Assessing and Improving Rules to Support Software Evolution

THÈSE

présentée et soutenue publiquement le 4 November 2014

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

André Cavalcante Hora

Composition du jury

Président :	Laurence DUCHIEN
Rapporteur :	Alexander SEREBRENIK, Salah SADOU
Examinateur :	Laurence DUCHIEN, Rainer KOSCHKE
Directeur de thèse :	Stéphane DUCASSE
Co-Encadreur de thèse :	Nicolas ANQUETIL

Acknowledgments

Abstract

Software systems evolve by adding new features, fixing bugs or refactoring existing source code. During this process, some problems may occur (*e.g.*, backward-incompatibility, missing or unclear method deprecation) causing evolving systems and their clients to be inconsistent or to fail, decreasing code quality. As nowadays software systems are frequently part of bigger ecosystems, such problems are even harder to handle because the impact may be large and unknown.

One solution to deal with such maintainability problems is the usage of rules to ensure consistency. These rules may be created by experts or extracted from source code repositories, which are commonly evaluated in small-scale case studies. We argue that existing approaches lack of: (i) a deep understanding of the benefits provided by expert-based rules, (ii) a better use of source code repositories to extract history-based rules, and (iii) a large-scale analysis of the impact of source code evolution on the actual clients.

In this thesis we propose to analyze and improve rules to better support developers keeping track of source code evolution. We cover three aspects:

- The benefits provided by expert-based rules: we report on an investigation of rules created based on expert opinion to understand whether they are worthwhile to be adopted given the cost to produce them.

- The improvement of history-based rules: we propose two solutions to extract better rules from source code history.

- The impact of source code evolution on a software ecosystem: we undergo an investigation, in a large-scale ecosystem, on the awareness of the client systems about source code evolution.

We evaluated the proposed approaches qualitatively and quantitatively in real-world case studies, and, in many cases, with the help of experts on the system under analysis. The results we obtained demonstrate the usefulness of our approaches.

Keywords: software evolution, mining software repositories, API evolution, software ecosystems, empirical software engineering

Résumé

Les systèmes logiciels evoluent continuellement pour ajouter de nouvelles fonctionnalités, corriger des bugs ou refactoriser du code source existant. Durant ce processus, certains problèmes peuvent survenir (par exemple, le manque de rétro-compatibilité, l'absence ou l'imprécision des deprecations explicites) provoquant l'inconsistance ou l'échec des systèmes en évolution et avec leurs clients, ce qui aboutit finalement à une baisse de la qualité du code. Comme de nos jours les systèmes logiciels font souvent partie de plus grands écosystèmes, ces problèmes sont encore plus difficiles à gérer car l'impact peut être grand et inconnu.

Pour faire face à ces problèmes de maintenabilité et garantir la consistance du code source, il est possible d'utiliser des règles. Ces règles peuvent être créées par des experts ou extraites de précédentes versions du code source. Elles sont couramment évaluées dans des études de cas à petite échelle. Nous soutenons que les approches existantes : (i) n'analysent pas précisément les avantages des règles créées par des experts; (ii) gagneraient à mieux utiliser les dépôt de codes sources pour extraire des règles basées sur l'historique, et (iii) devraient analyser à grande échelle et sur des cas réels l'impact de l'évolution du code source sur les clients.

Dans cette thèse, nous proposons d'analyser et d'améliorer les règles pour aider les développeurs à mieux suivre l'évolution du code source. Pour cela, nous étudions trois aspects différents :

- Les avantages prévus par les règles créées par des experts : nous analysons précisément ces règles pour comprendre si elles valent la peine d'être adoptées malgré le coût pour les produire.
- L'amélioration des règles basées sur l'historique : nous proposons deux solutions pour extraire de meilleures règles à partir du dépôt de codes sources.
- L'impact de l'évolution du code source sur un écosystème logiciel : nous étudions les conséquence de l'évolution de code source sur des systèmes clients dans le contexte d'un écosystème de grande échelle.

Les approches proposées dans cette thèse ont été évaluées qualitativement et quantitativement avec des études de cas issues du monde réel. Pour plusieurs de ces études, nous avons pu bénéficier de l'aide d'experts sur les systèmes en cours d'analyse. Les résultats que nous avons obtenus démontrent l'utilité de nos approches.

Mot clés: évolution du logiciel, exploration de données, évolution d'API, écosystèmes de logiciels, génie logiciel empirique

Contents

INTRODUCTION

Contents

1.1   Context

In software development, change is the only constant. Software systems evolve by adding new features, fixing bugs and refactoring existing source code. A software system that does not evolve may become outdated and irrelevant [Leh96]. In practice, most of the effort during a software system lifecycle is spent to support its evolution [Som10], which is also responsible for up to 80% of the total cost of the development [Erl00].

Nowadays, software systems are frequently part of bigger ecosystems [Lun09]. These ecosystems usually exist in large companies, organizations, or open-source communities such as Eclipse, Pharo, Android, Apache, Linux distributions, among others. In such environment, software systems are part of a collection, often interrelated among one another [BCP+13].

As any other software system, the ones composing an ecosystem must evolve over time. In this context, the evolution of core applications (*e.g.*, frameworks and libraries) is likely to impact many client systems. That it to say, the evolution of certain applications may trigger a propagation of changes over the ecosystem, known as ripple effect.

Robbes *et al.* [RLR12] found, for example, that in an ecosystem centered on a dynamic programming language, a simple API change potentially affected thousands of clients, but in practice only a minority of these clients were aware and reacted. In Eclipse, 42% of the methods in version 1.0 were not in version 2.0 [MWZ11], consequently, when migrating to the newer version, Eclipse's clients are clearly impacted by these changes.

In some cases it is hard to predict how a software system is used by clients. Developers of a large corporation pointed to us in a discussion that sometimes changes in their core systems would break other systems that they were not expecting. That is to say, the extension of the impact may be large

and sometimes unknown. This leads to the question: how can we ensure that software evolution is consistently propagated both to evolving systems and their clients?

In short, software evolution is naturally a complex task. This may be aggravated in environments where systems are very dependent among one another such as software ecosystems. Developers need help maintaining their systems.

1.2   Problem

To facilitate the time-consuming task of following the evolution of a system, some good practices should be adopted. For example, frameworks and libraries should be backward-compatible, *i.e.*, they should not cause a client built with an older version to fail under a newer version. Developers should also always deprecate methods with helpful replacement messages, before removing them.

In practice, many problems may occur mainly due to the large size and the amount of developers involved in the development process. Below we list some concrete problems detected by the literature:

- Researchers have found that frameworks and libraries are backward-incompatible [BTF05, DJ06, WGAK10], *i.e.*, they do not follow the basic principle of providing backward-compatibility for clients.

- Recommendations of deprecation messages are often missing or unclear on how to replace deprecated methods [RLR12]. Moreover, public methods may be simply removed without deprecation [HRA+15].

