How do Developers React to API Evolution? a Large-Scale Empirical Study

André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, Stéphane Ducasse

Noname manuscript No.
(will be inserted by the editor)

How do Developers React to API Evolution? a Large-Scale Empirical Study

André Hora · Romain Robbes · Marco Tulio Valente · Nicolas Anquetil · Anne Etien · Stéphane Ducasse

Received: date / Accepted: date

Abstract Software engineering research now considers that no system is an island, but it is part of an ecosystem involving other systems, developers, and users. When a framework or a library evolves, its clients often must adapt. For example, client developers might need to adapt to functionalities, client systems might need to be adapted to a new API, and client users might need to adapt to a new user interface. The consequences of these 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 an exploratory study aimed at observing API evolution and its impact on a large software ecosystem, Pharo, which has about 3,600 distinct systems, and six years of evolution. We analyze 118 API changes in the context of method replacement and suggestion, 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.

A. Hora
FACOM, Federal University of Mato Grosso do Sul, Brazil
hora@facom.ufms.br
M. T. Valente
ASERG Group, Department of Computer Science (DCC), Federal University of Minas Gerais, Brazil
mtov@dcc.ufmg.br
R. Robbes
PLEIAD Lab, Department of Computer Science (DCC), University of Chile, Santiago, Chile
rrobbes@dcc.uchile.cl
N. Anquetil, A. Etien, S. Ducasse
RMoD Team, Inria, Lille, France
{nicolas.anquetil, anne.etien, stephane.ducasse}@inria.fr

1 Introduction

As any software system, frameworks evolve over time, and, consequently, client systems must adapt their source code to use the updated Application Programming Interface (API). To facilitate this time-consuming task, APIs should be backward-compatible and include deprecated API elements. In practice, researchers have found that frameworks are backward-incompatible (Wu *et al.*, 2010) and deprecation messages are often missing (Robbes *et al.*, 2012; Brito *et al.*, 2016). To support developers dealing with these problems, some approaches have been developed to help client developers, for example, with the support of specialized IDEs (Henkel and Diwan, 2005), the help of experts (Chow and Notkin, 1996), or the inference of change rules (Wu *et al.*, 2010; Hora *et al.*, 2012; Meng *et al.*, 2012; Hora *et al.*, 2014; Hora and Valente, 2015).

Frequently, 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 (Lungu, 2009). In this context, it is hard for developers to predict the real impact of API evolution. For example, API deprecation may affect hundreds of clients, with several of these clients staying in an inconsistent state for long periods of time or not reacting at all (Robbes *et al.*, 2012). Thus, the impact of API evolution may be large and sometimes unknown and managing API evolution is a complex task (Bavota *et al.*, 2013).

To support developers 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.* (2012), to verify the impact of deprecated APIs on a software ecosystem. Recently, Wu *et al.* (2016) analyzed the impact of type and method changes in the context of Apache and Eclipse ecosystem. However, API evolution is not restricted to deprecation nor type/method changes. It may imply, for example, a better API design that improves client code legibility, portability, performance, security, etc. The first aforementioned study analyzes the adoption of a specific group of changes, methods annotated as deprecated, but it introduces a bias as developers will probably notice more readily changes documented and checked by the compiler than changes not advertised. The second study focuses on fine-grained API changes such as method parameter removal and loss of visibility. Thus, there is still space for analyzing the adoption of more generic API changes, *i.e.,* not explicitly marked as deprecated and not mandatory (but recommended) to be applied by clients.

In this paper, we analyze the impact of API changes in the context of method replacement and suggestion on a large set of 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. Thus, we investigate the following research questions to support our study:

- Magnitude. RQ1: How many systems react to API changes in an ecosystem?
- Extension. RQ2: How many systems are affected by API changes?
- Duration. RQ3: How long does it take for systems to react and propagate API changes?

– Consistency. RQ4: Do systems react to API changes uniformly? and RQ5: How followed are API changes by the ecosystem?

In this study, we cover the Pharo[1] software ecosystem, which has about 3,600 distinct systems, and six years of evolution. We extracted 118 (out of 344) API changes from Pharo Core framework in the context of method replacement (*i.e.,* method a() must be replaced by b()) and suggestion (*i.e.,* a() should be replaced by b()). We then investigated the impact of these changes in the ecosystem, which may happen at compile or runtime; in fact, as Pharo is a dynamically typed language, some API changes will only impact the ecosystem at runtime. Moreover, we investigated the influence of the time variable on the analysis and contrast our results with the study by Robbes *et al.* (2012) on API deprecation to better understand how these two types of API evolution affect client systems.

This work is an extension of our previous study (Hora *et al.*, 2015b). Specifically, this study extends the previous one in five points: (1) we provide a new research question to study API change consistency (RQ5); (2) we provide new observations on RQ2 to investigate the transition between two software repositories where our case studies are stored; (3) we provide new data analysis on RQ3 to address API change propagation time; (4) we extend all research questions to contrast the results provided by the two analyzed API change types (*i.e.,* method replacement and suggestion); finally, (5) we extend all research questions with new data analysis to investigate the influence of the time variable on the analysis.

Therefore, 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 clients are impacted by API changes not marked as deprecated.
2. We provide an analysis on the reactions to two distinct types of API changes (method replacement and suggestion).
3. We provide a comparison between our results and the ones of the previous API deprecation study (Robbes *et al.*, 2012).

In Section 2, we present the API changes considered in this study. We describe our study design in Section 3. We present and discuss the observational results in Section 4. We present the implications of our study in Section 5 and the threats to the validity in Section 6. Finally, we present related work in Section 7 and we conclude the paper in Section 8.

2 API Changes

2.1 Definition

In this work, we focus on API changes related to method replacements and suggestions, following the line studied by several researches in the context of framework migration (Dagenais and Robillard, 2008; Schäfer *et al.*, 2008; Wu *et al.*, 2010; Hora *et al.*, 2012; Meng *et al.*, 2012; Hora *et al.*, 2014). In the following, we define and provide examples on the two types of API changes considered in this paper.

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

Method replacement: In this type of API change, one or more methods in the old release of a framework 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. In another case, in a one-to-many mapping, the method CutCommand(DrawingView) was replaced by CutCommand(Alignment, DrawingEditor) *and* UndoableCommand(Command) (Wu *et al.*, 2010). In both examples, the removed methods have not been deprecated; they were simply dropped, which may cause clients to fail at compile-time and at run-time if they update to the new release of the framework.

Method suggestion: In this type of API change, one (or more) method in the old release of the framework 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 (Hora *et al.*, 2015a), 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 recommended to improve code maintenance. In the Moose platform[2], a convention states that calls to MooseModel.root() and MooseModel.add() should be replaced by MooseModel.install() when adding models. In fact, MooseModel.root() and MooseModel.add() are internal methods, so they should be avoided by clients. Again, all the methods are available, but MooseModel.install() is recommended to avoid dependence on internal elements and 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 (Dagenais and Robillard, 2008; Schäfer *et al.*, 2008; Wu *et al.*, 2010; Meng *et al.*, 2012; Hora *et al.*, 2014, 2015a). In this study, we adopt our previous approach (Hora *et al.*, 2014) to detect API changes, which covers both aforementioned cases.

