How Do Developers React to API Deprecation? The Case of a Smalltalk Ecosystem

Romain Robbes
PLEIAD @ DCC
University of Chile
rrobbes@dcc.uchile.cl

Mircea Lungu
Software Composition Group
University of Bern
lungu@iam.unibe.ch

David Röthlisberger
PLEIAD @ DCC
University of Chile
roethlis@dcc.uchile.cl

ABSTRACT

When the Application Programming Interface (API) of a framework or library changes, its clients must be adapted. This change propagation—known as a ripple effect—is a problem that has garnered interest: several approaches have been proposed in the literature to react to these changes.

Although studies of ripple effects exist at the single system level, no study has been performed on the actual extent and impact of these API changes in practice, on an entire software ecosystem associated with a community of developers. This paper reports on an empirical study of API deprecations that led to ripple effects across an entire ecosystem. Our case study subject is the development community gravitating around the Squeak and Pharo software ecosystems: seven years of evolution, more than 3,000 contributors, and more than 2,600 distinct systems. We analyzed 577 methods and 186 classes that were deprecated, and answer research questions regarding the frequency, magnitude, duration, adaptation, and consistency of the ripple effects triggered by API changes.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance and Enhancement**]: Restructuring, reverse engineering, and reengineering

Keywords

Ecosystems, Mining Software Repositories, Empirical Studies

1. INTRODUCTION

Most of the software engineering research focuses on tools and techniques for analyzing individual systems: quality assessment, defect prediction, automated test generation, impact analysis, all are techniques that aim at supporting the developer and improving the resulting software.

However, a software system does not exist in isolation, but instead, it is frequently part of a bigger software ecosystem [21] in which it usually depends on other systems and, sometimes, other systems are dependent on it. Ecosystems usually exist in large companies, organizations, or open source communities. As more and more of our society's infrastructure runs on software, the size and number of such ecosystems increases. In this context, research should also focus on designing tools and techniques to support developers working in software ecosystems.

A number of problems that are relevant for individual system analysis are likely to remain relevant at the ecosystem level; and the importance of some problems might even augment. In this article we set out to discover whether the problem of impact analysis and prediction that has been studied already at the level of individual systems is also relevant at the ecosystem level.

When a project that contains functionality reused by many others in the ecosystem changes, this might trigger a wave of changes in the ecosystem. At the moment there is no tool support for predicting such changes, so the developers often do not know whether their change will impact other systems or not. Two anecdotal examples illustrate the problems and opportunities associated with the lack of tool support for change impact analysis at the ecosystem level:

1. While discussing with developers of a large corporation, we discovered that sometimes a developer would make a change but he would only find out whether his change impacted some other systems multiple days later. This was a result of a very long build cycle.

2. While studying the mailing list archives of the Seaside project, part of the Squeak/Pharo ecosystem, we discovered an email in which one developer was asking about several classes that his application was depending on but he could not find in the latest version of the framework. One of the Seaside maintainers answers[1]:

> They have been dropped. A mail went out to this list if anybody still used them and nobody replied. [...] Personally I don't know of any application that uses these dialogs.

But how often do such changes that impact other systems happen, and how broad is their impact? Take the example of the following event in the Squeak/Pharo ecosystem: at some point the FillInTheBlank class, a broadly used utility class was deprecated and its responsibilities moved to the UIManager class. Figure 1 shows how the usage of FillInTheBlank initially increased, and then abruptly decreased as all the clients were moving towards using the UIManager instead. More than 35 projects were impacted by the deprecation.

These examples hint at the necessity of providing tool support for maintaining a continuous awareness of the potential impact of a change at the level of the ecosystem. However, they are only anecdotal. To the best of our knowledge, there has been no large-scale study

[1]Entire exchange available at: http://bit.ly/gnwNfV.

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

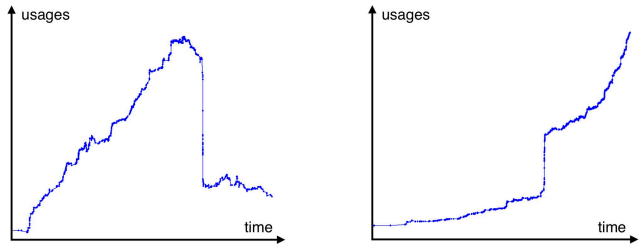

Figure 1: The usage over time of classes FillInTheBlank (left) and UIManager (right) which takes over its responsibilities

that would quantify how severe is the problem of change impact at the ecosystem level. The only studies available are performed at smaller scales, studying frameworks in isolation without taking their clients into account [7].

Research Goals

The goals of this article are to determine: (1) whether there is enough evidence for the necessity of building automated support for impact analysis at the ecosystem level; (2) if needed, provide a sketch of a solution; and (3), whether we can provide immediate actionable information to alleviate the problem. To do this we set out to quantify how often a change in a system will trigger changes in the other systems of the ecosystem. In this study, we focus on the special case of *deprecated* entities, *i.e.,* API elements that are explicitly marked to indicate they are scheduled for later removal.

This paper reports on an empirical study on the impact of API changes in the Squeak/Pharo software ecosystem. We analyzed 577 methods and 186 classes that were indicated as deprecated in various frameworks and core libraries by Squeak and Pharo developers. In each case, we recorded how projects in the ecosystem reacted to the deprecation of the method; we determined that 80 method deprecations and 13 class deprecations caused ripple effects in client projects later on. We investigate the following research questions in order to better characterize the ripple effect phenomenon:

- **Frequency.** $RQ1$. How often do deprecated API methods cause ripple effects in the ecosystem?
- **Magnitude.** $RQ2$. How many projects react to the API changes in the ecosystem and how many developers are involved?
- **Duration.** $RQ3$. How long does it take for projects to notice and adapt to a change to an API they use?
- **Adaptations.** $RQ4$. Do all the projects in the ecosystem adapt to the API changes?
- **Consistency.** $RQ5$. Do the projects adapt to an API change in the same way? $RQ6$. How helpful was the deprecation message, if any?

Structure of the paper. We start with a review of related work (Section 2). We then present a list of challenges for API change analysis (Section 3) and the characteristics of our case study, the Squeak/Pharo ecosystem (Section 4). We then introduce our ecosystem model and the way we model API changes and ripple effects (Section 5). We describe our methodology to detect ripple effects triggered by deprecation (Section 6), before presenting our results (Section 7). We close the paper with a presentation of the implications of our study (Section 8), a discussion of the threats to its validity (Section 9), and conclude in Section 10.

2. RELATED WORK

Our work directly relates and builds on three research areas: Analyses of software ecosystems and large collections of software projects, studies about ripple effects and the evolution of APIs, and approaches that assist developer confronted with API changes.

2.1 Software Ecosystems Analysis

The term software ecosystem has several meanings: some focus on the business aspect [24, 18]; others focus on the analysis of multiple software systems aspect [21, 19]. For the purpose of this work use the latter meaning, and more precisely we consider an ecosystem to be *"a collection of software projects which are developed and co-evolve in the same environment"* [21]. The environment can be an organization, or an open-source community. The focus is on analyzing multiple systems which have "common underlying components, technology, and social norms" [19].