- Even if client systems should only use public interfaces, developers often use internal and undocumented methods of frameworks to access functionalities not available in the public interfaces [BR06, DR08, Bus13, BSvdB13]. After evolution, these internals may cause client systems to fail.

These maintainability problems show the difficulty of software evolution, in particular to keep consistency of source code evolution. It is important to ensure that source code evolution is correctly propagated.

1.3   Ensuring Consistency with Rules

To deal with such maintainability problems, approaches have been proposed to support software evolution and reduce the efforts of developers.

One approach commonly adopted is the usage of rules to ensure consistency. These rules aim to point defects in source code [LZ05, WH05, KPW06, NNP$^+$10, SSPR12], ensure the use of best coding practices and conventions [Cop05, HP04, RBJ97, RDGN10], or describe classes and methods replacements [DR08, SJM08, WGAK10, MWZM12]. Overall, the literature provides two solutions to obtain these rules. Unfortunately, there is a lack in each solution as to support developers in their maintenance tasks, as described below.

A first solution is to obtain the rules with the help of experts of the system under analysis (we call these *expert-based* rules). These rules can focus on important problems of the system since they are created based on the previous experience of the developer. However, they must be manually defined by an expert, which is costly. In practice, getting access to experts and capturing their knowledge into rules is a difficult task that is rarely undergone. *While the study of generic rules that are not created by experts is well covered by the literature (e.g., [BvdB06, WDA$^+$08, CMSV12, BM08, BM09]), it is not clear whether expert-based rules are worthwhile to be adopted given the cost to produce them.*

Apart of being costly, it is expected that experts may not have the complete knowledge of the whole system or they may be no longer available. So, the reduction of the dependence on experts would be relevant to avoid those possible issues. Therefore, as a second solution to obtain rules, previous studies propose to use the source code history as a source of information to extract them (we call these *history-based* rules). These rules can be inferred from source code changes found in code repositories. *In this context, existing approaches present some limitations. First, current approaches [LZ05, WH05, KPW06, NNP$^+$10, SSPR12] do not properly focus on the discovering of system-specific conventions in source code usage. Second, related studies [DR08, SJM08, WGAK10, MWZM12] do not cover all the cases of classes and methods replacements.*

In common, those approaches to support software evolution are often evaluated in small-scale case studies or individual client systems; the literature still lacks of large-scale evaluation. As we explained before, a software system is frequently part of a bigger *software ecosystem*. The analysis of software evolution and its impact on ecosystems can help developers to better understand its real extension and what can be done to alleviate this impact. *Ecosystem impact analysis is a recent field of research, with restricted studies in terms of case study size [MRK13] and API analyzed type [RLR12]. The real impact of source code evolution on a large-scale case study is not known.*

In summary, existing approaches lack of: (i) a deep understanding of the benefits provided by expert-based rules, (ii) a better use of source code repositories to extract history-based rules, and (iii) an analysis of the impact of source code evolution on the actual clients.

1.4   Our Approach in a Nutshell

In this thesis we propose to analyze and improve rules to better support developers keeping track of source code changes. Such rules may ensure, for example, the usage of better APIs to improve performance, legibility, portability, etc. We cover three aspects: the benefits provided by *expert-based* rules, the improvement of *history-based* rules, and the impact of source code evolution (in the form of rules) on a *software ecosystem*.

Analyzing the benefits provided by expert-based rules. Existing approches to support the analysis of generic rules indicate that these rules do not prevent the introduction of defects in software. This may occur because the rules are too generic and do not focus on specific problems of the software under analysis. We undergo an investigation of rules created based on expert opinion to understand whether such rules are worthwhile enforcing, given the cost to produce them, in the context of defect prevention.

Improving history-based rules. Current approaches related to mining rules from source code repositories do not properly use these repositories to extract history-based rules. We propose two solutions in order to benefit from previous source code changes and generate better rules for developers. As a first solution, we extract rules using patterns, considering the spread of the same source code change over time to detect system-specific conventions. As a second solution, we extract rules using data-mining to produce more flexible rules and help client systems. We validate both approaches with the help of experts on the systems under analysis.

Analyzing the impact of source code evolution on a software ecosystem. An ecosystem analysis allows one to better understand the real impact of source code changes and how it could be alleviated. We undergo an investigation, in a large-scale ecosystem, on the awareness of the client systems about source code changes. We answer research questions regarding the frequency, magnitude, duration, extension, and consistency of such changes in the ecosystem.

1.5   Contributions

The main contributions of this thesis can be summarized as follows:

- **Contribution 1**. We provide new experiments on the lack of relationship between violations generated by generic rules and defects as well as novel experiments on the relationship between expert-based rules and defects [HADA12].

- **Contribution 2**. We provide two novel approaches to extract history-based rules from code repository to better support the evolving system

and the clients [HADV13, HEA$^+$14, HAE$^+$15b, HAE$^+$15a].

- **Contribution 3**. We provide a large-scale study, at the ecosystem level, to understand to which extent client developers are impacted by source code evolution [HRA$^+$15].

1.6 Structure of the Thesis

Chapter 2: State of the Art

This chapter presents the related work in the context of expert-based and history-based rules as well as ecosystem analysis.

Chapter 3: Benefits of Expert-based Rules

This chapter describes our approach to analyze the benefits provided by expert-based rules. We undergo an investigation of rules created based on expert opinion to understand whether such rules are worthwhile enforcing in the context of defect prevention.

Chapter 4: Supporting System-specific Conventions with History-based Rules

This chapter describes our first approach to extract history-based rules to detect system-specific conventions. We extract data from incremental revisions in source code history, and the rules are based on predefined patterns that ensure their quality. We validate our approach on open-source systems with the help of an expert, which is important to provide assessment about the rules.

Chapter 5: Supporting Client Systems with History-based Rules

This chapter describes our second approach to extract history-based rules. Rules are also mined from source code history but using data-mining instead of predefined patterns, which allows the generation of more flexible rules. The approach is evaluated on five open-source systems, and the rules are assessed with the help of experts.

Chapter 6: Impact of Software Evolution on Ecosystems

This chapter presents our analysis of the impact of source code changes on a community of developers. We cover the Pharo ecosystem, which has six years of evolution, about 3,600 distinct systems, and more than 2,800 contributors.

Chapter 7: Conclusion

This chapter concludes the thesis and presents future work. Finally, it also presents the collaboration work with the ASERG/UFMG group and with the Siemens Research & Technology Center.

STATE OF THE ART

Contents

2.1 Introduction

In this chapter we present the current approaches to support software evolution, focusing on the approaches that use rules, but not limited to them. We organize this chapter in three sections: (i) analysis of generic and expert-based rules, (ii) extraction of history-based rules, and (iii) analysis of software ecosystems.

We show that there are lacks in all approaches as to support the developers. We argue that: first, many related work is dedicated to study generic rules and there is little focus on the analysis of expert-based rules; second, existing work about history-based rules does not properly support evolving systems nor clients; and, finally, ecosystem impact analysis is a recent field of research, with restricted case study size and API analyzed type.