2.2 Detecting API Changes

In our approach, API changes are automatically produced by applying the *association rules* data mining technique (Zaki and Meira Jr, 2012) on the set of method calls that changed between two revisions 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 select rules that are more likely to represent relevant API changes. For example, rules can be ranked by confidence, support, or occurrences in different revisions.

More specifically, our approach is composed of two major steps. In the first step, we extract deltas from revisions in system history. Delta is a set of deleted and added calls of a method representing the differences between two revisions. Such step is

[2] `http://www.moosetechnology.org`

[3] See the mailing discussion in: `http://goo.gl/Ul3Sha`

done once for a system history under analysis. The second step is part of the rule discovering process. Rules are computed from the extracted deltas and indicate how calls should be replaced. We then apply *association rules* to the selected deltas to discover evolution rules. Please, refer to our previous study (Hora *et al.*, 2014) for an in-depth description about how the rules are generated.

3 Study Design

3.1 Selecting the Clients: Pharo Ecosystem

For this study, we select the ecosystem built by the Pharo open-source development community. Our analysis includes six years of evolution (from 2008 to 2013) with 3,588 client systems and 2,874 contributors. There are two factors influencing this choice. First, the ecosystem is concentrated in two repositories, SqueakSource and SmalltalkHub, which give us a clear inclusion criterion. Second, we are interested in comparing our results with the previous work by Robbes *et al.* (2012) and 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 several industrial and research projects[4]. The Pharo ecosystem has several important client systems. Pharo client systems are the ones using APIs provided by the Pharo Core framework (described in the next subsection). Comparing to the Eclipse ecosystem, Pharo client systems would be like plugins, *i.e.,* they are built on top of Pharo and use Pharo APIs. For example, Seaside[5] is a Web-development framework, in the spirit of Ruby on Rails, for rapid Web prototyping. Moose is an open-source platform for software and data analysis. Phratch[6], a visual and educational programming language, is a port of Scratch[7] to the Pharo platform. Many other systems (3,588 in total) 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 ecosystem that the Pharo community has built over the years. They are the *de facto* official repositories for sharing open-source code for this community, offering a complete view of the Pharo 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 improve availability, performance, and confidence. As a consequence, many Pharo projects migrated from SqueakSource to SmalltalkHub and, nowadays, new Pharo systems are released in SmalltalkHub.

Transition between SqueakSource and SmalltalkHub: We found that 211 client systems 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

[4] `http://consortium.pharo.org`
[5] `http://www.seaside.st`
[6] `http://www.phratch.com`
[7] `http://scratch.mit.edu`

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 system names in each repository to detect cases of similar but not equal project names. We detected that 93 systems had very 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.

3.2 Selecting the Frameworks: Pharo Core

Pharo Core frameworks: The frameworks on which we applied associations rules mining to extract the API changes come from the Pharo Core. These frameworks provide a set of APIs, including collections, files, sockets, unit tests, streams, exceptions, graphical interfaces. In fact, they are Pharo's equivalent to Java's JDK, which provides several APIs to handle collections, files, etc. Pharo Core also provides an IDE (compared to Eclipse), which is available to be used by client systems in the ecosystem, where clients can build applications on top.

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. We analyzed all the changes between these four releases, because our mining process work at revision level. Table 1 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 of which Pharo is a fork, explaining the drop in number of classes and lines of code.

Table 1 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 (Hora *et al.*, 2014), described in Section 2, to generate a list of API changes by mining each consecutive revisions. 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 revisions and the minimum confidence at 50% yields a good level of confidence. For comparison, Schäfer *et al.* (2008) use a

confidence of 33% in their approach to detect evolution rules. These minimum values reduce the number of rules to be manually analyzed.

Our approach produced 344 rules that were manually analyzed with the support of documentation and code examples to filter out incorrect ones. For example, the rule SortedCollection.new() → OrderedCollection.new() (Java's equivalent to SortedSet and List) came out from a specific refactoring 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.* (2012), 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*. Figure 1 shows the confidence distribution of the selected API changes: the 1st quartile is 86% while the median and the 3rd quartile is 100%. Thus, the selected API changes have high confidence and are strongly adopted in Pharo Core.

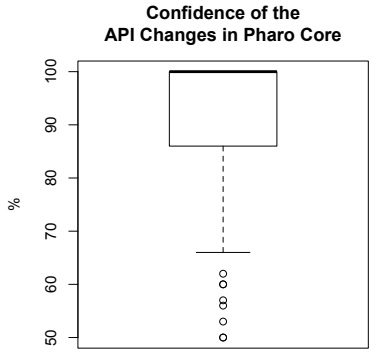

Fig. 1 Box plot for the confidence of the selected API changes in Pharo Core.

From these API changes, 59 are *method suggestions* (*i.e.,* both methods are available to the clients; *cf.* Section 2) and 59 are *method replacements* (*i.e.,* the old method is removed, so it is not available). Furthermore, 10 out of the 118 API changes involved the evolution of internal APIs of the frameworks. While public APIs are stable, supported, and documented (*i.e.,* they can be safely used by clients), internal APIs are unstable and unsupported (*i.e.,* they should not be used by clients because they are backward incompatible). For example, in Eclipse (and also in Pharo), internal APIs are implemented in packages with the word "internal", *e.g.,* org.eclipse.jdt.internal.ui.JavaPlugin, while in the JDK, internal APIs are in packages with the prefix "sun", *e.g.,* sun.misc.Unsafe (Dig and Johnson, 2005; Businge *et al.*, 2013; Hora *et al.*, 2016).

In Table 2, 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 a usage convention: Pharo advises not to use Object.log()-related methods, to avoid problems with the math *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 2 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 Pharo client systems (*i.e.,* systems build on top Pharo and using Pharo APIs) hosted at SqueakSource and SmalltalkHub, as described in Subsection 3.1. To verify a reaction to API change in these systems, we detect when the change was available. We consider that an API change is available to client systems from the first revision time stamp it was discovered in the framework. All commits in the client systems after this time stamp 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.

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 that we inferred). In the API deprecation study (Robbes *et al.*, 2012), the authors were primarily interested in the removals of calls to deprecated methods, but did not consider their replacement. Thus, our notion of reaction to API change is more strict than the one adopted by the deprecation study.

