How Do Developers React to API Evolution? The Pharo Ecosystem Case

André Hora*†, Romain Robbes‡, Nicolas Anquetil†, Anne Etien†, Stéphane Ducasse†, Marco Tulio Valente*
*ASERG Group, Department of Computer Science (DCC)
Federal University of Minas Gerais, Brazil
{hora, mtov}@dcc.ufmg.br
†RMod team, Inria Lille Nord Europe
University of Lille, CRIStAL, UMR 9189, Villeneuve d'Ascq, France
{firstName.lastName}@inria.fr
‡PLEIAD Lab, Department of Computer Science (DCC)
University of Chile, Santiago, Chile
rrobbes@dcc.uchile.cl

***Abstract*—Software engineering research now considers that no system is an island, but it is part of an ecosystem involving other systems, developers, users, hardware, ... When one system (*e.g.,* a framework) evolves, its clients often need to adapt. Client developers might need to adapt to functionalities, client systems might need to be adapted to a new API, client users might need to adapt to a new User Interface. The consequences of such changes are yet unclear, what proportion of the ecosystem might be expected to react, how long might it take for a change to diffuse in the ecosystem, do all clients react in the same way? This paper reports on an exploratory study aimed at observing API evolution and its impact on a large-scale software ecosystem, Pharo, which has about 3,600 distinct systems, more than 2,800 contributors, and six years of evolution. We analyze 118 API changes and answer research questions regarding the magnitude, duration, extension, and consistency of such changes in the ecosystem. The results of this study help to characterize the impact of API evolution in large software ecosystems, and provide the basis to better understand how such impact can be alleviated.**

I. Introduction

As frameworks evolve, client systems often need to adapt their source code to use the updated API. To facilitate this time-consuming task, frameworks should be backward-compatible and include deprecated methods. In practice, researchers have found that frameworks are backward-incompatible [1] and deprecation messages are often missing [2]. To deal with these problems, some approaches have been developed to help client developers. This can be done, for example, with the support of specialized IDEs [3], the help of experts [4], or the inference of change rules [1], [5], [6], [7], [8].

Commonly, these approaches are evaluated on small-scale case studies. In practice, many software systems are part of a larger software ecosystem, which often exists in organizations, or open-source communities [9]. In this context, it is hard to predict the real impact of API evolution. For example, in ecosystems, API deprecation may affect hundreds of clients, with several of these clients staying in an inconsistent state for long periods of time or do not reacting at all [2]. This suggests that the impact of API evolution may be large and sometimes unknown; in this context, managing API evolution is a complex and risky task [10].

To support developers to better understand the real impact of API evolution and how it could be alleviated, software ecosystems should also be studied. In that respect, a first large-scale study was performed by one of the authors of this paper, Robbes *et al.* [2], to verify the impact of deprecated APIs on a software ecosystem. However, API evolution is not restricted to deprecation. It may imply, for example, a better API design that improves code legibility, portability, performance, security, etc. But are client developers aware of such evolving APIs? How frequent and how broad is the impact on clients? The aforementioned study analyzes the adoption of a specific group of changes, methods explicitly annotated as deprecated. But this introduces a bias as people will probably notice more readily changes documented and checked by the compiler (explicit deprecation) than changes not advertised. Therefore, there is still space for analyzing the adoption of more generic API changes (not explicitly marked as deprecated).

In this paper, we analyze the impact of API changes, not related to explicit API deprecation, on client systems. We set out to discover (i) to what extent API changes propagate to client systems, and (ii) to what extent client developers are aware of these changes. Our goal is to better understand, at the ecosystem level, to what extent client developers are affected by the evolution of APIs, and to reason about how it could be alleviated. Thus, we investigate the following research questions to support our study:

- RQ1 (Magnitude): How many systems react to API changes in an ecosystem and how many developers are involved?
- RQ2 (Duration): How long does it take for systems to react to API changes?
- RQ3 (Extension): Do all the systems in an ecosystem react to API changes?
- RQ4 (Consistency): Do systems react to an API change in the same way?

978-1-4673-7532-0/15 © 2015 IEEE

Accepted for publication by IEEE. © 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

In this study we cover the Pharo[1] software ecosystem, which has about 3,600 distinct systems, more than 2,800 contributors and six years of evolution, and we analyze 118 API changes. We also compare our results with the Robbes *et al.* study on API deprecation on Pharo [2] to better understand how these two types of API evolution affect client systems.

The contributions of this paper are summarized as follows:

1) We provide a large-scale study, at the ecosystem level, to better understand to what extent client developers are impacted by API changes that are not marked as deprecated.
2) We provide a comparison between our results and the ones of the previous API deprecation study [2].

Structure of the paper: In Section II, we present the API changes considered in this study. We describe our experiment design in Section III. We present and discuss the experiment results in Section IV. We present the implications of our study in Section V, and the threats to the validity in Section VI. Finally, we present related work in Section VII, and we conclude the paper in Section VIII.

II. API Changes

A. Definition

In this work, we focus on API changes related to method replacements and improvements, following the line studied by several researches in the context of framework migration [1], [5], [6], [7], [11], [12], [13], [8]. Next, we define and provide examples on the two types of API changes considered in this paper.

Method replacements: In this type of API change, one or more methods in the old release are replaced by one or more methods in the new release. For example, in a one-to-one mapping, the method LineConnection.end() was replaced by LineConnection.getEndConnector() from JHotDraw 5.2 to 5.3 [1]. In another case, in a one-to-many mapping, the method CutCommand(DrawingView) was replaced by CutCommand(Alignment, DrawingEditor) *and* UndoableCommand(Command) [1]. In both examples, the removed methods have not been deprecated; they were simply dropped, causing clients to fail.