Ecosystem analysis research has two facets: the study of the social aspects, embodied in the developer collaboration, and the study of the source code of the component systems. Jergensen *et al.* study the way the developers migrate across the projects in an ecosystem [19]. In previous work we focused on reverse engineering a software ecosystem by generating high-level views capturing various aspects of its structure and evolution [22]. Other approaches focus on discovering and comparing methods extracting dependencies between the projects in an ecosystem [28, 23]. Gonzalez-Barahona *et al.* performed a study of the Debian Linux distribution, in terms of the evolution of its size and of its programming languages [11].

The analysis of large source code data does not necessarily involve an ecosystem. Code search was the first application of source code analysis beyond the scope of the individual system: the first academic search engine Sourcerer [2], was followed by several other efforts [17, 29]. In 2007, Mockus studied the phenomenon of large scale software reuse in a selected group of open-source projects [25]. Two years later, Mockus reported on his experience in amassing a very large index of version control systems [26].

Gruska *et al.* mined a large sample of Linux projects (6,000), to extract usage properties from their source code, to subsequently detect usage anomalies [13]. They do not consider the evolution.

2.2 Ripple Effects and API Evolution

In this paper we investigate how clients react to changes in frameworks and libraries they use. Our goal is to understand the propagation of the changes in the ecosystem as a whole, and to examine whether we can predict and support their propagation.

Similar studies have been performed at smaller scales. The work of Haney [14], Yau *et al.* [37], and Black [3] investigated ripple effects and possible methodologies to address them, at the scale of a single system. One of them is impact analysis [1, 4]. Later studies were performed on frameworks, libraries, and their clients in particular. Dig and Johnson studied the evolution of the API of four frameworks and one library, and concluded that in 80% of the cases, the changes that break clients of a library or framework are refactorings [7]. Similar earlier work was performed by Opdyke on refactorings [27], and Graver on the evolution of an object-oriented compiler framework [12]. Graver used refactorings successfully during the evolution of a compiler, when extending it to incorporate types; he does not comment on whether the documented changes impacted the API of the framework. Likewise, Tokuda and Batory applied refactoring operations on two object-oriented designs, in order to recreate the changes between versions of these systems. In their case, the API changes were extensive [34].

To our knowledge, we are the first to undertake a study on the effect of API changes on actual clients of libraries and frameworks.

2.3 Reacting to API Changes

Several automated approaches have been proposed to react to external changes. Zeller introduced delta debugging, which partitions the change sets in a version control archive in order to pinpoint the change being the cause of a bug [38].

Later, both Henkel and Diwan, and Ekman and Asklund, proposed to record the refactorings performed in an IDE on the source code of the framework in order to replay them in the client [15, 10]. Dig *et al.* built a refactoring-aware versioning system, storing the refactorings and the other source code changes in a single repository [8].

In the absence of recorded refactorings, several approaches detect them from several versions of a library. Weissgerber and Diehl used a technique based on signature changes and clone detection in the method bodies [35], while Dig *et al.* use shingle encoding of ASTs to quickly compute the similarity between candidates across versions, and then used semantic analyses to refine the candidates [6]. Taneja *et al.* expanded Dig's technique to work for library APIs that do not have callers [33]. Kim and Notkin identify systematic changes (including refactorings) based on a logic engine [20].

Several approaches go further and recommend changes to be made in order to recover from the API change. Dagenais and Robillard observe how the framework itself adapted to its changes [5], while Schaffer *et al.* observe how other clients adapted [31]. Wu *et al.* introduced an approach incorporating text similarity and handling more complex cases [36]. Dig *et al.* adopt the opposite approach, and introduce a layer between the client and the new version of the library, so that the client is shielded from the changes [9]. Holmes and Walker also adopt a different viewpoint, and monitor the versioning system of libraries used by a system and filter changes according to how relevant they are for the system (*i.e.*,, whether the system uses the entities that were modified or not) [16]. In all cases, the approaches were validated on a selected number of frameworks, for statistically typed programming languages.

3. CHALLENGES IN ECOSYSTEM AND API CHANGE ANALYSIS

To analyze API changes, and determine if they trigger ripple effects, we need a reliable process to detect them in the large amount of data at our disposal. This process has three main steps: (1) curating the data, (2) building a model of the ecosystem, and (3) generating a list of candidate API changes, and their impact on clients. This process is subject to a set of challenges:

1. **Curating the data.** Determining which projects are part of the ecosystem can prove to be a challenge in itself. Some communities—especially larger ones—may spread over multiple websites that need to be individually crawled to gather the data. In our case, this problem is simpler, since the Squeak/Pharo community mainly uses a single web site to store source code.

2. **Modeling thousands of evolving systems.** An evolving ecosystem contains a large amount of data that must be handled appropriately with the correct level of abstraction. In the case of ripple effect detection, we must model changes to the API of systems, and do so between individual versions of systems: The *Ecco* meta-model is our proposed solution.

3. **Detecting ripple effects.** After the model of the ecosystem is built, the ripple effects themselves—original API change and reactions and adaptation of the clients over time—must be modeled and detected.

The next three section detail how we addressed these challenges.

4. THE SQUEAK/PHARO ECOSYSTEM

Our first task was selecting an adequate ecosystem that would provide support for answering our research questions. We chose the ecosystem built around the Squeak and Pharo open-source development communities (Pharo is a fork of Squeak). It is hosted by the Squeaksource[2] source code repository. Since 2004, Squeaksource hosts a large number of individual repositories of a distributed, language-aware version control system named Monticello. Squeaksource is the foundation for the software ecosystem that the Squeak and Pharo communities have built over the years. As of March 2011, Squeaksource hosts 2,601 systems in which 3,036 contributors performed more than 127,000 source code commits. The combined size of all the versions is more than 11.4GB of compressed source code (Monticello stores versions as zip files).

4.1 The Squeak and Pharo Communities

Squeak is a dialect of Smalltalk, as well as an implementation of its programming environment. It was created in 1996, and has gathered around it an active community. The One Laptop Per Child initiative delivered the laptops with the EToys end-user programming implemented in Squeak. The Seaside web-development framework—the main competitor for Ruby on Rails as the framework of choice for rapid web prototyping—is developed by the Squeak community. The Moose analysis platform has been ported to the Pharo platform.

Pharo forked from Squeak in early 2008. It is now a distinct Smalltalk distribution with a distinct (but somewhat overlapping) community. One of the main goals of Pharo has been to provide a distribution of Smalltalk with a wide set of libraries, an IDE, and other applications, which are all open-source and liberally licensed. As a sign of the distinction between the two communities, Pharo has a distinct mailing list, a separate web presence,[3] and is rapidly evolving. If Squeak's goal is to be an inclusive development platform (featuring multimedia facilities, an environment for children programming, etc), Pharo is geared towards being a development platform more catering to the industry and the academia. As such, since the fork the Pharo community has been doing a large amount of refactoring and cleanup of its code base.