3.3 Presenting the Results

Section 4 presents the results. Each research question is organized in three parts:

1. *Results.* It answers the research questions and discusses the results.
2. *Time-based results.* It analyzes the most important results taking into account the age of the API changes. To perform such analysis, we first sorted the list of API changes by their introduction date (*i.e.,* first revision time stamp it was discovered in the framework). We then separated the sorted API changes in two groups: we classify the half older ones as *earlier API changes* and the half recent ones as *later API changes*. We want to verify whether the age of an API change

influences the results. For such comparison, we use the Mann-Whitney test for assessing whether one of two samples (earlier and later API changes, in our case) of independent observations tends to have larger values. This test is used when the distribution of the data is not normal and there is different participants (not matched) in each sample. As is customary, the tests will be performed at the 5% significance level. We also report *effect size* that indicates the magnitude of the effect and is independent of sample size. Effect size value between 0.1 and 0.3 is considered small, between 0.3 and 0.5 is medium, and greater than 0.5 is large.

3. *Comparison with API deprecation.* This part compares our results with the ones provided by Robbes *et al.* (2012) on API deprecation to better characterize the phenomenon of change propagation at the ecosystem level. The comparison is possible because, in most of the cases, the research questions are equivalent for both studies.

4 Observational Results

4.1 Magnitude of Change Propagation

RQ1. How many systems react to the API changes in an ecosystem?

Results

In this research question, we observe and compute 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 client system in the ecosystem while 56 (47%) did not cause any reaction. 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 APIs. We see in the next research questions that many systems take time to react to API changes. Thus, some API changes may not have been applied by all systems yet.

These reactions involved 178 (5%) distinct client systems and 134 (4.7%) developers. We present the distribution of such data (*i.e.,* the API changes that caused change propagation) in the box plots shown in Figure 2.

Reacting systems. Figure 2a shows the distribution of reacting systems per API change: 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). In addition, Figure 3 presents the reacting systems separated by API changes in the context of method suggestion and replacement. The median reaction is 3 for method suggestion against 2 for method replacement; the 3rd quartile is 6 against 4. In fact, as observed in the next research question, clients are more affected by method suggestion, thus, consequently, they are more likely to react to them, in absolute terms.

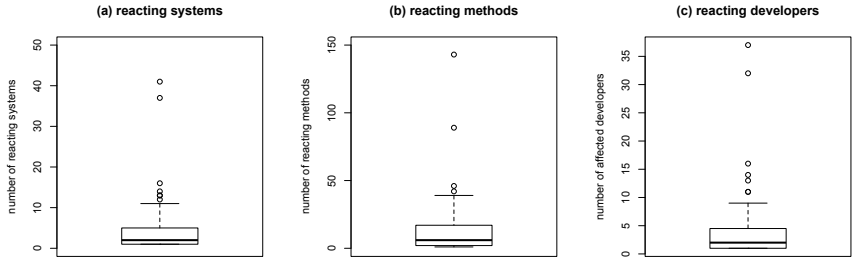

Fig. 2 Box plots for (a) systems, (b) methods, and (c) developers reacting to API changes.

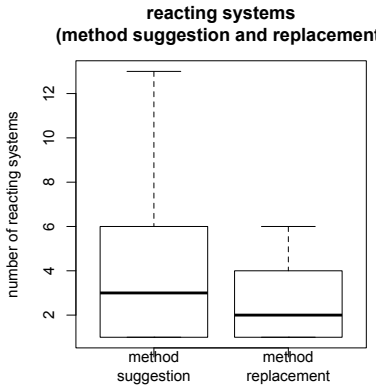

Fig. 3 Box plots for systems reacting to API changes separated by method suggestion and replacement.

Reacting methods. For methods (Figure 2b), the 1st quartile is 2, the median is 6, the 3rd quartile is 17, and the maximum is 39. These results show that for some systems several of their methods had to change to the same API change: the median system reaction is 2 while the median method reaction is 6.

Reacting developers. The number of developers impacted by API changes is shown in Figure 2c, 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. 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.

Time-based results

The number of reactions in the ecosystem may be also influenced by the age of the API changes. It is intuitively expected that, in total, an older API change has more reactions than a recent. In this context, we investigate whether earlier API changes are the ones with larger propagations.

Figure 4 shows the reacting systems with the API changes that caused change propagation separated in two groups: earlier changes and later changes. For the earlier changes, the median is 2, the 3rd quartile is 6.5, and the maximum is 13 whereas for the later changes, the median is 2, the 3rd quartile is 4, and the maximum is 8. Comparing both earlier and later gives a *p-value* > 0.05 and effect size $= 0.01$. While the median is equal for both groups, the 3rd quartile and maximum, as expected, show that earlier API changes have more reactions. Consequently, reactions to more recent API changes may be yet to come.

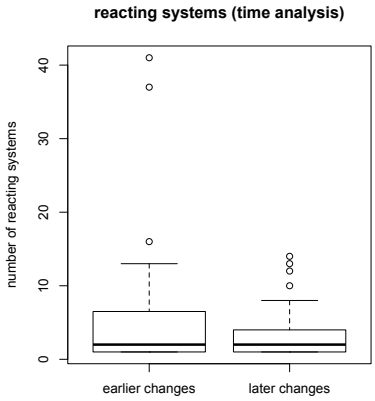

Fig. 4 Box plots for the reacting systems separated by earlier and later API changes.

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 difference is expected because 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 observation 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 and changes. More developers gain knowledge of the changes and more systems/methods are adapted.

4.2 Extension of Change Propagation

RQ2. How many systems are affected by API changes?

Results

In this research question, we investigate all client systems that are affected by API changes (*i.e.,* that feature calls to the old API) to better understand the extension of the API change impact: are the affected client systems in fact reacting to API changes?

Table 3 shows that 2,188 (61%) client systems are affected by the API changes, involving 1,579 (55%) distinct developers. From the 118 API changes, 112 API changes affected client systems, including the 10 internal ones. In the following, we analyze this distribution.

Table 3 Number of affected systems, methods, and developers.

Systems	Methods	Developers
2,188	107,549	1,579

Affected systems and methods. Figures 5a and 5b show the distribution of systems and methods affected by API changes in the ecosystem. The number of affected systems and methods are much higher than those that actually react to API changes (as shown in Figure 2 of RQ1). 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). In addition, we see in Figure 5c the distribution of systems affected by API changes in the context of method suggestion and replacement. In this case, the median is 96 for method suggestion against 21 for method replacement.