Method improvements: In this type of API change, one (or more) method in the old release is improved, producing one (or more) new method in the new release. For example, in Apache Ant, the method to close files was improved to centralize the knowledge on closing files [13], producing a one-to-one mapping where calls to InputStream.close() should be replaced by FileUtils.close(InputStream). In this case, both solutions to close files are available in the new release, *i.e.,* both methods can be used. However, the latter is the suggested one in order to improve maintenance. In the Moose platform[2], a convention states that calls to MooseModel.root() and MooseModel.add(MooseModel) should be replaced by MooseModel.install() when adding models. Again, all the methods are available to be used, but MooseModel.install() is the suggested one to improve code legibility.[3]

These types of API changes are likely to occur during framework evolution, thus their detection is helpful for client developers. Recently, researchers proposed techniques to automatically infer rules that describe such API changes [1], [6], [7], [11], [12], [13]. In this study, we adopt our previous approach [7] in order to detect API changes, which covers both aforementioned cases.

B. Detecting API Changes

In our approach, API changes are automatically produced by applying the *association rules* data mining technique [14] on the set of method call changes between two versions of one method. We produce rules in the format old-call $\rightarrow$ new-call, indicating that the old call should be replaced by the new one. Each rule has a *support* and *confidence*, indicating the frequency that the rule occurs in the set of analyzed changes and a level of confidence. We also use some heuristics to filter rules that are more likely to represent relevant API changes; for example, rules can be ranked by confidence, support or occurrences in different revisions. Please, refer to our previous study [7] for an in-depth description about how the rules are generated.

III. Experiment Design

A. Selecting the Clients: Pharo Ecosystem

For this study, we select the ecosystem built around the Pharo open-source development community. Our analysis included six years of evolution (from 2008 to 2013) with 3,588 systems and 2,874 contributors. There are two factors influencing this choice. First, the ecosystem is concentrated on two repositories, SqueakSource and SmalltalkHub, which gives us a clear inclusion criterion. Second, we are interested in comparing our results with the previous work of Robbes *et al.* [2]; using the same ecosystem facilitates this comparison.

The Pharo ecosystem: Pharo is an open-source, Smalltalk-inspired, dynamically typed language and environment. It is currently used in many industrial and research projects.[4] The Pharo ecosystem has several important projects. For example, Seaside[5] is a web-development framework, a competitor for Ruby on Rails as the framework of choice for rapid web prototyping. Moose is an open-source platform for software and data analysis. Phratch, a visual and educational programming language, is a port of Scratch to the Pharo platform. Many other projects are developed in Pharo and hosted in the SqueakSource or SmalltalkHub repositories.

The SqueakSource and SmalltalkHub repositories: SqueakSource and SmalltalkHub repositories are the basis for the software ecosystem that the Pharo community have built over the years. They are the *de facto* platform for sharing open-source code for this community offering a nearly complete view of

[1]http://www.pharo.org, verified on 25/03/2015
[2]http://www.moosetechnology.org, verified on 25/03/2015
[3]See the mailing discussion in: http://goo.gl/Ul3Sha, verified on 25/03/2015
[4]http://consortium.pharo.org, verified on 25/03/2015
[5]http://www.seaside.st, verified on 25/03/2015

the Pharo software ecosystem. The SqueakSource repository is also partially used by the Squeak open-source development community. SmalltalkHub was created after SqueakSource by the Pharo community to be a more scalable and stable repository. As a consequence, many Pharo projects migrated from SqueakSource to SmalltalkHub, and nowadays, new Pharo projects are concentrated in SmalltalkHub.

Transition between SqueakSource and SmalltalkHub: We detected that 211 projects migrated from SqueakSource to SmalltalkHub while keeping the same name and copying the full source code history. We count these projects only once: we only kept the projects hosted in SmalltalkHub, which hosts the version under development and the full code history.

In theory, the migration was done automatically by a script provided by SmalltalkHub developers, thus keeping the meta-data such as project name. However, to increase our confidence in the data, we calculated the Levenshtein distance between the projects in each repository to detect cases of similar but not equal project names. We detected that 93 systems had similar names (*i.e.,* Levenshtein distance = 1). By manually analyzing each of these systems, we detected that most of them are in fact distinct projects, *e.g.,* "AST" (from abstract syntax tree) and "rST" (from remote smalltalk). However, 14 systems are the same project with a slightly different name, *e.g.,* "Keymapping" in SqueakSource was renamed to "Keymappings" in SmalltalkHub. In these cases, again, we only kept the projects hosted in SmalltalkHub, as they represent the version under development and include the full source code history.

B. Selecting the Frameworks: Pharo Core

Pharo core frameworks: The frameworks from which we applied associations rules mining to extract the API changes come from the Pharo core. They provide a set of APIs, including collections, files, sockets, unit tests, streams, exceptions, graphical interfaces, etc. (they are Pharo's equivalent to Java's JDK). Such frameworks are available to be used by any system in the ecosystem.

We took into account all the versions of Pharo core since its initial release, *i.e.,* versions 1.0, 1.4, 2.0, and 3.0. Table I shows the number of classes and lines of code in each version. The major development effort between versions 1.0 and 1.4 was focused on removing outdated code that came from Squeak, the Smalltalk dialect Pharo is a fork of, explaining the drop in number of classes and lines of code.

TABLE I
PHARO CORE VERSIONS SIZE.

Version	1.0	1.4	2.0	3.0
Classes	3,378	3,038	3,345	4,268
KLOC	447	358	408	483

Generating a list of API changes: We adopted our previous approach [7], described in Section II, to generate a list of API changes. We set out to produce rules with a minimum support of 5, and a minimum confidence of 50%. The minimum support at 5 states that a rule has a relevant amount of occurrences in the framework, and the minimum confidence at 50% yields a good level of confidence (as an example, Schäfer *et al.* [12] use a confidence of 33% in their approach to detect evolution rules). Moreover, the thresholds reduce the number of rules to be manually analyzed.

This process produced 344 rules that were manually analyzed with the support of documentation and code examples to filter out incorrect or noisy ones. For example, the rule SortedCollection.new() → OrderedCollection.new() (*i.e.,* Java's equivalent to SortedSet and List, respectively) came out from a specific refactoring on a specific framework but clearly we cannot generalize this change for clients, so this rule was discarded. This filtering produced 148 rules.