If these two communities are growing apart, they still share the same version control infrastructure: Monticello for version control, and the Squeaksource repository.

4.2 The Monticello Version Control System

Monticello is a distributed, language-aware version control system.[4] When performing a commit of a project with Monticello, a snapshot of the source code is stored. Since this process does not involve computing deltas with the previous version, a Monticello repository is simply a file system directory. Besides the snapshot of the source code, Monticello versions record meta-information, such as author, time stamp, commit comment, and, particularly useful for evolution analysis, version information: Each Monticello version contains the list of all its ancestor version referenced by name.

Monticello is also language-aware; it is designed to version Smalltalk source code only. As such, versioning is not performed at the level of files and lines, but at the level of packages, classes, and methods. This allows the differencing and merging algorithms to use much richer information than more conventional versioning systems, easing the process for the developer—and for our analyses—at the price that other resources beyond source code need to be handled separately [30].

[2] http://www.squeaksource.com
[3] http://pharo-project.org
[4] http://www.wiresong.ca/monticello/

4.3 The Squeaksource Super-repository

Squeaksource is an online source forge, built with the Seaside Smalltalk framework, which is the foundation for the software ecosystem that the Squeak and Pharo community have built over the years.[5] Squeaksource has been operational since late 2003, and publicly announced in early 2004. It has quickly become the *de facto* platform for sharing open-source code for the Squeak and Pharo communities. Today, the overwhelming majority of Squeak and Pharo developers use Squeaksource as their source code repository, making it a nearly complete view of the Squeak and Pharo software ecosystem.

At its core, Squeaksource is simply a set of Monticello repositories (hence a super-repository), accessible via HTTP, where people can commit their source code. Squeaksource provides accounts for developers, who can define projects on the web site and specify who can access their projects (ie. granting read/write access to the project's individual repository). It also provides services such as per-project presentation pages, RSS feeds, wikis, and overall statistics, and enables one to browse the source code of any version stored on the web site, provided one has access rights to do so.

An alternative to our choice would have been to analyze a development community centered on a more popular language such as Java developers. This would have implied a distinct set of trade-offs. Even if Monticello and Squeaksource have shortcomings, they are extremely convenient to process as (1) the data is centralized on one server and is simply structured in file directories, and (2) we can reuse Monticello's language awareness to work with object oriented concepts instead of text files. To summarize, the reasons for the choice of the Squeak/Pharo ecosystem as a case study are:

- Squeaksource is the de facto source code repository for the overwhelming majority of the open-source code produced by the Squeak and Pharo communities.
- The community—more than 3,000 contributors strong—is large enough to be relevant.
- The versioning system used by the community, Monticello, is a high-level versioning system, working at the level of packages, classes and methods, instead of files and lines of code, thus considerably simplifying our analysis.
- Several large and widely used frameworks are hosted on Squeaksource, as well as a variety of smaller ones. Overall, several hundred projects can be classified as frameworks.
- The fact that the community has forked is a factor that contributes to the frequency of API changes and ripple effects. Pharo wants to "clean up" its code base, incurring a lot of changes making it less compatible with Squeak code, and triggering frequent updates for Pharo developers. This may or may not affect Squeak developers.

5. MODELING EVOLVING ECOSYSTEMS

5.1 The Ecco-Evol Metamodel

There are two conflicting requirements that need to be addressed by an ecosystem model that allows the detection of API changes and ripple effects in ecosystems: on the one hand, the model should be lightweight enough to allow the modeling of a large number of projects—each of which might in turn have a large numbers of versions; on the other hand, the model should represent ecosystem evolution and record the details of each of the versions, such that the structural changes between versions can be tracked and used to determine the occurrences of ripple effects. To build such a model we drew inspiration from two previous ecosystem models that we proposed in previous work:

- *Ecco* [23] is a lightweight, language-independent representation of the data in ecosystem snapshots, providing full (language-dependent) access to the details on demand. Ecco's main unit of abstraction is the system. Each system maintains lists of *provided* (defined in the system), and *required* (used by the system) entities over its lifetime. Based on these lists of provided and required entities one can recover dependencies between systems (*e.g.,* if system B requires a set of entities and system A is the only provider of these entities in the ecosystem, we can infer that B depends on A). Ecco does not represent multiple versions of a system.
- *RevEngE* [21] is a meta-model that supports modeling the structural evolution of an ecosystem; in our previous work we have proposed it as a generic language-independent ecosystem meta-model which supports evolutionary analysis. The model uses a full FAMIX model to represent every version of the individual systems. This turns out that is very expensive both in terms of building the models, maintaining them, and executing operations on them.

We enhanced Ecco with a model of versions inspired by RevEngE: we model an ecosystem with systems and versions, but for each pair of successive versions we only keep a delta. We call this new meta-model *Ecco-Evol*.

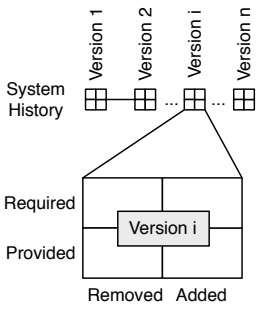

Figure 2: Ecco-Evol models every version as a delta

Figure 2 shows that in *Ecco-Evol*, each version keeps track of the changes between itself and its predecessor. These changes are high-level, and consist in sets of additions and removals of required and provided entities (methods and classes). In addition to changes, the meta-model keeps track of metadata (author, time stamp), and links to ancestors and successors versions. The model allows more than a single link to ancestors and successors in order to accommodate forks and merge operations. The model allows details-on-demand: each version contains a link to its original snapshot on the file system and can use Monticello's infrastructure to inspect the definitions contained in it, as well as performing differences with its ancestors and successors.

5.2 Detecting API Changes and Ripple Effects

At the level of the ecosystem, a ripple effect is determined by a change to a system's API which originates in one system and propagates to other systems. A ripple effect is a pair of two elements $(C_O, C_{i,i<n})$:

[5] http://www.squeaksource.com

- C_O - An original source code change in an original project (P_O). This change must mutate in some way the API of the project (*e.g.,* by adding, removing, or modifying artifacts)

- $C_{i,i<n}$ - A set of changes which belong to projects that are distinct from the original project (P_O), which are performed as the direct consequence of the original change (C_O) and mutate the source code of the impacted project (*e.g.,* by adding, removing or modifying artifacts).

We simplify the problem of ripple effects in the ecosystem in two ways. First, we consider that ripples propagate only one level from the source. Ripples of higher degrees, where a change triggers other changes which in turn trigger yet other changes, remain for future work. Second, we consider the ripple effects based only on a very specific semantic change, *deprecation*: even detailed static analysis may not always be able to infer whether a generic semantic change will affect other artifacts. This becomes even harder at the scale of an entire ecosystem. In contrast, a deprecation is an explicit statement of intent that the API is changing.