Relative analysis. The relative analysis of reacting and affected systems and methods provides a better view of the impact. In that respect, comparing the ratio of reacting systems over affected systems gives the distribution presented in Figure 6a, which

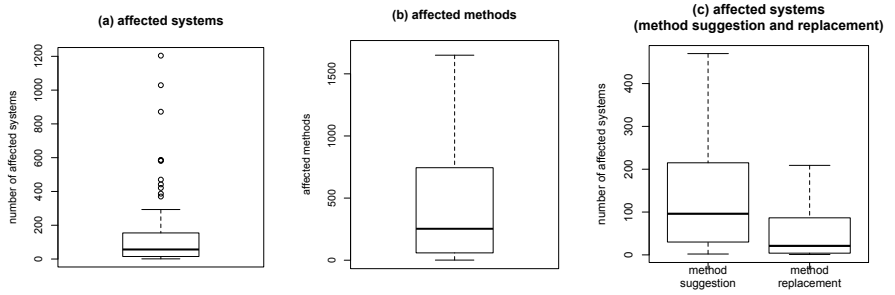

Fig. 5 Box plots for (a) systems, (b) methods affected by API changes as well as (c) systems separated by method replacement and suggestion.

shows that a low number of systems react: the median is 0%, the 3rd quartile is 3%, the maximum is 7%. From all affected client systems, only a minority react to API changes. However, some outliers are found: three API changes were completely applied by all clients. This happens because they affected few clients, for example, the method suggestion FSFilesystem inMemory() $\rightarrow$ FSFilesystem memory() only affected two clients and both reacted. Moreover, by performing the same analysis at method level (Figure 6b), we notice an even more extreme scenario: the median is 0%, and the 3rd quartile is only 1%. Again, some outliers are found; the top three points on the boxplot shows ratio of 100%, 85% and 60%, respectively. In addition, Figure 6c shows the ratio of reacting over affected systems in the context of method suggestion and replacement. In this case, the median is 0 for method suggestion against 1 for method replacement while the 3rd quartile is 2 against 4.5. We notice that both ratios are very low. As API changes about method suggestion are not mandatory to be applied, their ratio is even lower. In the following, we investigate possible reasons the low reaction.

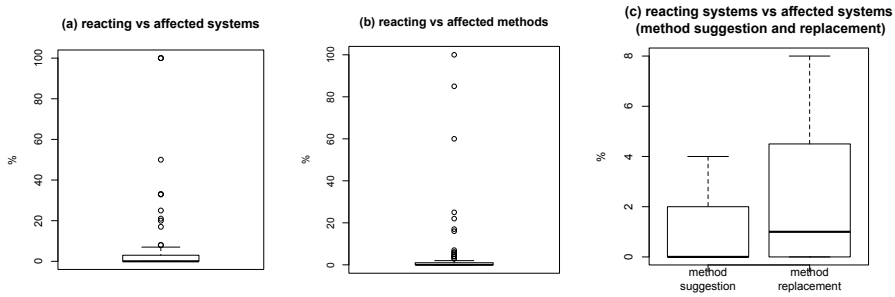

Fig. 6 Box plots for ratios of (a) reacting vs affected systems, (b) reacting vs affected methods, as well as (c) systems separated by method replacement and suggestion.

In an ecosystem, a possibly large amount of the systems may be stagnant or even dead (Robbes *et al.*, 2012). 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 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. Removing dead or stagnant systems (*i.e.,* keeping live systems only) produces the distribution shown in Figure 7b: the median is 2.5%, the 3rd quartile is 12%, and the maximum is 27%. Comparing with the previous analysis, we notice that the 3rd quartile increased from 3% to 12%, suggesting that in fact some systems may not react because they are stagnant or dead.

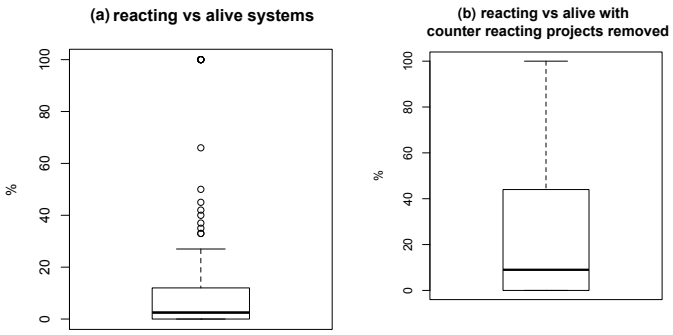

Fig. 7 Box plots for ratios of (a) reacting alive systems, and (b) reacting alive systems, removing counter reactions.

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. It 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 (Robbes *et al.*, 2012). 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. Removing these systems from the live ones gives the distribution shown in Figure 7c: the median is 9%, the 3rd quartile is 44%, and the maximum is 100%. This new distribution reveals that many systems may 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.

To further detect inactive systems, we analyze each repository, SqueakSource and SmaltalkHub, separately. SmalltalkHub repository was created by the Pharo community, so we expect that developers were more active in this repository in the recent years. In Figure 8, we report the ratio comparisons per repository. Firstly, in the ratio of reacting and affected systems, shown in Figure 8a, the median is 0%/0% (for SqueakSource/SmalltalkHub, respectively), the 3rd quartile is 1%/7%, and the maximum is 2%/16%. Secondly, in the ratio of reacting and live systems, shown in Figure 8b, the median is 0%/0%, the 3rd quartile is 7%/22%, and the maximum is

16%/50%. Finally, in the ratio of reacting and live systems without counter-reacting systems, shown in Figure 8c, the median is 0%/0%, the 3rd quartile is 22%/50%, and the maximum is 55%/100%. These results show that the community in SmalltalkHub is in fact more active, so reactions are more common in this repository.

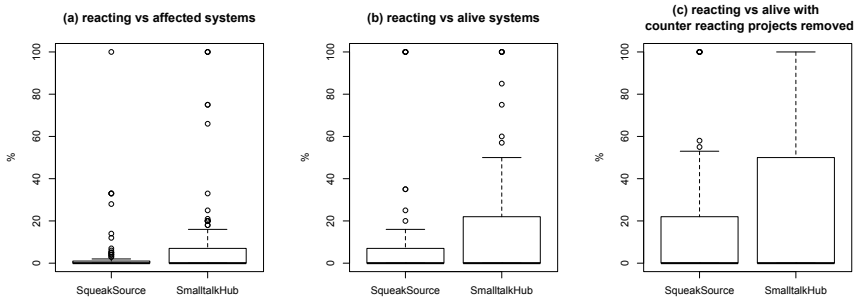

Fig. 8 Box plots, separated by repository, for ratios of (a) reacting affected systems, (b) reacting alive systems, and (c) reacting alive systems, removing counter reactions.

Time-based results

The age of the API changes may also influence the number of affected systems. We investigate whether earlier API changes affect more systems.

Figure 9 shows the number of affected systems separated in the groups earlier and later API changes. For the earlier changes, the 1st quartile is 18.5, the median is 55.5, and the 3rd quartile is 212 while for the later changes, the 1st quartile is 5.5, the median is 59, and the 3rd quartile is 130. Comparing both earlier and later give a *p-value* $= 0.14$ and effect size $= 0.13$. Earlier API changes affect slightly more systems. Even if the difference between both groups is small, the number of potentially affected client systems by the API changes tends to increase over time. Consequently, as time passes, it will be more complicated for these clients to migrate to new/better APIs.

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 live 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

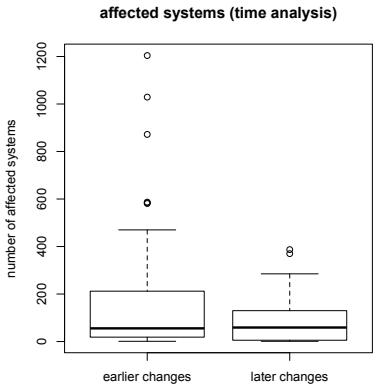

Fig. 9 Box plots for the affected systems separated by earlier and later API changes.

systems react more to API deprecation. However, our results show that reactions to API changes are not irrelevant.

