A Study of Ripple Effects in Software Ecosystems (NIER Track)

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

Mircea Lungu
SCG—University of Bern
lungu@iam.unibe.ch

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 early results of such an empirical study of API changes that led to ripple effects across an entire ecosystem. Our case study subject is the development community gravitating aroung the Squeak and Pharo software ecosystems: six years of evolution, nearly 3,000 contributors, and close to 2,500 distinct systems.

Categories and Subject Descriptors

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

General Terms

Measurement, Design

Keywords

Software Ecosystems, Mining Software Repositories, Empirical Studies

1. INTRODUCTION

In an email from July 7th, 2007, a user of Seaside, a popular web application framework, asks:

> I noticed that the Seaside 2.6 dialog classes listed below are not in Seaside 2.8a1.390. I have not been paying attention to Seaside 2.8 so don't know much about its status. I am wondering if these classes have been dropped, have not been ported to 2.8 or does their functionality exists elsewhere?

ICSE '11, May 21–28 2011, Honolulu, Hawaii

Soon after, the Seaside maintainers answer[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.

This situation is surprisingly commonplace. A recent email from the Moose mailing list—a reverse-engineering environment—asks about the fate of a class in the Mondrian visualization framework[2]:

> [...] where is MOLabelShape, why was it deleted?? I use it and now is gone!!! I even had a specialization of it [...]

These examples illustrate the consequences of API changes in frameworks and librairies when the developers are not always conscious of how their frameworks are used. When a part of the interface of a framework changes, ripple effects can occur in clients as they react to changes [4, 11]. These are made worse by the loose communication between users and providers of a framework: An API-breaking change may take months to get noticed. Several works in recent years have proposed solutions to this problem, showing that it is known to the community [3, 1, 9, 10, 5].

However, there has been no large-scale study on the actual impact of the problem in practice. The only studies available are performed at smaller scales, studying frameworks in isolation without taking their clients into account [2]. To perform such a study, one must raise the level of abstraction to the one of software ecosystems.

Software ecosystems are collections of projects that are developed and evolve together in the context of an organization or a development community [6]. The projects interact with one another in various ways: the developers that work on one project get involved in other projects, bringing along their legacy of experience; projects branch from one another and evolve simultaneously—sometimes incorporating changes in other branches, sometimes diverging; last but not least, multiple projects reuse functionality that other projects, libraries and frameworks provide.

In this work we present early results of an analysis of such a software ecosystem. The subject of our analysis is the ecosystem that emerged around the Squeak and Pharo development communities, which use a common code hosting and versioning service, the squeaksource.com website.

Structure of the paper. We describe the challenges that must be addressed by a ripple effect detection process (Section 2). We then introduce our ecosystem model and the way we model ripple effects (Section 3). We continue with a presentation of the qualitative evidence we have gathered during our preliminary study (Section 4). We then discuss future plans and conclude the paper (Section 5).

[1]Entire exchange available at:`http://bit.ly/gnwNfV`.
[2]`http://bit.ly/hiTeN8`

2. MINING FOR RIPPLE EFFECTS

At the level of the ecosystem, a ripple effect is a change to a software system's API which originates in one system and propagates to other systems. To detect 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) gathering the data, (2) building a model of the ecosystem, and (3) generating a list of candidate ripple effects and their impact on clients. This process is subject to a set of challenges:

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

2. **Modelling 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—evolving systems depending on one another—is built, the ripple effects themselves—original API change and reactions and adaptation of the clients over time—must be detected. In our analysis, we have recovered an initial set of ripple effects based on initial heuristics (e.g., deprecated methods will cause ripple effects), and use that set of ripple effects as a basis to define algorithms to detect other ripple effects.