2.2 Analysis of Generic and Expert-based Rules

A rule is a specific statement that describes an action to follow when you write code, or a description of a particular solution to a commonly encountered problem in software development[1]. Rules focus on a particular aspect of standards compliance or on an accepted practice to deal, for example, with bad code practices, performance or security issues, among others. They are used to ensure source code consistency, and their final goal is to ensure software quality.

Overall, rules can be either generic or system-specific. Generic rules are created by non-experts on the system under analysis and the same rule can be used by distinct systems. They are more commonly found in static analysis tools such as PMD [Cop05], FindBugs [HP04] and SmallLint [RBJ97]. For example, "checking variables and methods length", "detecting nesting with

---

[1]IBM documentation: `http://goo.gl/Wipppm`

more than one *if* statement", or "searching an assignment that has no effect"
are practices that can be ensured for any system.

In contrast, system-specific rules are created by experts on the system
under analysis (we call these *expert-based* rules), targeting a specific system.
For example, checking the correct API to be used, or checking methods that
should be used/defined together, etc., in the context of the system under
analysis.

As these rules are intended to improve code quality, a solution to evaluate
them is to verify whether they are likely to point to defects in source code.
In this context, Wagner *et al.* [WDA$^+$08] analyze two Java tools (FindBugs
and PMD) on two different software projects, to evaluate their use in defect
detection. The authors could not confirm the use of rules provided that such
tools to detect defects. Basalaj *et al.* [BvdB06] study the link between C++
generic violations and defects for 18 different projects and found a correlation
for only 12 out of 900 rules.

Couto *et al.* [CMSV12] state that overall there is no correspondence be-
tween the violations raised by FindBugs and the methods changed to remove
defects. Boogerd *et al.* empirically assess the relation between violations of
rules raised by MISRA C and defects in two industrial cases [BM08, BM09].
They have found that 10 out of 88 rules in one case study, and 12 out of 72
in another case study were significant predictors of defect location. However,
from such significant rules, both case studies overlapped only on one rule.

Even though distinct static analysis tools have been analyzed covering
different programming languages, those studies agree that such generic rules
do not prevent the introduction of defects in software. The results suggest the
importance of tailoring such rules to the system under analysis [BM09].

In this context, Renggli *et al.* [RDGN10] advocate the use of expert-based
rules to check and maintain system-specific best practices. Their empirical
validation demonstrates that expert-based rules significantly improve code
quality when compared with generic rules. The demonstration is done in two
long term evolution case studies. The authors also provide a survey on the
usefulness of such rules according to 16 experienced developers. They have
found that all developers that use the expert-based rules on a regular basis
found them useful; 69% of the developers stated that the expert-based pro-
duce more useful results than the generic ones while the other 31% could not
see a difference. Moreover, 81% of the developers stated that the expert-based
rules produce relevant results that help them to detect critical problems in
their application. Relating the violation reported by these rules with defects
is not in the scope of their research.

Summary

The analysis of the possible correlation between generic rules and defects is well covered by previous studies. In contrast, the study of system-specific rules created by experts for defect prevention is not yet covered, *i.e.*, it is not clear whether such rules are worthwhile to be adopted given the cost to produce them. In Chapter 3 we study the relation between such expert-based rules and defects to better understand whether these rules are worthwhile in such contexts.

2.3 Extraction of History-based Rules

In practice, getting access to experts and capturing their knowledge into rules is a difficult task since they need to be manually defined. Moreover, it is expected that experts may not have the complete knowledge of the whole system or they may be no longer available. Thus, the reduction of the dependence on experts would be relevant to avoid those possible issues.

As a solution, related studies propose to analyze the source code history to extract rules. We separate these researches in two groups, the first one focuses on system-specific analysis whereas the second is more general in the sense that it focuses on helping client systems to follow API evolution.

2.3.1 System-specific Analysis

Some studies have been proposed to learn from bug-fix changes found in code repositories. Kim *et al.* [KPW06] propose to discover system-specific defects by analyzing the diffs found in bug-fixes changes. Such changes are stored in a database (called memories) that can be then accessed to perform defect detection and change suggestion. Nguyen *et al.* [NNP+10] aim to discover recurring bug-fixes by analyzing two system versions. Based on graph-based representation, they recognize recurring bug-fixes and recommend fixes. Williams and Hollingsworth [WH05] focus on discovering violations likely to be real defects by mining code history. The authors investigate a specific type of violation (checking whether *returned value* is tested before being used) that is more likely to happen in C programs. The study improves bug-finding techniques by ranking violations based on past source code changes. Sun *et al.* [SSPR12] propose to extend a commercial static analysis tool by discovering specific defects. Such work focuses on mining a graph, with data or control dependences, to discover specific problems.

Mileva *et al.* [MWZ11] focus on discovering changes that must be systematically applied in source code. These changes are obtained by comparing two versions of the same system, determining object usage, and deriv-

ing patterns. By learning systematic changes, they are able to find places in source code where changes are not correctly applied. Livshits and Zimmermann [LZ05] propose to discover system-specific usage patterns over code history. Results show that the usage patterns such as method pairs (*e.g.,*lock() must happen with unlock()) can be found in practice. In addition, other studies propose to recover rules from execution traces [LKL08, LRRV12]. These studies extract rules via dynamic analysis of a single system version to produce temporal rules (*e.g.,* every call to m1() must be preceded by a call to m2()).

As another solution to obtain more focused rules, some studies propose to filter rules provided by static analysis tools. Such studies are based on the fact that a significant percentage of violations reported by these tools are false positives [KE07, RDGN10, FSV11]. In that respect, Kim *et al.* [KE07] help developers to discover the "best" rules by looking at source code history to detect which rule violations were more often fixed in the past. With such information in hands, the existing rules can be then ranked, *i.e.*, the more a rule is fixed, the more it is important for a given system. In a related work, Araujo *et al.* [FSV11] aim to discover the categories of rules (*e.g.*, correctness or performance) that make sense for a system under analysis. Yet, rules obtained with these approaches will remain generic and will not focus on specific problems of the system under analysis.

Motivating Examples

We present two concrete and real-world examples of system-specific rules that would be helpful to developers, but that are not yet covered by the literature.

First, in Apache Ant[2], a convention stating a new way to close files, *i.e.*, calls to InputStream.close() should be replaced by calls to FileUtils.close(*), was introduced in the system in 2004 to improve maintenance, centralizing the knowledge on closing files. After the addition, this specific convention continued to be occasionally applied in source code in the following *six years* by Apache Ant developers aware about it [HADV13]. If this convention were better known by developers, it would have been applied in source code at once or in a shorter timeframe.