When looking at the syntactic changes of the original source code change we consider two types of artifacts, classes and methods. For each type of entity there are three elementary changes that one can introduce in a project that can later propagate to other projects:

1. *Addition of provider*. The change introduces a new entity that will be provided by a project. In most of the cases this means extending the API of the system.

2. *Removal of provider*. The change removes an entity that was provided by the project. In most cases this means removing a method from the API of the project.

3. *Modification of provider*. This changes the provided entity in some way. A special case of this is annotating a method or class as *deprecated*, tagging it for later removal.

A provider addition does not render the client incompatible with the provider and therefore does not force him to react. Therefore, we can not be sure that simple additions will introduce ripple effects. On the other hand, removing or renaming a provided entity that a client uses effectively forces them to update or use the outdated version of the framework. A search for ripple effects based on method removals and renames will provide a lower bounds on the number of API changes causing ripple effects in the ecosystem.

The case of removal must be treated with attention. The developers of a library or a framework might decide to never actually remove a provided artifact but instead just mark it as deprecated (through annotations or special method calls), signaling in this way that in future versions of the system that artifact is likely not to be supported anymore. In fact, this is the most disciplined way to remove a method, while allowing the clients time to react to the change. In this article we focus on changes to the provider that deprecates it, as this is the clearer indication that an API change is taking place.

6. METHODOLOGY

To build and validate the list of API changes causing ripple effects occurring in the ecosystem as a result of method deprecations, we employ the following methodology.

Generate a list of candidates We query the model of the ecosystem for every usage of a given method. To generate our list of candidate changes to analyze, we operated under the assumption that *deprecated methods* are a marker of API changes, and as such a likely cause of ripple effects. Indeed, deprecated methods are explicitly tagged for later removal.

When deprecating a method, Smalltalk users insert a call to a deprecation method in the body of the deprecated method. Each call to the deprecated method will output a warning at run-time. The following method in Moose—a reverse-engineering environment—was deprecated; the deprecation message indicates the preferred alternative:

```
addEntity: anElement

    self deprecated: 'use add: instead'.
    ↑self add: anElement.
```

We query the ecosystem for all the commits that introduce a requirement to one of the deprecation methods; the actual deprecated method is then extracted from the differences between the two snapshots. In total, 577 methods were deprecated in the ecosystem in the period from 2004 to 2011. In addition, we expanded the scope of our search by also considering the 186 classes that featured a deprecated method. In some cases, a deprecated method is moved from one class to another; if we do not consider the classes, we do not see this kind of changes if the name of the method stays identical.

Assess usage of each candidate We filtered the initial set of deprecated methods leaving only the ones for which the method has been removed in at least 3 transactions over time. This filters a portion of the methods that are removed as a result of the natural evolution of the system[6]. We manually inspected each of the remaining candidates to assess whether they caused a ripple effect. To do so, we assessed the following parameters:

- Number of projects using the candidate: At least two distinct projects must be using the method/class (the originating project, and a client).

- Evolution of the usage of the candidate over time (as shown in the graphs of Figure 1). Deprecated API elements causing ripple effects experience a drop of usage after a point, whereas others will usually grow in usage continuously or stabilize. Methods or classes that are replacements of other entities may sometimes see their popularity surge.

Inspect changes to confirm, and determine replacements For each candidate, we analyze the related changes. We: (1) retrieve the list of changes between each two versions in which the candidate was involved (using Monticello's infrastructure), (2) filter the list of changes, keeping the list of methods that were changed and who have references to the candidate, and (3) display those changes in a Ripple Effect Browser. This tool we developed orders transitions between versions by date; for each commit, it shows the changed methods; for each method, it presents the differences between its previous and current versions, highlighted to emphasize additions and removals.

Using the Ripple Effect Browser, one can finally decide whether a deprecated candidate caused a ripple effect, based on the magnitude of the changes, their spread among projects, and the amount of unrelated changes. In most cases, the inspection allows one to infer what is the replacement to use. By manual inspection one can detect whether the deprecated method is replaced consistently with another method. If there is such method we (4) wrote it down in association with the deprecated method.

[6]By natural evolution we mean the evolution of the system independent of the API deprecation.

An additional benefit is that this process sometimes triggered the discovery of new ripple effect candidates that we added to the list and checked further (for instance, when a project adapted to two distinct API changes in the same commit). We found 14 additional methods and 6 additional classes in this manner, that were not explicitly marked as deprecation. We refer to these as *implicit deprecations*.

Using this process, the first author processed the initial list of candidates, and trimmed it down to 180 methods classified as "probable ripple effects", and 20 classes. At that stage, the second author repeated steps (2), (3) and (4) on the list, to corroborate the findings of the first author. Differences were discussed, and we opted for a lower threshold with respect to noise (*i.e.,* the amount of unrelated changes or changes related to the natural evolution that we tolerated). After this process, we had a final list of 113 confirmed ripple effects (94 methods, and 19 classes, counting the additional 14 methods and 6 classes we encountered during the inspection itself).

7. RESULTS

In this section, we answer the six research questions about API changes that we outlined in the introduction.

7.1 Frequency of Ripple Effects

$RQ1$**. How often do deprecated API methods cause ripple effects in the ecosystem?**

From the 577 deprecated methods in the ecosystem, we discovered that 80 (14%) produced non-trivial API changes effects that impacted at least one other project. From the 185 deprecated classes 13 (7%) provoked reactions that impacted at least one project. As such, a minority of deprecations appear to cause reactions in other projects.

There are two possible explanations for the low numbers:

1. Most deprecated entities are only used internally in a system, are not part of the public API, and hence not considered in our study. Developers, without knowing who are the users of their API follow defensive practices by deprecating even entities used only internally.

2. The clients are still unaware of the deprecation, and the updates are still pending. Later, we see that projects take time to react to the API changes, hence some pending changes may not have been discovered yet.

Anecdotal evidence gleaned from browsing the API changes and their reactions indicate that the first explanation account for the majority of the cases, but the second should not be excluded. In the rest of this section, we add to these numbers the 14 methods and 6 classes that we identified separately as implicit deprecations.

7.2 Magnitude of Ripple Effects

$RQ2$**. How many projects react to the API changes in the ecosystem and how many developers are involved?**

In this question, we seek to quantify the reactions to the API change, in the cases where reactions occurred. We quantify the reactions in terms of numbers of projects and packages affected, numbers of developers affected, and total number of changes. To determine the magnitude of a ripple effect, we first need to estimate when the change that caused it was introduced. To do this we employ a set of heuristics.

We first locate the change introducing the deprecation, if present. However, in some cases this change does not exist (for classes or methods identified separately), or is performed after the ripple effect already started. We hence also estimate the date of the ripple effect as the date when the method reached its peak usage, and return the last change in the originating project that modified or removed the deprecated entity. We return the change which has the earliest date between the deprecation change and the change before the peak. All commits after this change that remove a requirement to the deprecated entity are considered to be reactions to the API change.