3. MODELLING EVOLVING ECOSYSTEMS

To provide a meta-model of software ecoystems suitable to the detection of ripple effects, we built on the experience we have with two other ecosystem meta-models: RevEngE[6] and Ecco[7]

Ecco is a lightweight, language-independent representation of the data in ecosystem snapshots, providing full (language-dependent) access to the details on demand; we have already introduced *Ecco* in our earlier work on extracting dependencies between the systems in an ecosystem. Ecco's 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. An entity can be both required and provided. The entities are identifiers of classes and methods defined and/or used in each system. Based on these lists of provided and required entities one can recover dependencies between systems: if system B requires a set of entities and system A is the only provider of these entities in the ecosysem, we can infer that B depends on A. In previous work, we performed an empirical evaluation of multiple techniques for recovering inter-system dependencies [7].

To detect dependencies between systems it is sufficient to flatten the entire evolution of a system—considering the provided and required entities to be the union of all provided and required entities in any version. To perform evolutionary analysis one needs to model the individual versions of a system. The RevEngE meta-model supports evolution; in our previous work we have proposed it as a generic ecosystem meta-model which supports evolutionary analysis. The previous instantiations of the model used a full FAMIX model of every version of the individual systems. To detect ripple effects, we need to model the entire history of every version of every system in the ecosystem and for this RevEngE is too heavyweight.

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. 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 (one of each in the typical case, but more in case of forks, merges, or system splits). Our model—fully built—is nearly two orders or magnitude smaller than the initial dataset but allows on-demand access to it if required.

4. RIPPLE EFFECTS IN PRACTICE

As a case study we use the ecosystem built around the Squeak and Pharo open-source development communities (Pharo is a fork of Squeak). It is hosted by the Squeaksource [3] 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 December 2010, Squeaksource hosts 2463 systems in which 2924 contributors performed more than 92,000 source code commits. The combined size of all the versions is more than 9GB of compressed source code (Monticello stores versions as zip files).

Among the systems hosted by Squeaksource one can find multiple widely-used frameworks and tool suites: the Seaside rapid web application framework, the Moose reverse engineering environment, the Croquet 3D environment, the Magma, GOODS and GLORP databases, the Pier content management system, the Magritte meta-description and UI generation framework, the Swazoo web server, the Mondrian visualization toolkit, etc.

The reason for the choice of the Squeak and Pharo ecosystem are threefold: (1) the data is easy to access and delimit—practically Squeaksource stores all the open-source code produced by members of the community; (2) the data is relatively easy to process, as the versioning system in use, Monticello, stores program entities instead of text files [8]; and (3) the recent fork in the community provides us with an opportunity to better understand this phenomenom as it happens at such a large scale.

4.1 Types of Ripple Effects

In our model, ripple effects are sequences of changes across systems in the ecosystem, triggered by an earlier change to an entity they require. We only consider syntactic changes and not semantic changes: even detailed static analysis may not always be able to infer that the semantics differ between two versions of a method.

When looking at the syntactic changes we consider two types of entities, 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:

- *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.

- *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. A special case of this is annotating a method or class as *deprecated*, tagging it for later removal.

[3] `http://www.squeaksource.com`

Origin System	Deprecated Method	Systematic replacement (if stated)	# Aff. Syst.	# Aff. Dev.
System-Support	`Author initials`	`Author fullName`	120	110
Seaside	`WACanvas bold:`	`WACanvas strong:`	61	59
Seaside	`WASession registerForBackgracking:`	`(...) states`	43	47
Famix-Core	`FAMIXNamedEntity packagedIn`	`FAMIXNamedEntity parentPackage`	22	11
Kernel	`ClassDescription metaclass`	`ClassDescription theMetaClass`	31	34
System-Support	`Utilities authorInitials`	`Author initials`	37	45
Kernel	`Object isKindOf:orOf:`		12	24
Collections-Text	`Text isoToSqueak`		18	22

Table 1: Eight ecosystem ripples that resulted from API deprecation

- *Rename of provider*. This renames a provided entity. It can be seen as a composition of an addition and a 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. Our search for ripple effects will start with looking for ripples determined by method removals and renames.

4.2 Impact of Deprecation in the Ecosystem

To obtain a first glimpse of the ripple effects and their occurrence in the case-study ecosystem we looked at the effects of deprecating methods in the case-study ecosystem over its entire lifetime. These ripple effects are easy to detect and inspect because the API changes are explicitly marked as deprecated by the developers.