Second, in the Pharo language[3], there is a convention where calls to Collection.isEmpty().ifTrue(*) should be replaced by calls to Collection.ifEmpty(*) to improve legibility when testing empty objects in conditional statements. However, in Roassal[4], a system implemented in Pharo, the convention is the

---

[2]http://ant.apache.org
[3]http://www.pharo.org
[4]http://objectprofile.com/ObjectProfile.html

opposite: calls to Collection.ifEmpty(*) should be replaced by calls to Collection.isEmpty().ifTrue(*). An expert pointed that Roassal adopts this usage convention because it should be portable over different Smalltalk dialects, and as Collection.ifEmpty(*) is Pharo-specific, it should be avoided; we detected this convention being occasionally applied in Roassal source code during *one year and half*. Notice that, applying the Pharo-specific rule to Roassal, or the Roassal-specific rule to Pharo would actually decrease their code quality.

These two cases have some characteristics in common:

- *They can be described as change rules*: as they are recurrent and follow a specific format, they can be described as change rules similarly to recommendation rules found in current static analysis tools but with the advantage of being system-specific. Once described as rules, they can be used to ensure consistency, which is particularly interesting in the context of large-scale systems.

- *They are spread over time*: the changes occur in different revisions (commits) of the systems, differently, for example, from changes related to API evolution involving class/method renaming, which cannot be spread over time.

Previous researchers took advantage of the fact that similar source code changes are recurrent to support bug discovering (*e.g.*, [LZ05,WH05,KPW06, NNP+10,SSPR12]), or to filter rules provided by static analysis tools (*e.g.*, [KE07]). In the bug-discovering context, researchers restrict their analysis to bug-fix changes whereas in the rule filtering context researchers restrict their analysis to rule-fix changes. In both cases, they do not focus on the detection of change conventions, and the variable *time* is never considered. In fact, the presented examples are neither predefined rules nor related to bug-fix changes: they are system-specific conventions incrementally applied by developers over time.

Summary

Similar source code changes are recurrent over history. Based on the analysis of such data, previous studies propose to support bug discovering or to filter rules provided by static analysis tools. Overall, the analysis of source code changes to produce system-specific conventions are not yet covered by existing studies. In Chapter 4 we propose an approach that focuses on the extraction of system-specific conventions from source code history.

2.3.2 Supporting Client Systems

Many approaches have been developed to support the evolution of framework and libraries and reduce the efforts of client developers. Chow and Notkin [CN96] present an approach where the framework developers annotate changed methods with replacement rules that will be used to update client systems. Henkel and Diwan [HD05] propose CatchUp, a tool that uses an IDE to capture and replay refactorings related to the framework evolution.

Some studies compute rules by comparing two version of one system. Kim *et al.* [KNG07] automatically infer rules from structural changes. The rules are computed from changes at or above the level of method signatures, *i.e.*, the body of the method is not analyzed. Kim and Notkin [KN09] propose a tool (LSDiff) to support computing differences between two system versions. In such study, the authors take into account the body of the method to infer rules, improving their previous work [KNG07] where only method signatures were analyzed. Each version is represented with predicates that capture structural differences. Based on the predicates, the tool infers systematic structural differences. Nguyen *et al.* [NNW$^+$10] propose a tool (LibSync) that use graph-based techniques to help developers migrate from one library version to another. In this process, the tool takes as input the client system, a set of systems already migrated to the new library as well as the old and new version of the library in focus. Using the learned adaptation patterns, the tool recommends locations and update operations for adapting due to API evolution.

Some studies address the problem of discovering the mapping of APIs between different platforms that separately evolved. For example, Zhong *et al.* [ZTX$^+$10] target the mapping between Java and C# APIs while Gokhale *et al.* [GGP13] present the mapping between JavaME and Android APIs.

Dig and Johnson [DJ05] help developers to better understand the requirements for migration tools. For example, they found that 80% of the changes that break client systems are refactorings. Cossette and Walker [CW12] found that, in some cases, API evolution is hard to handle and needs expert assistance.

Another solution to support client systems is based on the identification of evolution rules. Such rules describe method call replacements, in particular, in the format *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many*, meaning that one more method calls should be replaced by one or more method calls. For example, an *one-to-one* rule may define that calls to method m1() should be replaced by calls to method m2(), whereas an *one-to-many* rule may define that calls to method m1() should be replaced by calls to method m2() and m3(). Next, we present existing studies that automatically identify

evolution rules.

Dagenais and Robillard [DR08] propose a tool (SemDiff) that suggests replacements for framework elements based on how a framework adapts to its own changes. The tool first detects code locations that used a method that was later deleted, then, it mines the replaced methods to understand how they were adapted. The tool is able to provide on request rules and examples extracted from the framework's source code to help developers. Schäfer *et al.* [SJM08] propose to mine framework usage change rules from client systems. The approach produces rules by comparing two versions of a class using the framework, and it is also able to produce rules related to fields and inheritance relations. Both studies produce *one-to-one* rules.

Wu *et al.* [WGAK10] propose an approach (AURA) that combines call dependency and text similarity analyses to produce evolution rules. They extract rules by comparing two major versions of the framework in order to produce *one-to-one*, *one-to-many* and *many-to-one* rules. Meng *et al.* [MWZM12] propose a history-based matching approach (HiMa) to support framework evolution. The rules are extracted from the revisions in code history together with comments recorded in the evolution history of the framework. HiMa improves the AURA approach by producing rules at revision level and by handling *many-to-many* rules. Both approaches extract rules from frameworks and libraries rather than from clients.

The approaches AURA and HiMa can produce different types of rules as opposed to only *one-to-one* rules. Such two approaches use deletion and addition of methods as the basis for detecting rules. On the one hand, they are able to produce rules about methods deleted from old release and added to the new one, *i.e.*, rules about method replacement (*e.g.*, due to renaming). On the other hand, they cannot produce rules about methods existing in the new release, *i.e.*, rules about method suggestion (*e.g.*, to improve performance ou legibility).

Motivating Examples

We present concrete and real-world cases in which evolution rules are helpful to developers, and we detail the lacks found in the literature.

1. One-to-one rules about method replacement. This type of evolution rule presents the basic case in which one method in the old release of the framework is replaced by another method in the new one. It is covered by several previous studies [DR08, SJM08, WGAK10, MWZM12]. For example, it would catch the case where in JHotDraw[5] 5.2 to 5.3 the method LineConnection.end() was replaced by LineConnection.getEndConnector() [WGAK10].

---

[5] http://www.jhotdraw.org

Thus, these rules would help a client of this framework searching a replacement for LineConnection.end(). Overall, such approaches produce rules either on the request of the developer or by automatically listing them.

2. One-to-one rules about method suggestion. This type of evolution rule presents the case in which one method should be replaced by another, but both methods exist in the new release. It is covered by some previous studies [DR08, SJM08]. For example, it would catch the Apache Ant case where a convention stating a new way to close files, *i.e.,* calling FileUtils.close(InputStream) instead of InputStream.close(), was introduced to close files [HADV13]. Note that both solutions to close files are available in the new release, *i.e.,* both methods InputStream.close() and FileUtils.close(InputStream) exist. However, the latter should be the one called.