Looking at the data like this reveals that the largest ripple effect concerns the deprecation of the class Preferences: 79 projects react; in the box plot shown in Figure 3 (i) this deprecation is depicted as the dot at the top (in a box plot all outliers are shown as dots). The third quartile is 12 (25% of the ripple effects cause reactions in 12 or more projects, forming the top of the box in the box plot), the median is 5 (middle of the box), the first quartile is 3 (bottom of the box), and the minimum is 1 (bottom whisker of the box). The top whisker at 20 marks the highest number of reacting projects that is still not considered to be an outlier. On average, 9.2 projects react to a deprecation causing a ripple effect.

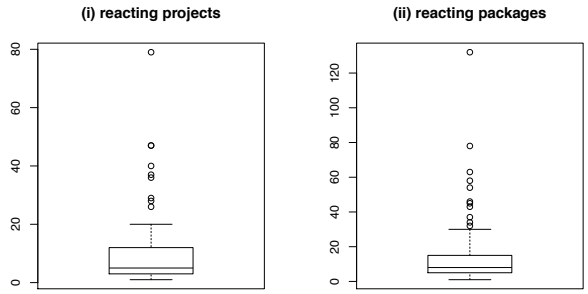

Figure 3: Box plots for (i) projects and (ii) packages reacting to API changes.

The figures at the level of packages and individual changes follow the same distribution. The deprecation of class Preferences caused reactions in 132 packages, (3rd quartile: 15; median: 8; 1st quartile: 5; average: 14). Preferences caused 531 individual reactions, *i.e.,* commits removing dependencies to it (3rd quartile: 28; median: 12; 3rd quartile: 7; average: 29). These distributions are shown as box plots in Figure 3 (ii) and Figure 4 (i). We expected at least one commit per package, but the distribution shows that more are required: the median ripple effect causes 5 projects to react, but triggers 12 individual changes. We clearly see that it usually takes several commits before resolving a ripple effect in a single project, so the changes are not trivial.

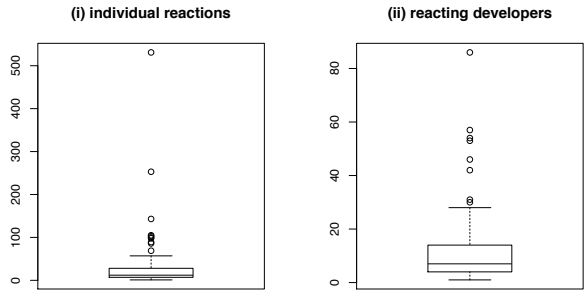

Figure 4: Box plots for (i) individual reactions, and (ii) affected developers

We compute the number of developers affected by the ripple effect simply as the number of authors of commits that react to the ripple effect, shown in Figure 4 (ii). As before, the largest ripple effect, Preferences, involved a large number of developers (86). The top 25% of ripple effects all involve more than 14 individual developers, strengthening the intuition that some ripple effects have large consequences. The median is 7, and the first quartile 4, showing that the majority of ripple effects involve much less people. We also see that the numbers of developers involved in the ripple effect is higher than the number of projects, implying that it is common that two or more developers involved in the same projects have to react to the changes, meaning that a second developer has to pick up where the other left off. This further confirms that reacting to an API change is not trivial, and that a developer can easily overlook some pending changes.

7.3 Duration of Ripple Effects

RQ3. **How long does it take for projects to notice and adapt to a change to an API they use?**

A quick reaction to API changes is desirable. As changes get older, their rationale become less clear as other changes accumulate. Likewise, when they adapt to an API change, individual projects should adapt at once to it, and not over a long period of time, as they are in an inconsistent state during that period.

We estimate the reaction time to an API change as the number of days between the starting time and the first reaction to the API change. The reaction time varies wildly as depicted in the first box plot in Figure 5. The first quantile is at 0 days, indicating that a non-negligible minority of API changes see a reaction on the same day as the change: this is possible if a developer works both on the library and some of its clients, or if developers coordinate via mailing lists, for instance. The median time is at two weeks, while the third quartile is at 84 days, indicating that a strong minority of API changes may take 3 months or more to be acted upon. Some API changes take even longer (more than a year) to be acted upon, which raises further concerns, as the older changes get, the harder they are to understand, and the more they tend to accumulate.

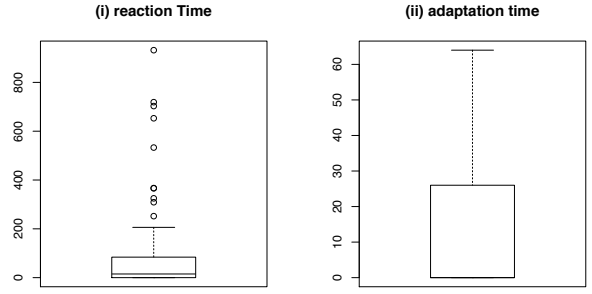

Figure 5: Box plots for (i) reaction time and (ii) adaptation time

For a large project, adapting to a simple API change may be a challenge due to the sheer size of the code base. We looked at the adaptation time for the ripple effect on a per-project basis. We measure the interval between the first and the last reaction to the ripple effect on the same project. As shown with the second box plot in Figure 5 [7], a large proportion of projects react quickly: the median is at 0 days, indicating that more than half of the projects fix an API change extremely quickly. With the help of refactoring tools, correcting an API change such as a rename can be done instantly, if the code base is manageable.

On the other hand, the third quartile is with 26 days much higher. Some projects may even take more than an entire year to adapt to an API change. Indeed, some of the largest projects on SqueakSource are software distributions, where the large size of the system makes it comparable to a small-scale ecosystem. At these scales, it is easy to overlook that a package has not been updated yet. This situation is compounded by the fact that Smalltalk is dynamically typed, so a call to an out of date method will not be found at compile time, but at runtime. This makes it possible that even smaller systems may miss some updates and not notice them for a long time. A statically-typed language will catch these errors at compile-time; as such, a comparison would be extremely interesting. The situation is not clear-cut: in large systems, a complete re-compilation may not be a daily occurrence.

The findings in this section lead one to conclude that the problems related to API changes are related to both the awareness of the changes, and to performing the update itself: in most cases, the update was carried out in a day; a minority of cases took much longer, with the associated consequences of adapting to changes late, and doing so partially.

7.4 Adaptations to Ripple Effects

RQ4. **Do all the projects in the ecosystem adapt to the API changes?**

As we have seen, some projects take a long time to react to an API change. Other projects do not react at all. Figure 6 (i) and (ii) show the distribution of projects and packages affected by API changes in the analyzed ecosystem. The numbers of affected projects and packages are much higher than those that actually react to API changes (as shown in Figure 3): the median of affected projects by an API change is 25 compared to 5 projects also reacting to it (packages: 44 compared to 8). The first quartile is 11 projects affected compared to 3 reacting and the third quartile is 61 projects affected compared to 12 reacting (packages: 1st quartile 20 to 5 and 3rd quartile 99 to 15). Most projects and packages are affected by the Preferences deprecation: 314 and 625, respectively.