Filtering the list of API changes by removing deprecation: Naturally, some of the API changes inferred by our approach are related to API deprecation. As such cases were studied by Robbes *et al.* [2], they are out of the scope of this paper. For this purpose, we first extracted all methods marked as deprecated found in the analyzed evolution of Pharo core; this produced 1,015 API deprecation. By discarding the API changes related to API deprecation, *our final list includes 118 API changes*.

From these API changes, 59 are about method suggestion (*i.e.,* both methods are available to be used by the client; *cf.* Section II) and 59 are about method replacement (*i.e.,* the old method is removed, so it is not available to be used). Furthermore, 10 out of the 118 API changes involved the evolution of internal APIs of the frameworks which, in theory, should not affect client systems. By internal API, we mean a public component that should only be used internally by the framework, *i.e.,* not by client systems. For instance, in Eclipse, the packages named with *internal* include public classes that is not part of the API provided to the clients [15], [16].

In Table II, we present some examples of API changes. The first API change improves code legibility, as it replaces two method calls by a single, clearer one. The second example replaces a method with a more robust one, that allows one to provide a different behavior when the intersection is empty. The third is an usage convention: Pharo advises not to use Object.log() methods, to avoid problems with the *log* function. Finally, the fourth one represents a class and method replacement due to a large refactoring: ClassOrganizer.default() does not exist anymore; ideally, it should have been marked as deprecated.

TABLE II
EXAMPLE OF API CHANGES.

id	API change (old-call → new-call)
1	ProtoObject.isNil() *and* Boolean.ifTrue(*) → ProtoObject.ifNil(*)
2	Rectangle.intersect(*) → Rectangle.intersectIfNone(*,*)
3	Object.logCr(*) → Object.traceCr(*)
4	ClassOrganizer.default() → Protocol.unclassified()

Assessing reactions of API changes in the ecosystem: When analyzing the reaction of the API changes in the ecosystem, we do not consider the frameworks from which we discovered the API changes, but only the client systems hosted at SqueakSource and SmalltalkHub, as described in Subsection III-A. To verify a reaction to API change in these systems, we need to detect when the change was available. We consider that an API change is available to client systems from the moment it was discovered in the framework. All commits in the client systems after this moment that remove a method call from the old API and add a call to the new API are considered to be reactions to the API change.

Notice that, in this study, we assess commits in the ecosystem that applied the prescribed API change (*i.e.,* the removals and additions of method call according to the rule we inferred). In the API deprecation study [2], the authors were primarily interested in the removals of calls to deprecated methods, but did not consider their replacement.

IV. Results

A. Magnitude of Change Propagation

RQ1. How many systems react to the API changes in an ecosystem and how many developers are involved?

1) Results: In this research question we verify the frequency of reactions and we quantify them in number of systems, methods, and developers.

Frequency of reactions: From the 118 API changes, 62 (53%) caused reactions in at least one system in the ecosystem. Moreover, from these API changes, 5 are internal, meaning client developers also use internal parts of frameworks to access functionalities not available in the public interfaces. We see in the next research questions that many systems take time to react to API changes. Hence, some API changes may not have been applied by all systems yet.

These reactions involved 178 (5%) client systems and 134 (4.7%) distinct developers. We show the distribution of such data (*i.e.,* the API changes that caused change propagation) in the box plots shown in Figure 1.

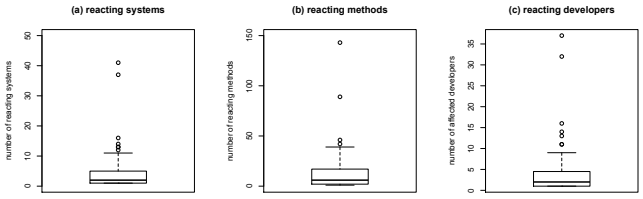

Fig. 1. Box plots for (a) systems, (b) methods, and (c) developers reacting to API changes.

Reacting systems: Figure 1a shows the distribution of reacting systems: the 1st quartile is 1 (bottom of the box), the median is 2 (middle of the box), the 3rd quartile is 5 (*i.e.,* 25% of the API changes cause reactions in 5 or more systems, forming the top of the box in the box plot), and the maximum is 11 (*i.e.,* it marks the highest number of reacting systems that is not considered an outlier, forming the top whisker of the box). The API change isNil().ifTrue(*) → ifNil(*) caused the largest reaction, 41 systems; this change is depicted as the dot at the top of the box plot in Figure 1a (in a box plot all outliers are shown as dots).

Reacting methods: For methods (Figure 1b), the 1st quartile is 2, the median is 6, the 3rd quartile is 17, and the maximum is 39. These results show that some systems reacted several times to the same API change: the median system reaction is 2 while the median method reaction is 6. For example, the API change isNil().ifTrue(*) → ifNil(*) caused reaction in 41 systems, but 89 methods.

Reacting developers: The number of developers impacted by API changes is shown in Figure 1c, as the number of commit authors that react to the API changes. In this case, the 1st quartile is 1, the median is 2, the 3rd quartile is 5, and the maximum is 11. The median at 2 shows that many change propagations involve few developers while the 3rd quartile at 5 shows that some of them involve several developers. The API change isNil().ifTrue(*) → ifNil(*), for example, involved a large number of developers (37). Overall, the distribution of the number of developers involved in the change is similar to the number of systems, implying that it is common that only one developer from a given system reacts to the API changes.

2) Comparison with API deprecation: Our magnitude results are different when we compare to explicit API deprecation. In the previous study there was a higher level of reaction to API changes. In the present study, 62 API changes caused reactions while in the API deprecation case, 93 deprecated entities caused reactions, *i.e.,* 50% more. The median of reactions in our study is 2, whereas it is 5 in the case of API deprecation. This is expected, since deprecated methods produce warning messages to developers while in the case of API changes no warning is produced.