3. One-to-many/many-to-one/many-to-many rules about method replacement. This type of evolution rule presents the case in which one or more methods in the old release is replaced by one or more methods in the new one. Wu *et al.* [WGAK10] cover *one-to-many* and *many-to-one* cases whereas Meng *et al.* [MWZM12] improve by adding *many-to-many*. For example, this type of rule would catch the case where in JHotDraw 5.2 to 5.3 calls to CutCommand(DrawingView) were replaced by calls to CutCommand(Alignment, DrawingEditor) *and* UndoableCommand(Command) [WGAK10].

4. One-to-many/many-to-one/many-to-many rules about method suggestion. This type of evolution rule presents the case in which one or more methods should be replaced by other methods, but they still exist in the new release. This case is not covered by any of the previous approaches because (i) they are restricted to the analysis of *one-to-one* rules [DR08, SJM08] or (ii) they rely on method deletion/addition as the basis for detecting rules [WGAK10, MWZM12]. We present here two examples in which this type of rule would be helpful for developers. First, in the Moose platform[6], calls to MooseModel.root() *and* MooseModel.add(MooseModel) were replaced by MooseModel.install(). In this case, all the methods are available but MooseModel.install() is suggested to be used according to a core developer[7]. Second, in Pharo, calls to UserManager.default() *and* UserManager.currentUser() were replaced by calls to Smalltalk.tools() *and* UserManager.userManager(). Again, all the methods are available, but as the method Smalltalk.tools() acts as a facade to access the system, it is the suggested solution. Notice that, an approach that uses deletion/addition of methods for detecting rules would not detect any of these two examples because such methods were neither removed nor added from the system.

In short, none of the previous approaches cover the types 1, 2 and 3 of

---

[6]http://www.moosetechnology.org
[7]http://goo.gl/b41V9C

evolution rules together, and no existing approaches handle type 4.

While only one approach provides on request rules [DR08], the others provide a list of rules; no existing approaches provide both (on request and a list of rules) to support the developers. In addition, only one approach [DR08] is able to display source code change examples. Table 2.1 compares the main characteristics of the studies in the context of evolution rules.

Table 2.1: Comparison of related approaches to detect API evolution rules; *repl* means rules about method replacement and *sugg* means rules about method suggestion; *ch hist* means that the approach mines all the change history and *two vers* means that the approach compares two versions to produce rules; *on req* means that rules are produced on request and *list* means that a list of rules is produced. (* AURA does not produce *m-to-n* rules).

Approach	Mapping				Input	Recommen-dation	Example helper
	1-to-1		1-to-m, m-to-1 m-to-n				
	repl.	sugg.	repl.	sugg.			
SemDiff [DR08]	yes	yes	no	no	ch. hist.	on req.	yes
Schafer *et al.* [SJM08]	yes	yes	no	no	two vers.	list	no
AURA [WGAK10]	yes	no	yes*	no	two vers.	list	no
Hima [MWZM12]	yes	no	yes	no	ch. hist.	list	no

Summary

Current studies about evolution rules do not properly describe *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many* changes, which are important because they can guide client developers towards new functionalities in the new release of the framework. In Chapter 5 we propose an approach that covers the four types of evolution rules, involving both method replacement and suggestion. Our rules are generated on the request of the developer or by listing, and our tool is able to display an example helper to better support client systems.

2.4   Analysis of Software Ecosystems

Software ecosystem is an overloaded term, which has several meanings. There are two principal facets: the first one focuses on the business aspect [MS05, JBC13], and the second on the artifact analysis aspect, *i.e.*, on the analysis of multiple, evolving software systems [JSW11, Lun09, RLR12]. In this work we use the latter one; we consider an ecosystem to be "*a collection of software projects which are developed and co-evolve in the same*

environment" [Lun09]. The environment can be an organization, or an open-source community, and these software systems have common underlying components, technology, and social norms [JSW11].

Software ecosystems have been studied under a variety of aspects. Jergensen *et al.* [JSW11] study the social aspect of ecosystems by focusing on how developers move between projects in the software ecosystems. The studies of Lungu *et al.* [LRL10] and of Bavota *et al.* [BCP$^+$13] aim to recover dependencies between the software projects of an ecosystem in order to support impact analysis. Lungu *et al.* [LLGR10] focus on software ecosystem analysis through interactive visualization and exploration of systems and their dependencies. Gonzalez-Barahona *et al.* [GBRM$^+$09] study the Debian Linux distribution to measure its size, dependencies, and commonly used programming languages. Manikas *et al.* [MH13] present an overview about current studies on software ecosystems.

Recently, Mens *et al.* proposed the investigation of similarities between software ecosystems and natural ecosystems found in ecology [MCG14, MCGS14]. In this context, they are studying the GNOME and the CRAN ecosystems [CMG14] to better understand how software ecosystems can benefit from natural ones. German *et al.* [GAH13] also analyze the evolution of the CRAN ecosystem, investigating the growth of the ecosystem, and the differences between core and contributed packages.

In the context of ecosystem impact analysis, McDonnell *et al.* [MRK13] investigate API stability and adoption on a small-scale Android ecosystem (10 systems). In such study, API changes are derived from Android documentation pages. The have found that Android APIs are evolving fast while client adoption is not catching up with the pace of API evolution.

In a large-scale study, Robbes *et al.* [RLR12] investigate the impact of a specific type of source code evolution, API deprecation, in a ecosystem that includes more than 3,600 projects. In this case, the API changes are extracted from source code, not from documentation. They have found that API deprecation caused a large impact in the ecosystem, but a strong minority of clients stayed in an inconsistent state for long periods of time or did not react at all.

Summary

Some studies have been done on ecosystem analysis. In addition, many studies have also been developed to support source code evolution, as presented in the previous section, but not in the context of software ecosystems. The impact of source code evolution covering software ecosystems remains unknown because current studies are either restricted to small-scale ecosystems or specific types of API evolution (*i.e.*, API deprecation). In Chapter 6 we provide a large-scale study, at the ecosystem level, to better understand to which

extent client developers are impacted by software evolution.

2.5 Summary

In this chapter we showed that there are lacks in current approaches to support software evolution, in particular to ensure source code consistency. We considered three aspects: (i) analysis of generic and expert-based rules, (ii) extraction of history-based rules, and (iii) analysis of software ecosystems.

We identified the following three problems. First, expert-based rules have not yet been evaluated as to defect prevention. Second, history-based rules do not properly use the source code repositories as to produce better rules. Finally, the real impact of source code evolution in the actual clients is not known. The next four chapters present our approaches to cover each of these problems.

Benefits of Expert-based Rules

Contents

3.1 Introduction

The evolution of frameworks and libraries is likely to impact their clients. Such impact has been shown to be very large and even unknown in the context of software ecosystems [RLR12]. As a solution to alleviate this issue, developers can make use of rules to better manage the impact. Overall, these rules can be generic or system-specific, created by experts. Such rules are normally targeted towards multiple goals, such as reliability or portability [BM08], but very few are focused on the system under analysis.