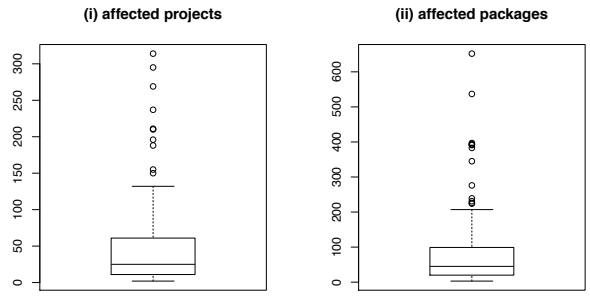

Figure 6: Box plots for (i) projects and (ii) packages affected by API changes

Comparing the ratio of reacting projects to the ratio of all projects using the deprecated entity gives the distribution depicted in Figure 7 (i). We find that a surprisingly low number of projects react, with the median at 20% (1st quartile: 13%; 3rd quartile: 31%). In the following we investigate possible reasons for this low reaction.

In a software ecosystem, a possibly large portion of the systems may be stagnant, or even dead projects. Our first hypothesis is

[7] For legibility, we filter outliers; the 90th percentile is 243 days

Project category	Description
Affected	Projects using the deprecated entity and thus being affected by the deprecation
Reacting	Subset of the affected projects that remove the usage of the deprecated entity
Broken	Affected projects not removing the dependency to the deprecated entity
Stagnating	Affected projects with little or no activity (less than 10 commits) after the deprecation
Dead	Affected projects without any activity after the deprecation
Counter-reacting	Affected projects not removing the usage of the deprecated entity but adding even more usages

Table 1: Classification of projects affected by an API change

that the projects that did not react, died before the ripple effect happened. We consider a project as *dead* if there are no commits to its repository after the API change that triggered the ripple effect. Likewise, a project is *stagnant* if a minimal number of commits (less than 10) have been performed after the API change. Table 1 lists the different classifications of projects regarding API changes. Removing dead or stagnant projects (*i.e.*,, keeping *alive* projects only) paints a different picture, seen in Figure 7 (ii): 40% of the projects with a reasonable level of activity do react to the changes (1st percentile: 29%; third percentile: 52%).

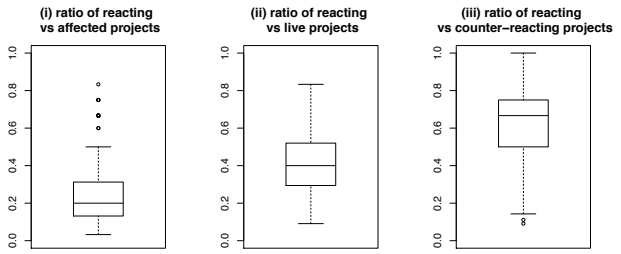

Figure 7: Box plots for ratios of: (i) reacting and affected projects; (ii) reacting and alive projects (dead or stagnating projects removed); and (iii) reacting, alive projects with counter-reactions removed

The second reason why a project would not react to a change is when it is using another version of the library or framework changing its API, one in which the API did not change. This can happen when a project does not have the manpower to keep up with the evolution of another and freezes its relationship with a given version that is sufficient for it. Another reason which might be particular to our case study is the split in the communities of Squeak and Pharo and the fact that Pharo more aggressively refactors its code base.

To estimate this effect, we measure the number of projects that actually *add* more usages of the deprecated entity, and never remove any usages of it (that is, they are *counter-reacting* to the deprecation); we assume that since they continue to actively use the deprecated entity, they are well aware of its status in their branch of the community or are not willing to update it. Removing these from the set of projects that do not react to an API change raises the median to 66% (1st quartile: 50%; 3rd quartile: 75%); the box plot of the distribution is presented in Figure 7 (iii).

These figures show that a large majority of projects that are moderately active and in the same part of the community do update regularly to API changes. However, the other reality is that a similarly considerable number of projects does not update either because they are dormant projects, or because they have frozen their dependency to an old version of the library or framework. This means that the effort of porting to newer versions or other forks becomes progressively more expensive as time goes by and changes accumulate.

7.5 Consistency of Adaptations

*RQ*5**. Do the projects adapt to an API change in the same way?**

Ideally, an API change should provide a single replacement, making the adaptation painless. In practice, some API changes have less clear-cut solutions, making each adaptation unique.

To study how consistent the client adaptations are after a deprecation of entity in the API, we computed a list of possible replacements for each deprecation based on co-change relationships: Each commit in the reaction removes usages of the deprecated entity. For each method in which such a reference to the entity is removed, we extract the list of entities that were added at the same time. We consider each of these to be a possible replacement for the entity. For each reacting commit over time, we compute the frequency of each candidate replacement to estimate the probability that each candidate is a valid replacement. Figure 8 shows such a frequency list of replacements, for the deprecation of method *addEntity:*.

```
72% · · · add:
11% · · · addModel:
 9% · · · addObject:
 6% · · · addAll:
 2% · · · addSibling:
```

Figure 8: Frequency of replacements for deprecated method *addEntity:* in the corpus of analyzed projects

Calculating the frequency of each replacements for all studied deprecations yields the following results: In 16% of the cases, the replacement is systematic, that is, one single way of replacing the deprecated entity usage has been applied to all cases. The median of the frequency distribution for the highest-ranked candidate is 60%, the first quartile is 46%, and the third quartile is 80% (see Figure 9). Of course, not all dependencies to a deprecated entity are tackled in an uniform manner: Some dependencies are simply dropped, so the method calls disappear, without replacements; in other cases, several alternatives may exist (as in the *addEntity:* deprecation shown in Figure 8); and some developers may replace the deprecated functionality with home-grown solutions.

However, these results provide evidence that in a large number of cases, a replacement for an API change can be confidently found by looking at the replacements performed by other clients of the framework, as suggested by Schaffer *et al.* [31]. For the rest of the cases, finding an algorithm that would allow the detection of the recommended replacement remains an open problem.

*RQ*6**. How helpful was the deprecation message, if any?**

When deprecating a method, it is customary to provide an indication of a replacement entity to be used instead, if possible. Deprecated methods in Squeak and Pharo often feature a text that describes the replacement. When present, we compared it with the most co-changing entities as detected for the previous research question. After manually inspecting the 80 deprecated methods, we classified the messages in the categories shown in Table 2 (the 14 implicitly deprecated methods had no message). If some messages pointed to a specific replacement method, others pointed to a class or a package instead (demanding further investigation), while some, instead of offering help, were informing the user that the functionality was to be removed, or was “ugly” and should not be used.

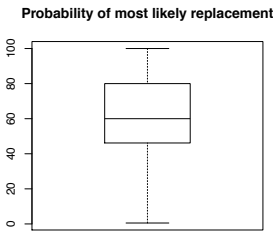

Figure 9: Box plot for the probability of the most likely replacement for the deprecated entity.