4.3 Duration of Change Propagation

RQ3. How long does it take for systems to react and propagate API changes?

Results

A quick reaction to API changes is desirable for clients to benefit sooner from the new API. Moreover, systems should apply at once to an API change and not over a long period as they may be in an inconsistent state during that time. Next, we evaluate the reaction and propagation time of the ecosystem.

Reaction time. We calculate the reaction time to an API change as the number of days between its introduction date (*i.e.,* the first time stamp it was detected in the framework) and the first reaction in the ecosystem. As shown in Figure 10a, 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 (Haenni *et al.*, 2014).

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, thus in addition to compile time, some API changes will only appear for developers at runtime, which can explain the long reaction time frame.

In addition, we analyze the reaction time considering the two categories of API changes, method suggestion and replacement, as shown in Figure 10b. For the API changes about suggestion, 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

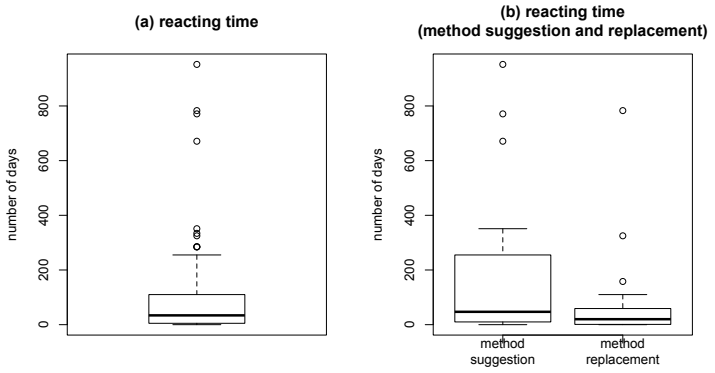

Fig. 10 Box plots for reaction time of (a) all API changes and (b) separated by method suggestion and replacement, both in number of days.

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

Propagation time. For a large system, apply a simple API change may not be trivial due to their source code size. Thus, we computed the propagation time for the changes on a per-system basis. We measured the interval between the first and the last reaction to the change propagation per system.

We observed that 86% of the propagation occur in 0 days, indicating that the majority of the systems fix an API change extremely quickly. It may occur with the help of refactoring tools found in current IDEs. Figure 11a shows the distribution for the other 14% of propagation: the 1st quartile is 12 days, the median is 71, the 3rd quartile is 284, and the maximum is 662 (we filter outliers for legibility). The median at 71 days shows that 50% of these systems take more than two months to apply certain API changes; some systems may even take years.

In Figure 11b, we analyze the propagation time considering the two categories of API changes. For the API changes about method suggestion, the 1st quartile is 7.5, the median is 121, the 3rd quartile is 334.5, and the maximum is 662. For the API changes about method replacement, the 1st quartile is 16, the median is 18, the 3rd quartile is 201, and the maximum is 211. Again, similarly to the reaction time, the adaptation time for the API changes about method suggestion is longer than the ones about replacement. It suggests that the former takes more time to be adopted in the same system by client developers.

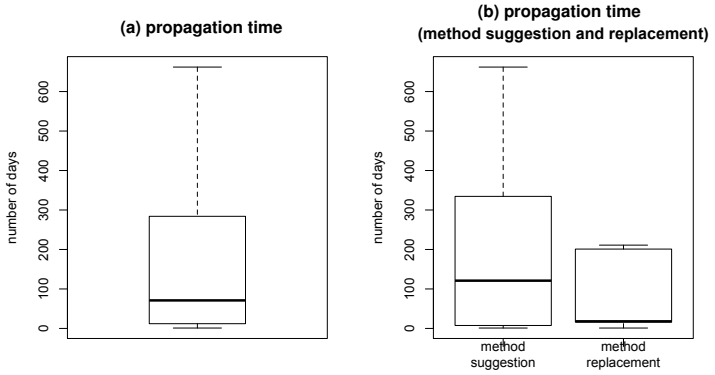

Fig. 11 Box plots for adaptation time of (a) all API changes and (b) separated by method suggestion and replacement, both in number of days.

In summary, the results show that the reaction time of API changes is not quick. Client developers, naturally, need some time to discover the new API change and apply them; this time is longer for API changes about method suggestion because Pharo is a dynamically typed language. In contrast, the propagation time of API changes in most of the systems occurs quickly. Still, some large systems may take a very long time to apply changes.

Time-based results

The age of the API changes may also influence the adaptation time. We investigate whether earlier API changes have a longer adaptation time, *i.e.,* more systems notice and react, making adaptation time longer.

Figure 12 shows the adaptation time (for the 14% of the adaptations that occur in more than 0 days) separated in the groups earlier and later API changes. For the earlier changes, the 1st quartile is 32, the median is 284, and the 3rd quartile is 454. For the later changes, the 1st quartile is 5, the median is 18, and the 3rd quartile is 133. Comparing both earlier and later give a *p-value* < 0.01 and effect size $= 0.39$. Thus, we confirm that earlier API changes have a longer adaptation time and that the variable time also plays an important role in the adaptation of a system.

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). If we only compare method suggestion and deprecation reactions (47 days against 14, on the median), we see that it takes more than 3 times longer to react to the former. Clearly, the reaction to deprecated APIs is faster than in the case of API changes. In fact, it is facilitated by

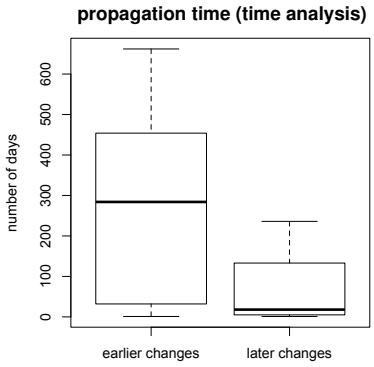

Fig. 12 Box plots for the adaptation time separated by earlier and later API changes.

warning messages produced by deprecated elements, which alert developers if they update the new framework release.

4.4 Consistency of Change Propagation

RQ4. Do systems react to API changes uniformly?

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 (Robbes *et al.*, 2012). For example, Table 4 shows three examples of API changes extracted from the analyzed frameworks and their reactions by the ecosystem.

Table 4 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()
Preferences.menuFont()	StandardFonts.menuFont()	40% Fonts.menuFont() 40% ECPref.menuFont()
SecHashAlgorithm.new()	SHA1.new()	63% HashFunction.new() 30% SHA1.new()

The first API change, doSilently() $\rightarrow$ suspendAllWhile(), is mostly followed by the ecosystem, presenting a confidence of 80% (*i.e.,* 80% of the commits that re-

moved 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, SecHashAlgorithm.new() → SHA1.new(), the ecosystem also reacts with two possible replacements: a main one with confidence of 63% and an alternative one with 30%[8]. In this replacement, the main one is not the same extracted from the analyzed framework, *i.e.,* it is HashFunction.new() instead of SHA1.new().