As these rules are intended to improve code quality, a solution to evaluate them is to verify whether they are likely to point to defects in source code. In this context, there are empirical evidences supporting the intuition that generic rules do not prevent the introduction of defects in software systems [KE07, CMSV12, BM09, BM08, BvdB06, KAYE04]. In general, violations and observed defects are independent, there is no correlation between them. We hypothesized that this happens because the rules used are not focusing on specific problems of the system under analysis. For example, studies indicate that the most prevalent type of defect is semantic or program specific [KPW06, LTW$^+$06, BM09]. These kinds of defects cannot be easily detected by generic rules [KE07]. In addition, using system-specific rules is not easy since they must be defined by an expert of the system under analysis, which is costly. The efficiency of such specific rules for defect prevention or reduction needs therefore to be demonstrated.

In this chapter, we report on a systematic study to investigate the relation between, on one side, generic or system-specific violations and, on the other

side, observed defects [HADA12]. For that, in a first step, we consider all violations reported by generic and system-specific rules. In a second step, we consider those violations that better correlate with the presence of defect.

The study is performed on Seaside, a web application framework, that has been used and maintained for years and for which system-specific rules have been created by experts [RDGN10]. The results show that system-specific rules provide a better defect prevention than generic ones.

The main contributions of this chapter can be summarized as follows:

1. We provide replication of previous experiments as to the lack of correlation between generic violations and defects.

2. We provide new experiments on the correlation between system-specific violations and defects.

3. We provide a comparison between the precision of generic and system-specific rules as to defect prediction.

Structure of the Chapter

Section 3.2 gives an overview of the approaches used to assess and match violations and defects in source code; these approaches are used to support our study. Section 3.3 presents our experiment setting. Section 3.4 details the results of the empirical study. Section 3.5 presents the evaluation of the results. Section 3.6 discusses threats to validity of the experiment, and Section 3.7 concludes the study.

3.2   Assessing and Matching Violations and Defects

This section describes the approach adopted to assess and match violations and defects in source code. More specifically, we describe how to assign violations and defects to software components such as methods or lines of code. Then, we present how to match violations and defects to lines of code.

3.2.1   Violations and Defects

Previous researches try to predict defects at the levels of classes, methods, or lines of code [CMSV12, OWB04, BM08, BM09]. This work is about evaluating the relation between violations and defects, which requires to match them to source code. Working at the level of line is considered more difficult but giving better results [KE07] because it gives a more detailed level of granularity. For example, static analysis tools usually give violations at the level of lines of code, therefore, it is best to identify defects at the same level so that both information match.

In contrast, a problem that may occur when adopting class or method levels is the possibility of wrong matching. Consider for example matching defects and violations at the class level, a violation may occur in one method of the class and a defect in another method. It is not clear in this case whether the match between a violation and a defect at the class level is really meaningful. The same mismatch happens, to a lesser extent, at the method level. For this reason, in our research, we work at the level of lines of code.

To match violations and defects, we must identify lines with violations and/or defects:

- **Identifying lines with violations.** As many rules provided by static analysis tools work at the level of line, a violation points directly to the rule breaking line.

- **Identifying lines with defects**. This task is done by mining commit messages in the software history to find bug-fix changes. Two approaches for this step are normally used: searching for keywords such as "Fixed" or "Bug" [MV00] or searching for references to bug reports [SZZ05]. Identifying a bug-fix commit allows one to identify the code changes that fixed the defect, and, therefore, where the defect was located in the source code. A line of code is related to a defect if it is modified by a bug-fix change, since to resolve a problem the line was changed or removed [KE07]. Such line is marked as defect related because a single bug may be caused by several defects (lines of code).

Therefore, each line of code may be marked as violation and/or defect related. If one considers that violations should prevent defects, one is interested in the lines that have both markers, called true positives (TP), *i.e.*, the cases where rules truly identified defects. Lines marked only with violations are false positives (FP), *i.e.*, rules pointed to lines without defects. Lines marked only with defects are false negatives (FN), *i.e.*, rules failed to identify defects. Finally, lines with neither violations nor defects are true negatives (TN), *i.e.*, rules correctly ignored lines without defects. Such approach is also used by [BM09, BM08, KE07] to assess the true positives.

Of course, rules may be actually created with other purposes than detecting defects, but in this research field we focus on their ability to prevent bugs. We will come back on this issue later in our experiment by considering only rules that better correlate with the presence of defects (*i.e.*, the *top rules*).

3.2.2   Unique Lines of Code

Bugs happen at various moments in software life and are corrected at different time. One needs to consider many versions of a system to collect data

on various bugs and be able to meaningfully correlate violations and defects. During the life of the system, lines are added, removed or changed for many reasons not all related to bug-fixes. Line-based rules will raise (or not) a violation for a line as long as the line remains unchanged. A defect will be marked on a line from the time it is corrected (actually, just before) back to the time it was created. Thus, previous studies work with the idea of *unique lines of code* (ULoC) [BM08, BM09].

A ULoC is a line that remains unchanged, possibly across several versions of the system. It is created at one point in time (a version of the system) and ends when it is changed or deleted. Different ULoCs in the system, even if they are physically contiguous in the code, may have different extension in time depending when they are created or ended. To match violations to defects at the line level, one needs to identify all the ULoCs in the system during the whole time period of the experiment. This is done by creating a graph that represents a method history (a method history contains one or more method versions) in which each node represents a physical line of code and each edge represents a non-changed line between two method versions; a path in the graph is a ULoC (see Figure 3.1). This approach is similar to the result of the SVN *annotate* and Git *blame* commands, where it is possible to know in one method version when the line was last modified. This graph is also close to an *annotation graph* [ZKZW06, KZPW06], but the former just keeps track of non-changed lines while an annotation graph also keeps track of modified lines.

Before creating the graph, the source code of the methods is normalized such that blank lines and formatting changes are ignored. By doing this, we avoid, for example, cases where a violation points to one line in one version and the same violation points to two lines in another version, due to changes of formatting between versions.

ULoCs support the computation of true and false positives and negatives. Figure 3.1 shows an example of the generated graph representing a method history with four versions (four commits). Numbers in the top left corner of the boxes are unique identifiers for the ULoCs. We see that the first physical line of code has never been modified in these four versions, therefore, it makes a single ULoC (ULoC-1). The second physical line of code in version 1 was modified in version 2, therefore, it also makes a single ULoC (ULoC-2), and another one, ULoC-3, for the same physical line of code starting at version 2. In total, there are eleven ULoCs.