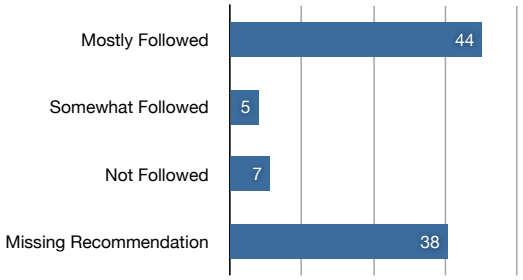

Figure 10: How developers follow the advice in deprecation messages

Figure 10 shows the distribution of the classification for the 94 deprecation messages we studied. In a strong minority of cases (44), the recommendation in the deprecation was the most co-changing entity. In a few cases, it came in second (recommendation somewhat followed, 5). However, in 45 out of 94 cases (47.87%), the recommendation was not clear on how to replace the deprecated entity, or developers chose to bypass the recommendation and use another mechanism to solve their problem (either removing the dependency to the deprecated entity without replacement or using another, not recommended way to replace it) .

These results show that one cannot always rely on a deprecation indication to be helpful. The deprecation may either be absent, not offer concrete advice, or offer advice, that after investigation from the developer, is found to be insufficient.

8. IMPLICATIONS

By presenting the characteristics of the ripple effects (frequency, magnitude, time to react and adapt, kinds of adaptations performed, *etc.*) we provide the ground work for a better understanding of API changes and their ecosystem-wide ripple effects and for future research and development of tools and techniques to support ecosystem evolution. The answers to our research questions allow us to formulate the following implications of our study:

The quality of deprecation guidelines should be improved. In the shorter term, our study provides immediate actionable information: Our investigation of deprecation messages suggests that developers should write more helpful deprecation messages. In a strong minority of cases the deprecation message was absent, letting developers on their own to find a replacement, vague, or—arguably even worse—providing a recommendation that was ultimately not followed by the majority of users of the deprecated entity. API developers should provide concrete, specific instructions on what element of the new API users of the deprecated API should turn to.

Classification	Description
Mostly Followed	The recommendation in the deprecation message was followed in most of the cases (*i.e.,* the recommendation for *addEntity:* in Figure 8, was indeed *add:*)
Somewhat followed	The recommendation in the deprecation message comes second in the list of replacements developers did (*i.e.,* this would be the case if the recommendation for *addEntity:* would have been *addModel:*)
Rarely followed	Usage of the deprecated entity was mostly replaced in a different way than advised
Missing Recommendation	No specific replacement was mentioned in the deprecation message or the deprecations message was missing altogether

Table 2: Classification of the deprecation messages and of how they were followed.

Some API deprecations have a large impact on the ecosystem. The initial goal of the study was to discover whether ripple effects happen at the ecosystem level. To answer this we focused our study on the ripple effects that result from method deprecations and thus, our results are only a lower bound for the actual number of ripple effects in software ecosystems which are due to API changes.

Our study shows that even if only a minority of API deprecations trigger ripple effects in other systems (14% of all the deprecated methods trigger reactions in client projects), some of these can have a large impact if we consider the number of projects and developers that are affected (up to 80 developers, and up to 120 projects). They take time before being noticed, and, once noticed, performing the update itself may not be simple: more than one commit is usually necessary, more than one developer may be involved, and some changes may be overlooked, leaving the project in an inconsistent state until they are addressed. In the face of this, some projects do not react at all, either voluntarily, preferring to freeze their dependencies on a given version, or involuntarily as they are not aware of the changes. This only delays the problem.

Developers do not know how their software is used. The fact that the majority of the deprecated methods are not used by any other project illustrates the situation in which the library developers do not know whether and how their code is used by clients. On the one hand, this is the normal approach to reuse: the developer of the library should not care about how his code is used. On the other hand, we believe that, without excessive effort, infrastructure can be built that would maintain the awareness and enable the querying of inter-project dependencies, and thus pave the way to a situation in which library developers can make better informed decisions about the evolution of their system.

Reactions to API changes can be partially automated. Adapting a system to use a newer version of a framework, or porting it to the other branch of a fork may be costly, and the cost will likely increase over time. However, we observed that similar reactions to a ripple effect happen across projects and this could allow the partial automation of the adaptation. Indeed, strategies can be developed to assist developers impacted by API changes by reusing and analyzing previous reactions to it. The previous reactions can be analyzed qualitatively and quantitatively to help estimate which are the changes required to adapt to a new API but also how much effort it takes to upgrade to using a new API.

9. THREATS TO VALIDITY

9.1 Threats to Construct Validity

The threats to construct validity are related to the quality of the data we analyze, and the manual analysis that was involved.

The Monticello versioning system is distributed. At each commit, a stand-alone snapshot of the source code is stored. However, not every commit is stored on SqueakSource; some are stored as commits in a local repository on the developer's hard drive. As such, there may be gaps in the history of each systems. This artificially inflates the changes between the versions and introduces some imprecisions. Versions with a large amount of changes and/or a large gap between them were filtered out of the manual inspections to reduce noise.

The ecosystem presents some instances of large-scale duplication (around 15% of the code [32]), where packages are copied from a repository to another (*e.g.,*, a developer keeping a copy of a specific library version). This may overestimate the number of projects that react to a given ripple effect, and sometimes makes the project originating the change unclear.

Some method names—for instance simple action verbs such as "move"—are very common. This introduces noise as projects may use unrelated methods bearing the same name. Since Smalltalk is dynamically typed, the problem is made more difficult. Extremely common method names make ripple effects hard—if not impossible—to detect, while we applied extra care while analyzing method names that were less, but still somewhat common. In cases where the noise was too high, we discarded the API change, but underestimate the frequency with which API changes trigger ripple effects as a result.

Our study mostly consider methods and classes that were explicitly deprecated. Other API changes may not have been deprecated beforehand, especially if the developer was not aware of their usage by clients. As such, we may underestimate the amount of ripple effects in the ecosystem. We also underestimate the size of ripple effects since we only consider their immediate propagation, and not transitive changes.

Some methods may have been removed as a consequence of the natural evolution of a project, rather than a reaction to a ripple effect. We tried to account for that during our manual analysis of the changes, when determining if a method is the cause of a ripple effect. In some instances, both use cases may happen at once, which overestimates the importance of some ripple effects.

A significant part of the analysis was performed manually, and is as such error-prone. We made sure to double-check our classification and to discuss differences on a case-by-case basis.

9.2 Threats to External Validity

The threats to external validity are concerned with how generalizable our results are.

We performed the study on a single ecosystem. It needs to be replicated on other ecosystems in other languages in order to characterize the phenomenon of ripple effects more broadly; our results are limited to a single community in the context of open-source software. One aspect that is particular to a Smalltalk ecosystem is the fact that the source code from all the projects was always available. In a different community and ecosystem one would have to take into account that some projects would not have the source code available but only binaries.

The SqueakSource 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 inconsistent updates in a given project. This may be mitigated by infrequent builds, so a replication on such an ecosystem would be very insightful.