Another difference relies on the number of developers involved in the reaction. In our study, it is common that *one* developer reacts to the API changes while in the API deprecation study it is more common that *several* developers of the same system react. One possible explanation is again that changes involving deprecated methods are usually accompanied by warnings, thus they can be performed by any client developer. In contrast, the API changes evaluated in this work can only be performed by developers that previously know them. This confirms that reacting to an API change is not trivial, thus, sharing this information among developers is important.

These results compared to the previous study reinforce the need to explicitly annotate API deprecation. More people gain knowledge of the changes and more systems/methods are adapted.

B. Duration of Change Propagation

RQ2. How long does it take for systems to react to API changes?

1) Results: A quick reaction to API changes is desirable for clients to benefit sooner from the new API. Next, we evaluate the reaction time of the ecosystem.

We calculate the reaction time to an API change as the number of days between its creation date (*i.e.,* the first time it was detected in the framework) and the first reaction in the ecosystem. As shown in Figure 2a, the minimum is 0 days, the 1st quartile is 5 days, the median is 34 days, the 3rd quartile is 110 days. The 1st quartile at 5 days shows that some API changes see a reaction in few days: this is possible if developers work both on frameworks and on client systems, or coordinate API evolution via mailing lists [17].

In contrast, the median at about 34 days and the 3rd quartile at 110 days indicate that some API changes take a long time to be applied. In fact, as Pharo is a dynamically typed language, some API changes will only appear for developers at runtime which can explain the long time frame.

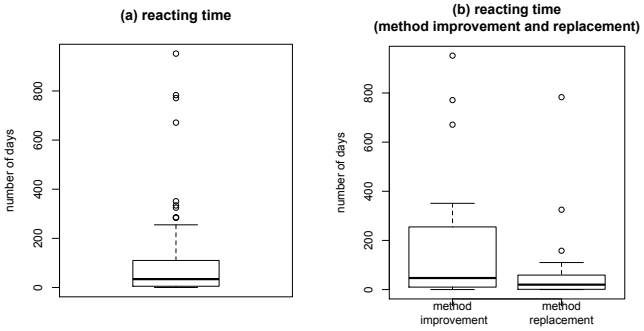

Fig. 2. Box plots for reaction time of (a) all API changes and (b) separated by method improvement and replacement, both in number of days.

In addition, we analyze the reaction time considering the two categories of API changes, method improvement and replacement, as shown in Figure 2b. For the API changes about improvement, the 1st quartile is 10 days, the median is 47 days, the 3rd quartile is 255 days, and the maximum is 351 days. In contrast, for the API changes about replacement, the 1st quartile is 1 days, the median is 20 days, the 3rd quartile is 59 days, and the maximum is 110 days.

Therefore, the reaction time for the API changes due to method improvements is longer than the ones about replacement, implying that the former is harder to be detected by client developers. This is explained by the fact that in the case of method improvements, the old method is still valid, so client developers are not forced to update their code. However, they would benefit if such API changes are suggested to them beforehand. In practice, many developers are simply not aware.

In summary, the results show that the reaction to API changes is not quick. Client developers need some time to discover and apply the changes; this time is longer for API changes related to method improvements.

2) Comparison with API deprecation: The reaction time of the API changes considered in our study is longer when we compare to the reaction time of API deprecation. In the API deprecation case, the 1st quartile is 0 days, the median is 14 days, and the 3rd quartile is 90 days (compared to 5, 34 and 110 days, respectively, in our API changes). Clearly, the reaction to deprecated APIs is faster than in the case of API changes. This is facilitated by the warning messages produced by deprecated methods.

If we compare method improvement in this study with method explicit deprecation in the previous study, we see it takes more than 3 times longer (47 days against 14) to react without explicit deprecation. To complement the result in section IV-A, explicit deprecation allows more developers to know of the change and more quickly.

C. Extension of Change Propagation

RQ3. Do all the systems in an ecosystem react to API changes?

1) Results: In the previous subsection we concluded that some systems take a long time to react to an API change. Here, we see that other systems do not react at all. To determine whether all systems react to the API changes, we investigate all the systems that are potentially affected by them, *i.e.,* that feature calls to the old API.

Table III shows that 2,188 (61%) client systems are potentially affected by the API changes, involving 1,579 (55%) distinct developers. Moreover, we detected that 112 API changes (out of 118), including the 10 internal API changes, potentially affected systems in the ecosystem. In the rest of this subsection, we analyze the distribution of such data.

TABLE III
EXTENSION OF CHANGE PROPAGATION.

Number of affected...		
Systems	**Methods**	**Developers**
2,188	107,549	1,579

Affected systems and methods: Figures 3a and 3b show the distribution of systems and methods affected by API changes in the ecosystem. We note that the number of affected systems and methods are much higher than those that actually react to API changes (as shown in Figure 1). The 1st quartile of affected systems is 15 compared to only 1 system reacting (methods: 59 compared to 2). The median of affected systems by an API change is 56.5 compared to only 2 systems reacting to it (methods: 253 compared to 2). The 3rd quartile of affected systems is 154.5 compared to 5 systems reacting (methods: 744.5 compared to 17).

Relative analysis: The relative analysis of reacting and affected systems produces a better overview of the impact. In that respect, comparing the ratio of reacting systems to the ratio of affected systems gives the distribution shown in Figure 4a. It shows that a very low number of systems react: the median is 0%, the 3rd quartile is 3%, the maximum is 7%. We investigate possible reasons for this low amount of reactions.

In an ecosystem, a possibly large amount of the systems may be stagnant, or even dead [2]. Thus, we first investigate the hypothesis in which systems that did not react either died before the change propagation started or were stagnant. A system is dead if there are no commits to its repository

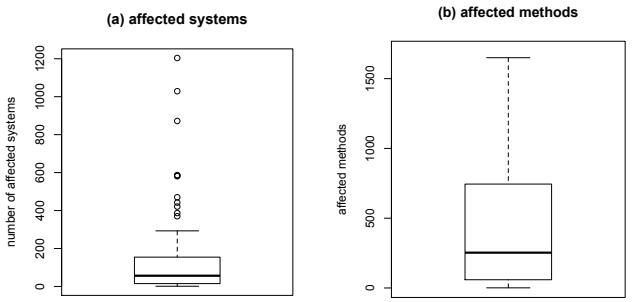

Fig. 3. Box plots for (a) systems and (b) methods affected by API changes.