In Figure 3.1 there are also two bug-fixes, in version 2 and 4. The lines changed or removed to correct the bugs are marked with the word "defect". For illustration, the actual line of code is shown in the node just before and after the bug-fix. ULoCs with violation and defect (TP) are presented in light
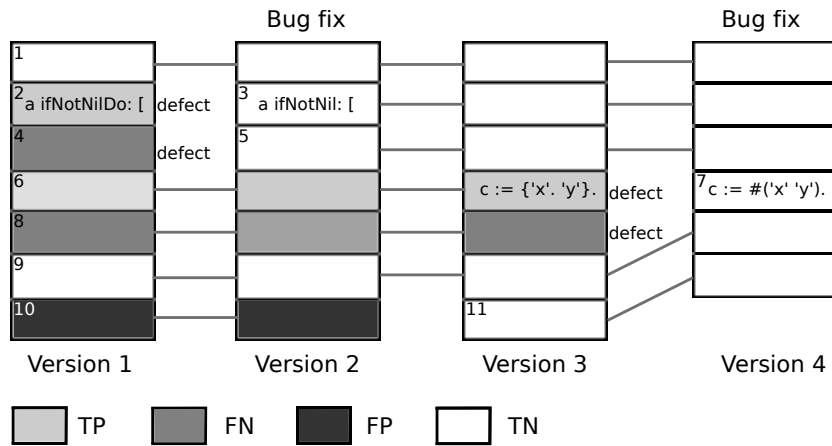
Figure 3.1: Example of graph representing a method history with four versions and with ULoCs marked with violations and defects

gray, violation-only ULoCs (FP) in strong gray, defect-only ULoCs (FN) in medium gray, and ULoCs with neither markers (TN) are left in white. Therefore, in Figure 3.1 there are:

- 2 TPs: ULoCs 2 and 6;

- 1 FP: ULoC 10;

- 2 FNs: ULoCs 4 and 8;

- 6 TNs: ULoCs 1, 3, 5, 7, 9 and 11.

One can measure the efficiency of a rule from the portion of violations predicted correctly over all ULoCs marked with violations, *i.e.*, $TP/(TP + FP)$.

3.3 Experiment Setting

In this section we plan our experiment as suggested in [WRH$^+$00].

3.3.1 Research Questions

We want to assess the correlation between generic violations and defects as well as between system-specific violations and defects. Moreover, we want to study if system-specific rules are better bug predictors than generic rules. As part of this research we need to replicate previous experiments (*e.g.*, [KE07, CMSV12,BM09,BM08,BvdB06]) on our case study, showing that generic coding rules generate many false positives with regard to bug prediction.

We rephrase here our three contributions in the form of three questions. These questions will then be formalized into more specific research questions that will allow us to define formal hypotheses.

1. Can generic violations be used for defect prevention?

2. Can system-specific violations be used for defect prevention?

3. Are system-specific violations more likely to point to defects than generic violations?

We derive two versions of such questions. The first simply considers *all* generic rules and *all* system-specific rules. The second considers a more restricted set of generic and system-specific rules that we call *top* rules.

Evaluating All Rules. The next three questions take into account all rules. A first question is about the relation between generic violations and defects. We want to know if these two variables are related or independent:

RQ1 *Is there a relation between generic violations and defects?*

This question has already been answered, mostly negatively, in other studies. We replicate it here because we work with a language that was not considered previously (Smalltalk) and with a static analysis tool similarly not studied (SmallLint).

We also want to know if there is a relation between system-specific violations and defects and this motivates our second research question:

RQ2 *Is there a relation between system-specific violations and defects?*

The previous questions analyze the relation between violations and defects. In addition, we want to assess whether system-specific violations are better than generic violations with respect to defect prevention:

RQ3 *Are system-specific violations more likely to point to defects than generic violations?*

Evaluating Top Rules. Generic rules are somehow easier to define than the system-specific ones, because one can do it once for distinct systems. In contrast, system-specific rules must be created for each new system. With more generic rules, one should expect more violations and consequently (as is typical in information retrieval) one should also expect better bug coverage (more bugs will be covered) but with lower precision. That is to say, each generic violation will have a lower probability of indicating a bug. Thus, we may expect that generic rules will naturally fare lower (*e.g.*, RQ3) than system-specific ones, simply because they are more numerous and will give more hits. To have a fairer comparison, we will perform the same experiments for the top rules of the two sets, *i.e.*, the rules that better correlate with the presence of bugs.

Given selected groups of top generic and top system-specific rules we can ask the same questions as RQ1, RQ2, and RQ3.

RQ4 *Is there a relation between top generic violations and defects?*

RQ5 *Is there a relation between top system-specific violations and defects?*

RQ6 *Are top system-specific violations more likely to point to defects top generic violations?*

By answering such questions we can define which groups of rules are worthwhile for defect prevention in the case study under analysis.

3.3.2   Case Study

The *context* of the experiment is real systems for which source code, commits logs linked to an issue database, and generic and system-specific rules are available. We need real systems to ensure that our experiment is meaningful. Moreover, we need systems with commits linked to issue database and generic and system-specific rules to assess the relation between defects and violations. System-specific rules are defined by the experts of the system under analysis.

One difficulty of this research is to find systems that fulfill these requirements, and particularly for which a set of system-specific rules is defined. We selected Seaside [DLR07] to perform our empirical studies mainly because of its set of system-specific rules [RDGN10], but also because it has the advantage of being a real-world and non-trivial application, with a consolidated number of users and a relevant history of bugs.

Seaside[1] is an open-source framework for developing web applications written in Smalltalk [DLR07]. It is a competitor for Ruby on Rails as the framework of choice for rapid web prototyping. This system defines various internal domain-specific languages to configure application settings, nest components, define the flow of pages, and generate XHTML. We analyze the impact of system-specific rules for defect prevention when compared with generic rules on a long term evolution of Seaside. We analyze 943 snapshots of Seaside core, which were produced in almost four years of development (from November 2007 to September 2011). Table 3.1 presents an overview of the size of our case study.

Seaside includes **Slime** [RDGN10], a Seaside-specific program checker consisting of a set of rules, created by experts, working at the level of the abstract syntax tree (AST); we call these *system-specific rules*. Smalltalk includes **SmallLint** [RBJ97], a generic program checker consisting of rules also working at AST level; we call these *generic rules*. SmallLint can be compared to other static analysis tools such as FindBugs [HP04] and PMD [Cop05] and it comes with rules targeting common bugs and code smells in Smalltalk.

---

[1] http://www.seaside.st

Table 3.1: Overview of Seaside core size

Number of snapshots	943
Average classes per snapshot	216
Average methods per snapshot	1,592
Average LOC per snapshot	6,428

3.3.3 Experiment for RQ1, RQ2, RQ4 and RQ5

This first experiment is about the relation between violations and defects. We want to know if these two variables are related or independent.

Hypotheses Formulation. The null and alternative hypotheses are formalized as:

$H_0^{1,2,4,5}$ *Violations and defects are independent.*

$H_a^{1,2,4,5}$ *Violations and defects are related.*

Variable and Subject Selection. The *independent variable* is the violations raised by the rules on lines of code. It is categorical and can take two values: *with violation* and *without violation*. The *dependent variable* in this study is the defects, which are raised by bug-fix changes on lines of code. It is also categorical and can take two values: *with defect* and *without defect*.