Further, the community we picked has a peculiar history as it has experienced a fork of the community during the time it was observed. This may produce exceptional results. In particular, the Pharo subcommunity has a heavy focus on refactoring the core libraries, which may influence the amount of API changes encountered. On the other hand, projects from the Squeak sub-community may understandably not adapt to changes that concern only the Pharo sub-community.

We performed our study mostly on methods and classes that were explicitly deprecated, with a few additional methods and classes we encountered during our analysis. Other ripple effects may propagate as the results of methods which are silently removed or change semantics, and behave differently. We did not notice particular differences between the two kinds of ripple effects, but we will assess whether that is indeed the case by devising algorithms to detect these ripples and integrate more of them in our study.

10. CONCLUSIONS AND FUTURE WORK

This paper presented an empirical study on the actual incidence of API changes, and in particular API deprecations, causing ripple effects in practice. We analyzed the usage of deprecation in an open-source development community, and found that a number of deprecated methods and classes were causing ripple effects, where client projects would have to adapt to the changes in the libraries and frameworks they use. We inspected 113 occurrences of ripple effects in the ecosystem we studied. We reiterate here the most interesting conclusions we derived from the analysis:

1. A number of API changes caused by deprecation can have a very large impact on the ecosystem, either considered in terms of projects or developers that are impacted by the change, or measured by the overall amount of changes.

2. A strong minority of reactions to API changes can remain undiscovered long after the original change is introduced (more than three months). In some cases, adaptations are not done for entire project at once; parts of it remain in an inconsistent state for long periods of time.

3. In a large number of cases, a suitable replacement for a deprecated method can be discovered by analyzing the replacements already performed for that method in the ecosystem.

4. Deprecation messages are not always useful. In nearly half of the cases (48%), the instructions are either absent, unclear, or the developers decide not to take into account the advice that they received.

We expect that there are other API changes in our ecosystems which are not using the deprecation mechanism. Thus, the results on the impact of API changes we are reporting represent only a lower bounds on the actual magnitude of the phenomenon. In future work, we will develop algorithms to detect additional ripple effects in the ecosystem and corroborate the results we presented here. Once that step is complete we will work on building tools that monitor the ecosystem and support its evolution in the presence of ripple effects.

Acknowledgments. We thank Niko Schwarz for feedback on early versions of the manuscript. We thank the administrators of Squeaksource, Adrian Lienhard and Lukas Renggli, for providing us access to the data. Romain Robbes is partially funded by FONDECYT Project No. 11110463, Chile; Mircea Lungu is funded by the Swiss SNF Project No. 200020-131827; David Röthlisberger is funded by the Swiss SNF Project No. PBBEP2 135018.

11. REFERENCES

[1] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[2] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Proceedings of OOPSLA 2006*, pages 681–682, 2006.

[3] S. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance*, 13(4):263–279, 2001.

[4] L. C. Briand, J. Wüst, and H. Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *ICSM*, pages 475–482, 1999.

[5] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of ICSE 2008*, pages 481–490, 2008.

[6] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of ECOOP 2006*, pages 404–428, 2006.

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

[8] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proceedings of ICSE 2007*, pages 427–436, 2007.

[9] D. Dig, S. Negara, V. Mohindra, and R. E. Johnson. Reba: Refactoring-aware binary adaptation of evolving libraries. In *Proceedings of ICSE 2008*, pages 441–450, 2008.

[10] T. Ekman and U. Asklund. Refactoring-aware versioning in eclipse. *Electr. Notes Theor. Comput. Sci.*, 107:57–69, 2004.

[11] 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*, 2008.

[12] J. O. Graver. The evolution of compiler an object-oriented framework. *Software Practice and Experience*, 22(7):519–535, July 1992.

[13] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6, 000 projects: lightweight cross-project anomaly detection. In *Proceedings of ISSTA 2010*, pages 119–130, 2010.

[14] F. M. Haney. Module connection analysis: a tool for scheduling software debugging activities. In *Proceedings of AFIPS 1972*, pages 173–179, 1972.

[15] J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *Proceedings of ICSE 2005*, pages 274–283, 2005.

[16] R. Holmes and R. J. Walker. Customized awareness: recommending relevant external change events. In *Proceedings of ICSE 2010*, pages 465–474, 2010.

[17] O. Hummel, W. Janjic, and C. Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25(5):45–52, 2008.

[18] S. Jansen, S. Brinkkemper, and A. Finkelstein. Providing transparency in the business of software: A modeling technique for software supply networks. In *Virtual Enterprises and Collaborative Networks*, pages 677–686, 2007.

[19] C. Jergensen, A. Sarma, and P. Wagstrom. The onion patch: migration in open source ecosystems. In *Proceedings of ESEC/FSE 2011*, pages 70–80, 2011.

[20] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of ICSE 2009*, pages 309–319, 2009.

[21] M. Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, October 2009.

[22] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes. The Small Project Observatory: Visualizing software ecosystems. *Science of Computer Programming*, 75(4):264–275, 2010.

[23] M. Lungu, R. Robbes, and M. Lanza. Recovering inter-project dependencies in software ecosystems. In *Proceedings of ASE 2010*, pages 309–312, 2010.

[24] D. G. Messerschmitt and C. Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. The MIT Press, 2005.

[25] A. Mockus. Large-scale code reuse in open source software. In *Proceedings of FLOSS 2007*, page 7, 2007.

[26] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Proceedings of MSR 2009*, pages 11–20, 2009.

[27] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[28] J. Ossher, S. K. Bajracharya, and C. V. Lopes. Automated dependency resolution for open source software. In *Proceedings of MSR 2010*, pages 130–140, 2010.

[29] S. P. Reiss. Semantics-based code search. In *Proceedings of ICSE 2009*, pages 243–253, 2009.

[30] R. Robbes and M. Lanza. Versioning systems for evolution research. In *Proceedings of IWPSE 2005*, pages 155–164, 2005.

[31] T. Schäfer, J. Jonas, and M. Mezini. Mining framework usage changes from instantiation code. In *Proceedings of ICSE 2008*, pages 471–480, 2008.

[32] N. Schwarz, M. Lungu, and R. Robbes. On how often code is cloned across repositories. In *Proceedings of ICSE 2012, NIER Track*, pages 1289–1292, 2012.

[33] K. Taneja, D. Dig, and T. Xie. Automated detection of api refactorings in libraries. In *Proceedings of ASE 2007*, pages 377–380, 2007.

[34] L. Tokuda and D. S. Batory. Evolving object-oriented designs with refactorings. *Autom. Softw. Eng.*, 8(1):89–120, 2001.

[35] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proceedings of ASE 2006*, pages 231–240, 2006.

[36] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. Aura: a hybrid approach to identify framework evolution. In *Proceedings of ICSE 2010*, pages 325–334, 2010.

[37] S. Yau, J. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *Proceedings of COMPSAC 1978*, pages 60 – 65, 1978.

[38] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of ESEC/FSE 1999*, pages 253–267, 1999.