We analysed all the changes that incorporated usage of deprecated methods to build the set of method deprecations that result in ripple effects. These methods are: `deprecated:` (used in Squeak and Pharo), `deprecatedApi:` (used in Seaside), and some others. We reviewed each of the changes and recorded the deprecated methods, and, if present in the error message or the comments, which method to use instead. Ecco allows us to efficiently query for ripple effects across system versions. Searching for commits relevant to a set of changed items of interest (classes or methods added or removed between two versions) takes seconds.

Table 1 presents 8 of the several dozen deprecation instances that we inspected in our study. For each API deprecation, the table indicates: the originating system, the name of the deprecated method and its replacement (if mentioned in the deprecation rationale), and the numbers of systems and developers affected by the change. These numbers are upper-bound estimates as we report on all the users of the method over the studied interval (including new users that appeared after the API change). Based on the table and further inspection, we observe the following:

- Some API changes can have a wide impact. The reason for this is that all five systems in the table are systems which are heavily depended on: Parts of the base system (Kernel, System-Support), standard APIs (Collections), and popular frameworks (Seaside, Famix-Core).

- Different deprecations require different degrees of complexity to adapt. Some deprecation messages directly mention the new method to call; ripple number 3 requires a more complex change (instead of calling a method, a new method must be overriden); ripple number 6 involves both a class and a method renaming; ripples 7 and 8 do not provide specific instructions: the developers are "on their own".

- Some API refactorings happen in multiple stages: ripples 1 and 6 affect the same methods.

These results provide initial evidence that ripple effects caused by API changes do happen in practice at the ecosystem level and can affect large numbers of developers and systems.

4.3 Four Months in the Life of a Ripple Effect

In this section, we elaborate on one of the ripple effects listed above, specifically, the one which is the result of the renaming of the `packagedIn` method to `parentPackage` in Famix-Core.

To support our analysis, we implemented an interactive tool—the Ecosystem Ripple Inspector—to support detailed inspections of ripples; it uses Ecco's detail-on-demand approach to allow the exploration of the effects of the ripple in terms of actual changes to the source code. The tool also implements an ecosystem viewpoint [4] dubbed the Ripple Propagation Viewpoint which captures how a ripple propagates across systems in the ecosystem chronologically.

Figure 1 presents the ripple propagation viewpoint as implemented in the Ripple Inspector. In the figure time flows left to right; systems are ordered from top to bottom; the top system is the originator of the API change; the other systems are affected by the ripple effect. For each system, we represent the succession of versions including branching and merging. In all the systems, we highlight the versions in which one of the methods involded in the original API change is added or removed as required or provided: these versions are part of the ripple effect.

The legend in the figure explains the visual conventions for highlighting the affected versions. We represent each version as a bipartite rectangle: the left part corresponds to the removed method and the right part to the added method. In both cases addition is green, and removal, orange. We highlight two types of combined changes: removal of old and addition of new method is magenta (reaction to the ripple) and the reverse is red (reverting the reaction). We also highlight the original API change in cyan. By analyzing the figure, we make the following observations:

- Ripples can appear long after the original change is introduced. The changes highlighted by mark d appear 4 months after the original change. In other systems one can still see removals of the old method months after the original API change.

- In one instance we see a ripple followed a few days later by the opposite change (mark a), which means a revert to depending on the old version of the provider system. The final update occurs at mark d.

- Sometimes the change does not propagate at once in the entire client system. Instead, the client remains in an inconsistent

[4] A visual representation of a particular aspect of the ecosystem [6]