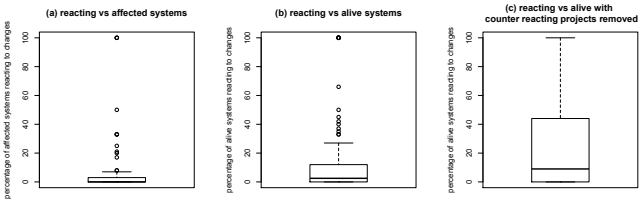

Fig. 4. Box plots for ratios of: (a) reacting affected systems; (b) reacting alive systems; and (c) reacting alive systems, removing counter reactions.

after the API change that triggered the change propagation. A system is stagnant if a minimal number of commits (less than 10) was performed after the API change. Thus, removing dead or stagnant systems (*i.e.,* keeping alive systems only) produces the distribution shown in Figure 4b: the median is 2.5%, the 3rd quartile is 12%, and the maximum is 27%.

A second reason why a system would not react to a change is when it is using another version of the framework, one in which the API did not change. This may occur when a system does not have the manpower to keep up-to-date with the evolution and freezes its relationship with a version that works [2]. To estimate this effect, we measure the number of systems that actually add more calls to the old API change, *i.e.,* they are counter reacting to the API evolution. Thus, removing these systems from the alive ones gives the distribution shown in Figure 4c: the median is 9%, the 3rd quartile is 44%, and the maximum is 100%. This new distribution reveals that many systems do not update to the new framework versions, even after filtering out dead, stagnant, and counter-reacting systems. As a result, the effort of migrating to newer versions becomes more expensive over time due to change accumulation.

2) Comparison with API deprecation: The presented reacting ratios are very different when compared to the API deprecation study. For the *ratio of reacting and affected systems*, the 1st quartile is 13%, the median is 20%, and the 3rd quartile is 31% in the API deprecation case (compared to 0%, 3% and 7%, respectively, in our API changes), which confirms the difference between both types of API evolution. These percentages increase in the other ratio comparisons. For the *ratio of reacting and alive without counter reacting systems*, the 1st quartile is 50%, the median is 66%, and the 3rd quartile is 75% for API deprecation (compared to 0%, 9% and 44%, respectively, in our API changes). Clearly, client systems react more to API deprecation. However, our results show that reactions to API changes are not irrelevant.

D. Consistency of Change Propagation

RQ4. Do systems react to an API change in the same way?

1) Results: The API changes analyzed in the previous research questions described the main way the analyzed frameworks evolved. However, some API changes may allow multiple replacements [2]. For example, Table IV shows three examples of API changes extracted from the analyzed frameworks, and their reactions by the ecosystem.

TABLE IV
EXAMPLES OF API CHANGES; THE NUMBERS SHOW THE CONFIDENCE OF THE REPLACEMENT IN THE ECOSYSTEM.

Old call (framework)	New call	
	Framework	Ecosystem
doSilently()	suspendAllWhile()	80% suspendAllWhile()
Pref.menuFont()	Fonts.menuFont()	40% Fonts.menuFont() 40% ECPref.menuFont()
HashAlgorithm.new()	SHA1.new()	63% HashFunction.new() 30% SHA1.new()

The first API change, doSilently() → suspendAllWhile(), is mostly followed by the ecosystem, presenting a confidence of 80% (*i.e.,* 80% of the commits that removed the old call also added the new call). For the second API change, Preferences.standardMenuFont() → StandardFonts.menuFont(), the ecosystem reacts with two possible replacements, both with confidence of 40%. For the third API change, SecureHashAlgorithm.new() → SHA1.new(), the ecosystem also reacts with two possible replacements: a main one with confidence of 63% and an alternative one with 30%.[6] Notice that, in this case, the main replacement is not the one extracted from the analyzed framework, *i.e.,* it is HashFunction.new() instead of SHA1.new().

To better understand such cases, we analyze the consistency of the API changes by verifying the reactions of the ecosystem. *Consistency of main and alternative replacements in the ecosystem:* Figure 5a presents the confidence distribution of the main and alternative replacements in the ecosystem. For the main replacement, the 1st quartile is 36%, the median is 60%, and the 3rd quartile is 100%. For the alternative replacement, the 1st quartile is 20%, the median is 25%, and the 3rd quartile is 31%. These results show that alternative replacements are found in the ecosystem (such as the second and third examples in Table IV), but with less confidence than the main ones. Thus, alternative replacements explain a minority of the cases where affected systems do not react to the prescribed API changes.

[6]Main and alternative replacements of API changes in the ecosystem are determined by verifying how the ecosystem replaces the old calls. This is done by applying our approach described in Section II in the ecosystem itself.

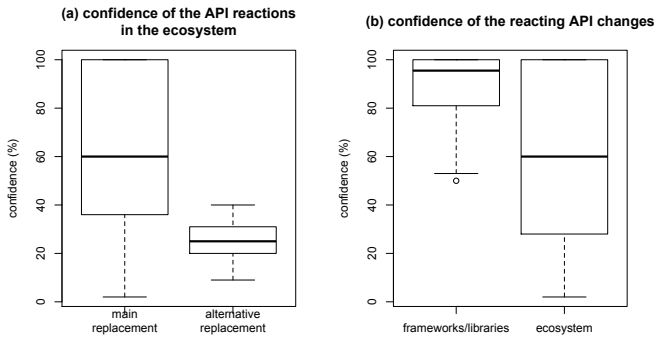

Fig. 5. Box plots for the confidence of (a) the reaction in the ecosystem (main and alternative replacements) and (b) the reacting API changes (frameworks/libraries and ecosystem).