To better understand these changes, 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 13a 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 4), 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. In addition, we performed this analysis considering the two types of API changes: method suggestion and replacement. For the main replacement, the median is 42% for method suggestion against 91% for method replacement (for the alternative replacement: 22% against 30%). This result shows that API changes related to method suggestion present less confidence than the ones related to method replacement. In fact, with method suggestions, developers have more freedom to adopt other solutions, decreasing the confidence of these changes.

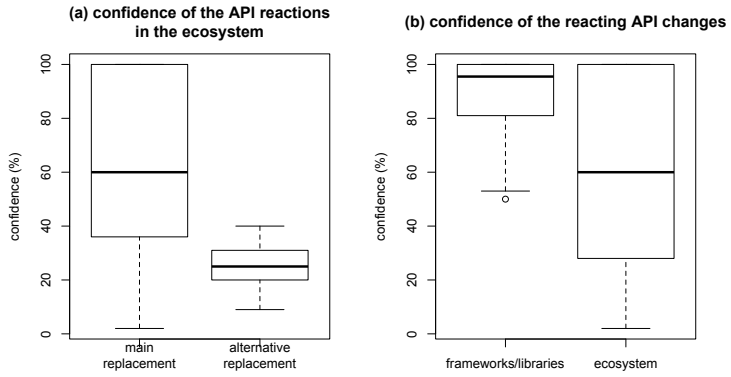

Fig. 13 Box plots for the confidence of (a) the reaction in the ecosystem (main and alternative replacements) and (b) the reacting API changes (frameworks/libraries and ecosystem).

[8] 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 2 in the ecosystem itself.

Consistency of API changes in the frameworks and in the ecosystem: Figure 13b compares the confidence distribution of the 62 reacting API changes both in the analyzed frameworks (*i.e.,* Pharo Core) 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. It suggests that many replacements are not resolved in a uniform manner in the ecosystem: client developers may adopt other replacements (such as the second and third examples in Table 4); 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).

Time-based results

The age of the API changes may also influence the consistency of reactions. We investigate whether earlier API changes are more heterogeneous in their reactions (more reactions, more opportunity to diverge). Figure 14 presents the distribution shown in Figure 13b separated by earlier and later API changes.

In the frameworks (Figure 14a), the median is 95% for the earlier changes and 100% for the later changes. Comparing both earlier and later give a *p-value* > 0.05 and effect size $= 0.06$. Even though the difference between the median is small, earlier API changes present overall less confidence, implying that their reactions are slightly more heterogeneous than the later ones.

In the ecosystem (Figure 14b), the difference between earlier and later API changes is clearer. In this case, for the earlier changes, the 1st quartile is 21%, the median is 35%, and the 3rd quartile is 60%. For the later changes, the 1st quartile is 61%, the median is 85%, and the 3rd quartile is 100%. Comparing both earlier and later give a *p-value* < 0.01 and effect size $= 0.46$. This observation confirms that, in the ecosystem, earlier API changes are more heterogeneous in their reactions. We can conclude that as old API changes produce more reactions over time (as shown in the time-analysis of RQ3), such reactions are more likely to diverge.

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). 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 (Robbes *et al.*, 2012).

Finally, we present the results of our last research question, which is also about the consistency of change propagation.

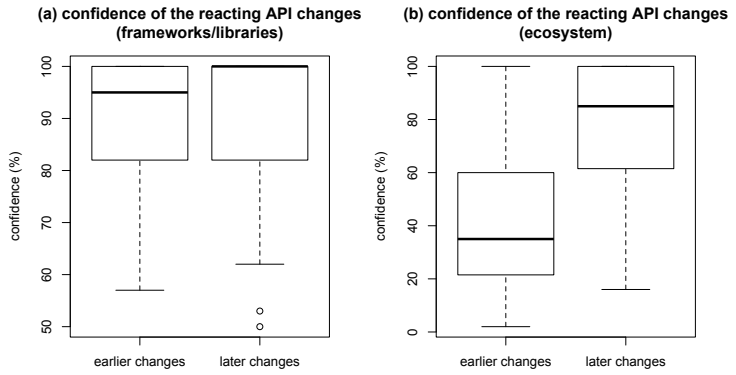

Fig. 14 Box plots for the confidence of earlier and later reacting API changes in the (a) frameworks and (b) ecosystem.

RQ5. How followed are the API changes by the ecosystem?

Results

In the previous research questions, we have seen that the ecosystem may adapt with other replacements instead the main one prescribed by the frameworks. Even if such cases happen in practice, ideally, a single replacement should be provided and adopted by clients. Next, we verify how followed are the API changes by the ecosystem, classifying them in three categories:

- *Rarely followed*: confidence is $\leq 10\%$.
- *Somewhat followed*: confidence is between 10% and 50%.
- *Mostly followed*: confidence is $\geq 50\%$.

Figure 15a shows the distribution of the classification for the 62 reacting API changes. The minority of the API changes, 4 (6%) are rarely followed, 21 (34%) are somewhat followed, and 37 (60%) are mostly followed; from such, 18 (29%) are totally followed with a confidence of 100%. In addition, we performed this analysis considering the two types of API changes (*i.e.,*, method suggestion and replacement). For the mostly followed classification, we have 45% for method suggestion against 76% for method replacement; for the somewhat followed classification, we have 45% against 21%; and for the rarely followed classification, we have 9% against 3%. In fact, as expected, method replacement provides more *followed classification* than method suggestion.

Figure 15b presents that only 13 (21%) API changes have multiple replacements in the ecosystem. Thus, this explains roughly half of the cases where the API changes are not consistently followed by the ecosystem (*i.e.,* the *rarely* and *somewhat* categories). The other half of the cases where the API changes are not consistently followed may happen due, for example, to the drop of method calls with no replacement or the use of local solutions by the client developers.

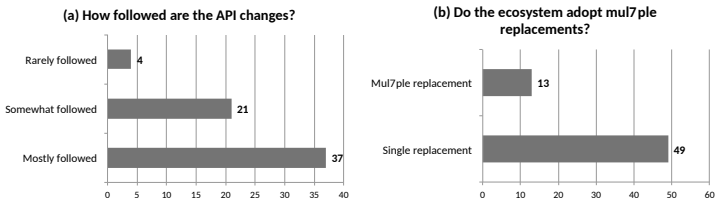

Fig. 15 (a) How client developers follow the API changes, and (b) number of single and multiple replacements in the ecosystem.

Time-based results

We investigate whether earlier API changes are more or less followed. Figure 16a shows that, for the earlier API changes, 4 (13%) are rarely followed, 16 (52%) are somewhat followed, and 11 (35%) are mostly followed. In contrast, for the later API changes, 0 are rarely followed, 5 (16%) are somewhat followed, and 26 (84%) are mostly followed. Figure 16b shows that 11 (35%) earlier API changes and 2 (6%) later API changes have multiple replacements.

