 1, pages 35–71, March 1978. 64

[Sawant 2016] Anand Ashok Sawant, Romain Robbes and Alberto Bacchelli. *On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs*. In International Conference on Software Maintenance and Evolution (ICSME), pages 400–410. IEEE, 2016. 18, 20, 27, 95

[Schäfer 2008] Thorsten Schäfer, Jan Jonas and Mira Mezini. *Mining framework usage changes from instantiation code*. In International Conference on Software Engineering (ICSE), pages 471–480, New York, NY, USA, 2008. ACM. 21, 22, 24, 46, 76, 82, 85, 95, 99, 103

[Schärli 2004] Nathanael Schärli, Andrew P. Black and Stéphane Ducasse. *Object-oriented Encapsulation for Dynamically Typed Languages*. In Proceedings of 18th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'04), pages 130–149, October 2004. 83

[Smith 1984] Brian Cantwell Smith. *Reflection and Semantics in Lisp*. In Proceedings of POPL'84, pages 23–3, 1984. 58

[Spoon 2004] S. Alexander Spoon and Olin Shivers. *Demand-Driven Type Inference with Subgoal pruning: Trading Precision for Scalability*. In Proceedings of ECOOP'04, pages 51–74, 2004. 50, 84

[Steyaert 1996] Patrick Steyaert, Carine Lucas, Kim Mens and Theo D'Hondt. *Reuse Contracts: Managing the Evolution of Reusable Assets*. In Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'96, pages 268–285. ACM Press, 1996. 113

[Suzuki 1981] Norihisa Suzuki. *Inferring types in Smalltalk*. In Symposium on Principles of Programming Languages (POPL'81), pages 187–199, New York, NY, USA, 1981. ACM Press. 49, 50, 84

[Teyton 2013] Cédric Teyton, Jean-Rémy Falleri and Xavier Blanc. *Automatic discovery of function mappings between similar libraries*. In Working Conference on Reverse Engineering (WCRE), pages 192–201. IEEE, 2013. 21, 22, 24, 46, 82, 95, 99, 103

[Wohlin 2000] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell and Anders Wesslén. Experimentation in software engineering: an introduction. Kluwer Academic Publishers, Norwell, MA, USA, 2000. 41

[Wu 2010] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol and Miryung Kim. *Aura: a hybrid approach to identify framework evolution*. In International Conference on Software Engineering (ICSE), volume 1, pages 325–334. IEEE, 2010. 21, 22, 24, 46, 95

[Xavier 2017a] Laerte Xavier, Aline Brito, Andre Hora and Marco Tulio Valente. *Historical and impact analysis of API breaking changes: A large-scale study*. In International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 138–147. IEEE, 2017. 18, 19, 27, 28, 81, 97

[Xavier 2017b] Laerte Xavier, Andre Hora and Marco Tulio Valente. *Why do we break APIs? first answers from developers*. In International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 392–396. IEEE, 2017. 18, 19, 20, 27, 28, 43, 81

[Xing 2007] Eleni Xing ZhenchangandStroulia. *API-evolution support with diff-catchup*. IEEE Transactions on Software Engineering, vol. 33, pages 818 – 836, 2007. 21, 22, 23, 46, 95

[Yau 1978] Stephen S. Yau, J. S. Collofello and T. MacGregor. *Ripple effect analysis of software maintenance*. In The IEEE Computer Society's Second International Computer Software and Applications Conference, pages 60–65. IEEE Press, nov 1978. 2

[Zaitsev 2020a] Oleksandr Zaitsev, Stéphane Ducasse and Nicolas Anquetil. *Characterizing Pharo Code: A Technical Report*. Technical report, Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille ; Arolla, January 2020. 6

[Zaitsev 2020b] Oleksandr Zaitsev, Stéphane Ducasse, Alexandre Bergel and Mathieu Eveillard. *Suggesting Descriptive Method Names: An Exploratory Study of Two Machine Learning Approaches*. In International Conference on the Quality of Information and Communications Technology, pages 93–106. Springer, 2020. 5

[Zaitsev 2022a] Oleksandr Zaitsev, Stéphane Ducasse, Nicolas Anquetil and Arnaud Thiefaine. *DepMiner: Automatic Recommendation of Transformation*

Rules for Method Deprecation. In ICSR 2022-20th International Conference on Software and System Reuse, 2022. 5, 83

[Zaitsev 2022b] Oleksandr Zaitsev, Stéphane Ducasse, Nicolas Anquetil and Arnaud Thiefaine. *How Libraries Evolve: A Survey of Two Industrial Companies and an Open-Source Community*. In 29th Asia-Pacific Software Engineering Conference, 2022. 5, 28

[Zaitsev 2022c] Oleksandr Zaitsev, Jordan Monta no Sebastian and Stéphane Ducasse. *How Fast is AI in Pharo?*
Benchmarking Linear Regression. In IWST 2022-International Workshop on Smalltalk Technologies, 2022. 5

[ZeroTurnAround 2012] ZeroTurnAround. *What developers want: The end of application Redeployes*. http://files.zeroturnaround.com/pdf/JRebelWhitePaper2012-1.pdf, 2012. 64