Consistency of API changes in the frameworks and in the ecosystem: Figure 5b compares the confidence distribution of the 62 reacting API changes both in the analyzed frameworks and in the ecosystem. In the analyzed frameworks, the minimum is 53%, the 1st quartile is 81%, the median is 95%, and the 3rd quartile is 100% (recall that a minimum confidence of 50% was adopted to generate the API changes). In the ecosystem, for the *same* API changes, the minimum is 2%, the 1st quartile is 28%, the median is 60%, and the 3rd quartile is 100%.

There is a difference in the confidence: the API changes are more consistently followed by the frameworks than by the ecosystem. This suggests that many replacements are not resolved in a uniform manner in the ecosystem: client developers may adopt other replacements in addition to the prescribed ones (such as the second and third examples in Table IV); method calls may be simply dropped, so they disappear without replacements; and developers may replace the old call by local solutions. Thus, this result provides evidence that API changes can be more confidently extracted from frameworks than from clients (*i.e.,* the ecosystem).

2) Comparison with API deprecation: For the main replacement in the API deprecation study, the confidence of the 1st quartile is 46%, the median is 60%, and the 3rd quartile is 80% (compared to 36%, 60%, and 100%, respectively, in our study). Note that the distribution of the main replacement is mostly equivalent in both cases. In fact, in the case of API deprecation, it is common the adoption of alternative replacements and home-grown solutions due to empty warning messages.

V. Summary and Implications

In summary, our study shows that 53% (62 out of 118) of the analyzed API changes caused reaction in only 5% of the systems and affected 4.7% of the developers. Overall, the reaction time of API changes is not quick (median 34 days). Client developers, naturally, need some time to discover and apply the new API; this time is even longer in the case of method improvement. In contrast, a large amount of systems are potentially affected by the API changes: 61% of the systems and 55% of the developers. In fact, the number of affected systems are much higher than those that actually react to API changes. As a result, the effort of porting to newer versions becomes more expensive due to change accumulation.

The answers to our research questions allow us to formulate the following implications of our study.

Deprecation mechanisms should be more adopted: Half of the API changes analyzed in this work (59 out of 118) are about method replacements. It means that such API changes are probably missing to use deprecation mechanisms. Ideally, they should have been marked as deprecated by the framework developers. In fact, in large frameworks, developers may not know whether their code is used by clients: this may cause a growth [2] or a lack in the use of deprecation [1], [15].

In our study, this lack of deprecation was mainly due to large refactorings in the frameworks. For instance, the framework for dealing with files completely changed after Pharo 1.4. As a result, some APIs missed to be marked as deprecated; *e.g.,* in the Moose migration to Pharo 3.0, a developer noticed this issue and commented[7]: "In FileSystem, ensureDirectory() was renamed to ensureCreateDirectory() without a deprecation", the framework developer then answered: "Fill up a bug entry and we will add this deprecation. Good catch". In fact, for such cases, asking in Question and Answer sites[8] or mailing lists[9] is the current alternative for client developers.

Based on these results we conclude the following:

> Many deprecation opportunities are missed by the developers (we found at least 59 instances in our study). Recommenders can be built to remind API developers about these missed opportunities.

Client developers use internal parts of frameworks: Internal APIs are unstable and unsupported interfaces [16], so they should not be used by clients. However, all the internal APIs (*i.e.,* 10 cases) analyzed in this work are used by clients. From such, 5 caused the clients to react as in the frameworks. Thus, our results reinforce (at large-scale and ecosystem level) previous studies [11], [18], [16], showing that client systems use internal parts of frameworks to access functionalities not available in the public interfaces for a variety of reasons.

Based on these results we conclude the following:

> Internal APIs are sometimes used by client developers in the ecosystem under analysis. Recommenders can be build to help API developers identify often used internal APIs; those are candidates to be public APIs to keep clients using stable and supported interfaces.

Replacements are not resolved in a uniform manner: Many replacements are not resolved uniformly in the ecosystem. Clients may adopt other replacements in addition to the

[7]http://forum.world.st/moving-moose-to-pharo-3-0-td4718927.html, verified on 25/03/2015

[8]Coordination via Question and Answer sites: http://stackoverflow.com/questions/15757529/porting-code-to-pharo-2-0, verified on 25/03/2015

[9]Coordination via mailing lists: http://goo.gl/50q2yZ, http://goo.gl/k9Fl0K, http://goo.gl/SkMORX, verified on 25/03/2015

prescribed ones; method calls may be simply dropped; and developers may replace the old call by local solutions. In this context, some studies propose the extraction of API changes from frameworks (*e.g.,* [11]) other propose the extraction from clients (*e.g.,* [12]).

Based on these results we conclude the following:

> There is no clear agreement on the best extraction source to detect API changes: frameworks or client systems. This study reinforces frameworks as a more reliable source.

Reactions to API changes can be partially automated: As we observed, many systems do not react to the API changes because they are not aware. Moreover, in the case of large client systems, the adaptation may take a long time and is costly if done manually.

Based on these results we conclude the following:

> Most of the API changes that we found in this work can be implemented as rules in static analysis tools such as FindBugs [19], PMD [20], and SmallLint [21]. These rules could help client developers to keep their source code up-to-date with the new APIs.

VI. Threats to Validity

Construct Validity: The construct validity is related to whether the measurement in the study reflects real-world situations. In our study, the main threat is the quality of the data we analyze and the degree of involved manual analysis.

Software ecosystems present some instances of duplication (around 15% of the code [22]), where packages are copied from a repository to another (*e.g.,* a developer keeping a copy of a specific framework version). This may overestimate the number of systems reacting to an API change.

Smalltalk (and Pharo) is a dynamically typed language, so the detection of API change reaction may introduce noise as systems may use unrelated methods with the same name. This means that an API change that uses a common method name makes change propagation hard to be detected. This threat is alleviated by our manual filtering of noisy API changes.