In summary, earlier API changes are less followed than the later ones. Moreover, as API changes become old, other replacements are adopted by the ecosystem than the ones prescribed by the frameworks.

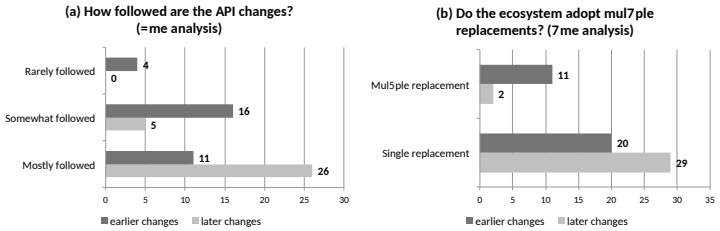

Fig. 16 Time analysis for (a) how client developers follow the API changes, and (b) number of single and multiple replacements in the ecosystem.

Comparison with API deprecation

Such observation was not performed by the API deprecation study.

5 Summary and Implications

In summary, our study shows that only 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 suggestion. In contrast, the propagation time to API changes in most of the systems occurs quickly.

Moreover, 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. The majority of the affected systems do not react neither in API changes nor in API deprecation. As a result, the effort of porting to newer versions becomes more expensive due to change accumulation. This could happen due to two reasons: either because they are unaware or dormant systems or because developers follow a specific framework version. A minority of this lack of reactions is explained by client developers reacting in a way different of the one proposed by our API changes (*i.e.,* they are not following the main recommendation of the frameworks).

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 probably lack 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 (Robbes *et al.*, 2012) or a lack in the use of deprecation (Wu *et al.*, 2010; Dig and Johnson, 2005). In our study, this lack of deprecation was mainly due to large refactorings in the frameworks. For example, 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[9]: "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 StackOverflow[10] or mailing lists[11] is the current alternative for client developers, confirming they are popular tools developers use to satisfy their ecosystem-related information needs (Haenni *et al.*, 2014). Based on these results we conclude:

> 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 sometimes use internal parts of frameworks: Internal APIs are unstable and unsupported interfaces (Businge *et al.*, 2013), 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 (Dagenais and Robillard, 2008; Boulanger and Robillard, 2006; Businge *et al.*, 2013; Hora *et al.*, 2016), showing that client systems use internal parts of frameworks to access

[9] http://forum.world.st/moving-moose-to-pharo-3-0-td4718927.html

[10] Coordination via Question and Answer sites: http://stackoverflow.com/questions/15757529/porting-code-to-pharo-2-0

[11] Coordination via mailing lists: http://goo.gl/50q2yZ, http://goo.gl/k9Fl0K, http://goo.gl/SkMORX

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 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.,* Dagenais and Robillard (2008)) other propose the extraction from clients (*e.g.,* Schäfer *et al.* (2008)). 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 shows evidences that frameworks are a more reliable source.

6 Threats to Validity

Construct Validity: The construct validity is related to whether the measurement in the study reflects real-world situations.

Software ecosystems present some instances of duplication (around 15% of the code (Schwarz *et al.*, 2012)), where packages are copied from a repository to another (*e.g.,* a developer keeping a copy of a specific framework version). It may overestimate the number of systems affected to API changes.

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 because 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 (Robbes *et al.*, 2012). 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.

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.

We also identify one threat regarding the time-based analysis. This analysis may suffer from the threat that the propagation time has been computed by analyzing a finite change history where new reactions in the systems could happen in the future.

External Validity: The external validity is related to the possibility to generalize our results.

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. In this context, recently, Wu *et al.* (2016) analyzed the impact of API evolution in the Apache and Eclipse ecosystems.

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 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 client systems and more than 2,800 contributors, it is large enough to be relevant.

7 Related Work

7.1 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 (Messerschmitt and Szyperski, 2005; Jansen *et al.*, 2013), and the second on the artefact analysis aspect, *i.e.,* on the analysis of multiple, evolving software systems (Jergensen *et al.*, 2011;

Lungu, 2009; Robbes *et al.*, 2012). 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*" (Lungu, 2009). These software systems have common underlying components, technology, and social norms (Jergensen *et al.*, 2011).

Software ecosystems have been studied under a variety of aspects. Jergensen *et al.* (2011) study the social aspect of ecosystems by focusing on how developers move between projects in the software ecosystems. The studies of Lungu *et al.* (2010b) and Bavota *et al.* (2013) aim to recover dependencies between the software projects of an ecosystem to support impact analysis. Lungu *et al.* (2010a) focus on the software ecosystems analysis through interactive visualization and exploration of the systems and their dependencies. Gonzalez-Barahona *et al.* (2009) study the Debian Linux distribution to measure its size, dependencies, and commonly used languages.

Mens *et al.* (2014) 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.* (2013) 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.* (2013) investigated API stability and adoption on a small-scale Android ecosystem. In this study, API changes are derived from Android documentation. They 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. In fact, the amount of changes in Android APIs usually triggers questions in StackOverflow, for example, about API behavior or removal (Linares-Vásquez *et al.*, 2014). In a large-scale Android ecosystem analysis, Bavota *et al.* (2015) verified facts that could impact application ratings. Specifically, the authors investigated whether application ratings correlated with the fault- and change-proneness of their depending APIs; they show that applications with high user ratings use APIs that are less fault- and change-prone. Our study is complementary, presenting that API changes may also not propagate to the ecosystem. Recently, Wu *et al.* (2016) analyzed both API changes and usages in frameworks and client systems of Apache and Eclipse. While the authors focus on API changes at class, interface, and method level (*e.g.,* delete type, decrease access, delete method parameter) our changes are in the context of evolution rules (*e.g.,* method `foo()` must or should be replaced by `bar()`).

In a large-scale study, Robbes *et al.* (2012) investigate the impact of a specific type of API evolution, API deprecation, in an ecosystem that includes more than 2,600 client systems; 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 understand the phenomenon of change propagation at the ecosystem level.

7.2 API Evolution Analysis

Many approaches have been developed to support API evolution and reduce the efforts of client developers. Chow and Notkin (1996) present an approach where the API developers annotate changed methods with replacement rules that will be used to update client systems. Henkel and Diwan (2005) propose CatchUp!, a tool that uses an IDE to capture and replay refactorings related to the API evolution. Hora *et al.* (2014) and Hora and Valente (2015) present tools to keep track of API evolution and popularity.

Kim *et al.* (2007) 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 and Notkin (2009) propose 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 (Kim *et al.*, 2007) where only method signatures were analyzed. Each version is represented with predicates that capture structural differences. Based on the predicates, the tool infers systematic structural differences. Nguyen *et al.* (2010) propose LibSync that uses graph-based techniques to help developers migrate from one framework version to another. In this process, the tool takes as input the client system, a set of systems already migrated to the new framework as well as the old and new version of the framework in focus. Using the learned adaptation patterns, the tool recommends locations and update operations for adapting due to API evolution.