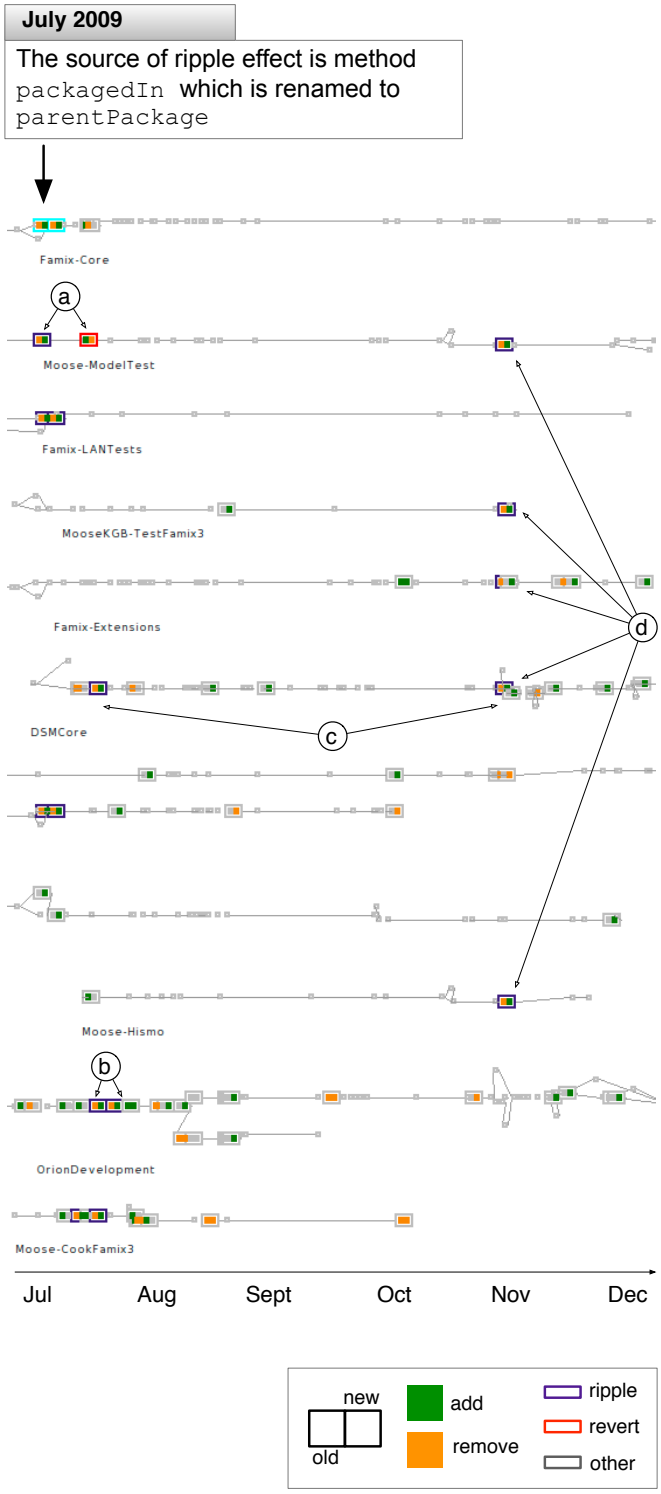

Figure 1: Ripples generated by renaming *packagedIn* to *parentPackage* in Famix-Core

state until later. This is the situation in DSMCore, where there are two ripples at a few months distance (mark *c*). This is also the case in another system, OrionDevelopment, where there are two successive ripples a few days apart from each other (mark *b*).

To sum up, even a simple change such as a method rename can take a long time—months—to propagate to all interested parties. Such a change can leave projects in an inconsistent state if care is not taken to properly update the clients.

5. CONCLUSIONS AND FUTURE WORK

In software engineering, managing API changes to libraries and frameworks is a known problem for which several solutions exist in the literature. However, we are not aware of any large-scale studies of actual occurences of the ripple effects caused by API changes performed so far. This paper presented initial steps in this direction: we gathered data about an open-source development community, built an infrastructure to detect and analyze ripple effects due to API changes as they appear in the ecosystem, and presented examples of actual ripple effects that occured in the ecosystem. We have shown that ripple effects do happen in practice, and that the update process is far from smooth.

We expect that there are other ripple effects in our ecosystems which are not using the deprecation mechanism. In future work, we will develop algorithms to detect all the ripple effects in an ecosystem and present quantitative results. Once that step is complete we want to work on building tools that monitor the ecosystem and support its evolution in the presence of ripple effects.

6. REFERENCES

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

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

[3] 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.

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

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

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

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

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

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

[10] 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.

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