Another factor that alleviates this threat is our focus on specific evolution rules (*i.e.,* a specific replacement of one or more calls by one or more calls). For the first three research questions, we include only commits that are removing an old API *and* adding a new API to detect an API reaction. Requiring these two conditions to be achieved, decreases—or in some cases eliminates—the possibility of noise. For the fourth research question, we require the presence of the methods that contain a call to the old API. In this case, the noise could have been an issue, however, this threat is reduced since we discarded the API changes involved with common methods, *i.e.,* the noisy ones.

We also identify two threats regarding the comparison with the API deprecation study [2]. First, the time interval studied is not the same one: we analyzed the ecosystem evolution in the period from 2008 to 2013 while the API deprecation study analyzed from 2004 to 2011. Second, the way API changes are selected is different: while we deprecation study simply collected the list of API deprecation, we inferred the API changes from commits in source code repository; these API changes were manually validated by the authors of the paper with the support of documentation and code examples to eliminate incorrect and noisy ones. For these reasons, we can not claim that this is an exact comparison. Parts of the differences observed may be due to other factors.

Internal Validity: The internal validity is related to uncontrolled aspects that may affect the experimental results. In our study, the main threat is the possible errors in the implementation of our approach.

Our tool to detect API changes has been (i) used by several members of our laboratory to support their own research on frameworks evolution, and (ii) divulged in the Moose reengineering mailing list, so that developers of this community can use it; thus, we believe that these tasks reduce the risks of this threat.

External Validity: The external validity is related to the possibility to generalize our results. In our study, the main threat is the representativeness of our case studies.

We performed the study on a single ecosystem. It needs to be replicated on other ecosystems in other languages to characterize the phenomenon of change propagation more broadly. Our results are limited to a single community in the context of open-source; closed-source ecosystems, due to differences in the internal processes, may present different characteristics. However, our study detected API change reactions in thousands of client systems, which makes our results more robust.

The Pharo ecosystem is a Smalltalk ecosystem, a dynamically typed programming language. Ecosystems in a statically typed programming language may present differences. In particular, we expect static type checking to reduce the problem of noisy API changes for such ecosystems.

As an alternative to our choice of ecosystem, we could have selected a development community based on a more popular language such as Java or C++. However, this would have presented several disadvantages. First, deciding which systems to include or exclude would have been much more challenging. Second, the potentially very large size of the ecosystem could prove impractical. We consider the size of the Pharo ecosystem as a "sweet spot": with about 3,600 distinct systems and more than 2,800 contributors, it is large enough to be relevant.

VII. Related Work

A. Software Ecosystems Analysis

Software ecosystem is an overloaded term, which has several meanings. There are two principal facets: the first one focuses on the business aspect [23], [24], and the second on the artefact analysis aspect, *i.e.,* on the analysis of multiple, evolving software systems [25], [9], [2]. In this work we use the latter one; we consider an ecosystem to be "*a collection of software projects which are developed and co-evolve in the same environment*" [9]. These software systems have common underlying components, technology, and social norms [25].

Software ecosystems have been studied under a variety of aspects. Jergensen *et al.* [25] study the social aspect of ecosystems by focusing on how developers move between projects in the software ecosystems. The studies of Lungu *et al.* [26] and Bavota *et al.* [10] aim to recover dependencies between the software projects of an ecosystem to support impact analysis. Lungu *et al.* [27] focus on the software ecosystems analysis through interactive visualization and exploration of the systems and their dependencies. Gonzalez-Barahona *et al.* [28] study the Debian Linux distribution to measure its size, dependencies, and commonly used programming languages.

Recently, Mens *et al.* [29] proposed the investigation of similarities between software ecosystems and natural ecosystems found in ecology. In this context, they are studying the GNOME and the CRAN ecosystems to better understand how software ecosystems can benefit from biological ones. German *et al.* [30] also analyze the evolution of the CRAN ecosystem, investigating the growth of the ecosystem, and the differences between core and contributed packages.

In the context of API evolution and ecosystem impact analysis, McDonnell *et al.* [31] investigate API stability and adoption on a small-scale Android ecosystem. In such study, API changes are derived from Android documentation. They have found that Android APIs are evolving fast while client adoption is not catching up with the pace of API evolution. Our study does not rely on documentation but on source code changes to generate the list of APIs to answer different questions. Moreover, our study investigates the impact of API evolution on thousands of distinct systems.

In a large-scale study, Robbes *et al.* [2] investigate the impact of a specific type of API evolution, API deprecation, in an ecosystem that includes more than 2,600 projects; such ecosystem is the same that is used in our work. Our study considers API changes that were not marked as deprecated. Thus, there is no overlap between the changes investigated in our work and the ones investigated by that work. In fact, these studies complement each other to better characterize the phenomenon of change propagation at the ecosystem level.

B. API Evolution Analysis

Many approaches have been developed to support API evolution and reduce the efforts of client developers. Chow and Notkin [4] present an approach where the API developers annotate changed methods with replacement rules that will be used to update client systems. Henkel and Diwan [3] propose CatchUp!, a tool that uses an IDE to capture and replay refactorings related to the API evolution. Hora *et al.* [7], [8] present tools to keep track of API evolution and popularity.

Kim *et al.* [32] automatically infer rules from structural changes. The rules are computed from changes at or above the level of method signatures, *i.e.,* the body of the method is not analyzed. Kim *et al.* [33] propose a tool (LSDiff) to support computing differences between two system versions. In such study, the authors take into account the body of the method to infer rules, improving their previous work [32] where only method signatures were analyzed. Nguyen *et al.* [34] propose LibSync that uses graph-based techniques to help developers migrate from one framework version to another. Using the learned adaptation patterns, the tool recommends locations and update operations for adapting due to API evolution.

Dig and Johnson [15] help developers to better understand the requirements for migration tools. They found that 80% of the changes that break client systems are refactorings. Cossette *et al.* [35] found that, in some cases, API evolution is hard to handle and needs the assistance of an expert.