Dig and Johnson (2005) help developers to better understand the requirements for migration tools. They found that 80% of the changes that break client systems are refactorings. Cossette and Walker (2012) 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.* (2010) target the mapping between Java and C# APIs while Gokhale *et al.* (2013) present the mapping between JavaME and Android APIs.

8 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 performed in the context of a large-scale software ecosystem, Pharo, with around $3{,}600$ distinct systems. We analyzed 118 API changes extracted from frameworks and we found that 53% impacted other systems. We reiterate the most interesting conclusions from our results:

- API changes ca affect the whole 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. Such analysis can be influenced by the age of the API change.
- 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.

– API changes and deprecation can present different characteristics, for example, API change reaction is slower and smaller.

As future work, we plan to extend this research to analyze ecosystems based on statically typed languages. Therefore, 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, FAPEMIG, Fundect-MS (007/2015), and ANR (ANR-2010-BLAN- 0219-01).

References

Bavota G, Canfora G, Penta MD, Oliveto R, Panichella S (2013) The evolution of project inter-dependencies in a software ecosystem: the case of Apache. In: International Conference on Software Maintenance

Bavota G, Linares-Vasquez M, Bernal-Cardenas CE, Di Penta M, Oliveto R, Poshyvanyk D (2015) The impact of api change-and fault-proneness on the user ratings of android apps. IEEE Transactions on Software Engineering 41(4)

Boulanger J, Robillard M (2006) Managing concern interfaces. In: International Conference on Software Maintenance

Brito G, Hora A, Valente MT, Robbes R (2016) Do developers deprecate APIs with replacement messages? a large-scale analysis on Java systems. In: International Conference on Software Analysis, Evolution and Reengineering

Businge J, Serebrenik A, van den Brand MG (2013) Eclipse API usage: the good and the bad. Software Quality Journal

Chow K, Notkin D (1996) Semi-automatic update of applications in response to library changes. In: International Conference on Software Maintenance

Cossette BE, Walker RJ (2012) Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In: International Symposium on the Foundations of Software Engineering

Dagenais B, Robillard MP (2008) Recommending adaptive changes for framework evolution. In: International Conference on Software engineering

Dig D, Johnson R (2005) The role of refactorings in API evolution. In: International Conference on Software Maintenance

German DM, Adams B, Hassan AE (2013) The evolution of the R software ecosystem. In: European Conference on Software Maintenance and Reengineering

Gokhale A, Ganapathy V, Padmanaban Y (2013) Inferring likely mappings between APIs. In: International Conference on Software Engineering

Gonzalez-Barahona JM, Robles G, Michlmayr M, Amor JJ, German DM (2009) Macro-level software evolution: a case study of a large software compilation. Empirical Software Engineering 14(3)

Haenni N, Lungu M, Schwarz N, Nierstrasz O (2014) A Quantitative Analysis of Developer Information Needs in Software Ecosystems. In: European Conference on Software Architecture Workshops

Henkel J, Diwan A (2005) Catchup!: Capturing and replaying refactorings to support API evolution. In: International Conference on Software Engineering

Hora A, Valente MT (2015) apiwave: Keeping track of API popularity and migration. In: International Conference on Software Maintenance and Evolution, http://apiwave.com

Hora A, Anquetil N, Ducasse S, Allier S (2012) Domain Specific Warnings: Are They Any Better? In: International Conference on Software Maintenance

Hora A, Etien A, Anquetil N, Ducasse S, Valente MT (2014) APIEvolutionMiner: Keeping API Evolution under Control. In: Software Evolution Week (European Conference on Software Maintenance and Working Conference on Reverse Engineering)

Hora A, Anquetil N, Etien A, Ducasse S, Valente MT (2015a) Automatic detection of system-specific conventions unknown to developers. Journal of Systems and Software 109

Hora A, Robbes R, Anquetil N, Etien A, Ducasse S, , Valente MT (2015b) How do developers react to API evolution? the Pharo ecosystem case. In: International Conference on Software Maintenance and Evolution

Hora A, Valente MT, Robbes R, Anquetil N (2016) When should internal interfaces be promoted to public? In: International Symposium on the Foundations of Software Engineering

Jansen S, Brinkkemper S, Cusumano M (2013) Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry. Edward Elgar Pub

Jergensen C, Sarma A, Wagstrom P (2011) The onion patch: migration in open source ecosystems. In: European Conference on Foundations of Software Engineering

Kim M, Notkin D (2009) Discovering and Representing Systematic Code Changes. In: International Conference on Software Engineering

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

Linares-Vásquez M, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D (2014) How do API changes trigger stack overflow discussions? a study on the android SDK. In: International Conference on Program Comprehension

Lungu M (2009) Reverse Engineering Software Ecosystems. PhD thesis, University of Lugano, Switzerland (October 2009)

Lungu M, Lanza M, Gîrba T, Robbes R (2010a) The small project observatory: Visualizing software ecosystems. Science of Computer Programming 75(4)

Lungu M, Robbes R, Lanza M (2010b) Recovering inter-project dependencies in software ecosystems. In: International Conference on Automated Software Engineering

McDonnell T, Ray B, Kim M (2013) An empirical study of API stability and adoption in the android ecosystem. In: International Conference on Software Maintenance

Meng S, Wang X, Zhang L, Mei H (2012) A history-based matching approach to identification of framework evolution. In: International Conference on Software

Engineering
Mens T, Claes M, Grosjean P, Serebrenik A (2014) Studying evolving software ecosystems based on ecological models. In: Mens T, Serebrenik A, Cleve A (eds) Evolving Software Systems, Springer Berlin Heidelberg
Messerschmitt DG, Szyperski C (2005) Software ecosystem: understanding an indispensable technology and industry. MIT Press Books 1
Nguyen HA, Nguyen TT, Wilson G Jr, Nguyen AT, Kim M, Nguyen TN (2010) A graph-based approach to API usage adaptation. In: International Conference on Object Oriented Programming Systems Languages and Applications
Robbes R, Lungu M, Röthlisberger D (2012) How do developers react to API deprecation? The case of a smalltalk ecosystem. In: International Symposium on the Foundations of Software Engineering
Schäfer T, Jonas J, Mezini M (2008) Mining framework usage changes from instantiation code. In: International Conference on Software engineering
Schwarz N, Lungu M, Robbes R (2012) On how often code is cloned across repositories. In: International Conference on Software Engineering
Wu W, Gueheneuc YG, Antoniol G, Kim M (2010) Aura: a hybrid approach to identify framework evolution. In: International Conference on Software Engineering
Wu W, Khomh F, Adams B, Guéhéneuc YG, Antoniol G (2016) An exploratory study of API changes and usages based on apache and eclipse ecosystems. Empirical Software Engineering
Zaki M, Meira Jr W (2012) Fundamentals of data mining algorithms
Zhong H, Thummalapenta S, Xie T, Zhang L, Wang Q (2010) Mining API mapping for language migration. In: International Conference on Software Engineering