Some studies address the problem of discovering the mapping of APIs between different platforms that separately evolved. For example, Zhong *et al.* [36] target the mapping between Java and C# APIs while Gokhale *et al.* [37] present the mapping between JavaME and Android APIs.

VIII. Conclusion

This paper presented an empirical study about the impact of API evolution, in the specific case of methods unrelated to API deprecation. The study was done in the context of a large-scale software ecosystem, Pharo, with about 3,600 distinct systems. We analyzed 118 important API changes from frameworks, and we found that 53% impacted other systems. We reiterate the most interesting conclusions from our experiment results:

- API changes can have a large impact on the ecosystem in terms of client systems, methods, and developers. Client developers need some time to discover and apply the new API, and the majority of the systems do not react at all.
- API changes can not be marked as deprecated because framework developers are not aware of their use by clients. Moreover, client developers can use internal APIs to access functionalities not available in the public interfaces.
- Replacements can not be resolved in a uniform manner in the ecosystem. Thus, API changes can be more confidently extracted from frameworks than from clients.
- Most of the analyzed API changes can be implemented as rules in static analysis tools to reduce the adaptation time or the amount of projects that are not aware about a new/better API.
- API changes knowledge can be concentrated in a small amount of developers. API changes and deprecation can present different characteristics, for example, reaction to API changes is slower and less clients react.

As future work, we plan to extend this research to analyze ecosystems based on statically typed languages. Thus, the results presented in our study will enable us to compare reactions of statically and dynamically typed ecosystems to better characterize the phenomenon of change propagation.

Acknowledgment

This research was supported by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil), ANR (Agence Nationale de la Recherche - ANR-2010-BLAN-0219-01) and FAPEMIG.

REFERENCES

[1] W. Wu, Y.-G. Gueheneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *International Conference on Software Engineering*, 2010.

[2] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? The case of a smalltalk ecosystem," in *International Symposium on the Foundations of Software Engineering*, 2012.

[3] J. Henkel and A. Diwan, "Catchup!: Capturing and replaying refactorings to support API evolution," in *International Conference on Software Engineering*, 2005.

[4] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *International Conference on Software Maintenance*, 1996.

[5] A. Hora, N. Anquetil, S. Ducasse, and S. Allier, "Domain Specific Warnings: Are They Any Better?" in *International Conference on Software Maintenance*, 2012.

[6] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *International Conference on Software Engineering*, 2012.

[7] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "APIEvolutionMiner: Keeping API Evolution under Control," in *Software Evolution Week (European Conference on Software Maintenance and Working Conference on Reverse Engineering)*, 2014.

[8] apiwave, "Discover and track APIs," http://apiwave.com, 2015.

[9] M. Lungu, "Reverse Engineering Software Ecosystems," *PhD thesis, University of Lugano, Switzerland (October 2009)*, 2009.

[10] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: the case of Apache," in *International Conference on Software Maintenance*, 2013.

[11] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *International Conference on Software engineering*, 2008.

[12] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *International Conference on Software engineering*, 2008.

[13] A. Hora, N. Anquetil, S. Ducasse, and M. T. Valente, "Mining System Specific Rules from Change Patterns," in *Working Conference on Reverse Engineering*, 2013.

[14] M. Zaki and W. Meira Jr, "Fundamentals of data mining algorithms," 2012.

[15] D. Dig and R. Johnson, "The role of refactorings in API evolution," in *International Conference on Software Maintenance*, 2005.

[16] J. Businge, A. Serebrenik, and M. G. van den Brand, "Eclipse API usage: the good and the bad," *Software Quality Journal*, 2013.

[17] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz, "A Quantitative Analysis of Developer Information Needs in Software Ecosystems," in *European Conference on Software Architecture Workshops*, 2014.

[18] J. Boulanger and M. Robillard, "Managing concern interfaces," in *International Conference on Software Maintenance*, 2006.

[19] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," in *Object Oriented Programming Systems Languages and Applications*, 2004.

[20] T. Copeland, *PMD Applied*. Centennial Books, 2005.

[21] D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems*, vol. 3, 1997.

[22] N. Schwarz, M. Lungu, and R. Robbes, "On how often code is cloned across repositories," in *International Conference on Software Engineering*, 2012.

[23] D. G. Messerschmitt and C. Szyperski, "Software ecosystem: understanding an indispensable technology and industry," *MIT Press Books*, vol. 1, 2005.

[24] S. Jansen, S. Brinkkemper, and M. Cusumano, *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar Pub, 2013.

[25] C. Jergensen, A. Sarma, and P. Wagstrom, "The onion patch: migration in open source ecosystems," in *European Conference on Foundations of Software Engineering*, 2011.

[26] M. Lungu, R. Robbes, and M. Lanza, "Recovering inter-project dependencies in software ecosystems," in *International Conference on Automated Software Engineering*, 2010.

[27] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes, "The small project observatory: Visualizing software ecosystems," *Science of Computer Programming*, vol. 75, no. 4, 2010.

[28] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German, "Macro-level software evolution: a case study of a large software compilation," *Empirical Software Engineering*, vol. 14, no. 3, 2009.

[29] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik, "Studying evolving software ecosystems based on ecological models," in *Evolving Software Systems*, T. Mens, A. Serebrenik, and A. Cleve, Eds. Springer Berlin Heidelberg, 2014.

[30] D. M. German, B. Adams, and A. E. Hassan, "The evolution of the R software ecosystem," in *European Conference on Software Maintenance and Reengineering*, 2013.

[31] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *International Conference on Software Maintenance*, 2013.

[32] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *International Conference on Software Engineering*, 2007.

[33] M. Kim and D. Notkin, "Discovering and Representing Systematic Code Changes," in *International Conference on Software Engineering*, 2009.

[34] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[35] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *International Symposium on the Foundations of Software Engineering*, 2012.

[36] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API mapping for language migration," in *International Conference on Software Engineering*, 2010.

[37] A. Gokhale, V. Ganapathy, and Y. Padmanaban, "Inferring likely mappings between APIs," in *International Conference on Software Engineering*, 2013.