The subjects for these experiments will be ULoCs from our case study (Seaside) and we will measure the number of ULoCs in each of four categories: with violation/with defect (TP), with violation/without defect (FP), without violation/with defect (FN), and without violation/without defect (TN).

Experiment Design. To test the hypotheses $H^{1,2,4,5}$ we use the Chi-squared test, which can be used when there are two categorical variables, each with two or more possible values. The null hypothesis is that the frequencies for the dependent variable (defects) are the same for different values of the independent variable (violations). If we cannot reject the null hypothesis, we must conclude that the variables are in fact independent. When we can reject the null hypothesis (*i.e.*, the variable are dependent), it is also important to understand how the variables are related. This is done by observing the Pearson residuals, which measure the difference between the observed and expected frequencies. When the absolute value of the residual is greater than two (> 2), one considers that the observed frequency is significantly higher than the expected and that more of the independent variable should induce more of the dependent one.

As is customary, the tests will be performed at the 5% significance level which means there will be a probability of 5% or less of erroneously rejecting the null hypothesis. We will also report the *effect size* which measures the distance between the null hypothesis and alternative hypothesis, and is independent of sample size. Effect size value 0.1, 0.3 and 0.5 are considered small, medium and large effects, respectively.

One of the traditional requirements for Chi-square is independence of observations, *i.e.*, presence or absence of a violation or a defect in one ULoC should not affect presence or absence of a violation or a defect in another ULoC. In this context, let us consider the following example: a ULoC with a violation is modified to address such violation, creating a new ULoC that is supposed to be violation-free. However, the new ULoC is not guaranteed to be violation-free; the developer may fix a violation by adding another one.

3.3.4 Experiment for RQ3 and RQ6

Hypotheses Formulation. The null and alternative hypotheses are formalized as:

$H_0^{3,6}$ *System-specific and generic violations are equally precise in identifying defects.*

$H_a^{3,6}$ *System-specific violations are more precise in identifying defects than generic violations.*

Notice that we make a directional (one-tailed) hypothesis. This should be made when there is evidence to support such a direction. This evidence will stem from the results of the first experiments where we will answer negatively to RQ1 (there is no relation between generic violations and defects) and positively to RQ2 (there is a relation between system-specific violations and defects).

Variable and Subject Selection. The *independent variable* in this study is the group of rules (generic or system-specific). The *dependent variable* could have been the precision of the coding standard rules where precision is the percentage of ULoC with violation and defect among all ULoC with violation. However, we will show in the result section that we do not have enough rules with non-null precision to perform a test: many rules didn't give any violation, so precision is undefined for them, and other got violations but not on ULoCs with defects, so precision would be null for them.

To bypass this issue, we will group ULoCs according to another criterion. We will consider all ULoCs in the history of a method as one subject. Another way to see it is to say that the combined versions of one method will be a subject. We considered working with "normal" methods as subject, that is to say one method in one version, but this would have the drawback that

the same ULoC can appear in several versions of a method (if it does not change between these versions, *e.g.* see ULoC-1 in Figure 3.1), and therefore would have been counted more than once. By taking the whole history of each method, we avoid giving more weight to some ULoCs.

The metric used will be the Positive Predictive Value (PPV) which has the same formula as precision[2]: proportion of ULoC with defects among those with violation. A high PPV indicates that the method, in its history, tends to have defects where it breaks some generic (or system-specific) coding standard rule.

Experiment Design. For this experiment, we use an unpaired setting, which means the methods composing one sample may not be the same as those composing the other sample. This is due to the fact that not all methods break generic and system-specific coding standard rules, many of them break either one or the other category of rule, and even fewer methods would have a non null PPV for both categories.

We test the hypotheses H^3 and H^6 with a Mann-Whitney test which is used for assessing whether one of two samples of independent observations tends to have larger values than the other. It can be used when the distribution of the data is not normal and there is different participants (not matched) in each condition. The null hypothesis is that the median PPV is the same for both samples.

Again the tests will be performed at the 5% significance level and we will also report the *effect size*.

3.3.5 Instrumentation

Defects. For our research, we use the prediction at the level of lines of code because it is a more precise level of granularity and it also avoids the issues stated in Section 3.2.1. To identify bug-fix changes, we use the technique of searching for keywords since Seaside has a normal practice of writing bug-fix commits with the keyword "Issue". Seaside history contains $14,416$ ULoCs, from which 664 (4.6%) contained defects.

All Rules. With respect to the used rules, we considered two sets of rules, the first one with 91 generic rules and the second one with 29 system-specific rules. We considered only rules that work at the level of lines of code, excluding, for example, rules like "method too long". Below, we briefly describe the groups of generic rules. The number of individual rules by group is shown between parentheses.

---

[2]Recall is not calculated because it would imply discovering all the bugs in the history of system under analysis, even the ones not marked in the commits, which is an impracticable task.

- *Unnecessary code (20).* It targets code that is not needed or can be avoided (replaced) since other pieces of code can be more efficient or legible, *e.g.*, an assignment that has no effect.

- *Spelling (5).* It looks at identifiers to find words wrongly spelled.

- *Possible bugs (20).* It targets general code that is considered likely to cause bugs, *e.g.*, an unconditional recursion, or modification of a collection while iterating over it.

- *Pharo bugs (7).* It searches for code patterns specific to Pharo (Smalltalk dialect used in this work) that could cause bugs, *e.g.*, debugging code left in a method.

- *Bugs (7).* Another kind of code that can cause bugs, *e.g.*, a missing super implementation, a method that overrides a "system" message.

- *Miscellaneous (13).* It searches for different patterns that, for example, a programmer coming from other languages might produce, *e.g.*, in arithmetic expressions[3].

- *Intention revealing (19).* It searches for code related to the intention revealing pattern, *e.g.*, a code that breaks the Law of Demeter, variable capitalization, or code using the wrong iterator.

The system-specific rules are separated into the following groups:

- *Portability (8).* Seaside runs without modification on 7 different platforms which differ slightly in both the syntax and the libraries they support [RDGN10]. Thus, this category targets code patterns specific to some platforms, *e.g.*, code that uses dynamic arrays, or some specific methods/classes not portable across different Smalltalk platforms.

- *ANSI compatibility (8).* It targets code that is not ANSI compatible and is also related to the portability of Seaside.

- *Possible bugs (12).* It targets Seaside-specific code that is likely to cause bugs, *e.g.*, code in which a given message is not the last in a specific sequence of method calls.

- *Formatting (1).* It targets code in which a specific pattern must be followed, *e.g.*, a correct pattern to deprecate an API protocol.

Some of these rules would clearly not be related to bug prevention (for example the spelling group), thus, we also experimented with the *top rules* as already introduced in Section 3.2 and detailed in the next subsection.

Table 3.2 presents the number of violations raised by all rules as well as the percentage of ULoC impacted.

---

[3]In Smalltalk arithmetic operators are normal methods, so "arithmetic expressions" are evaluated from left to right without operator precedence, $1 + 2 * 3$ is interpreted as (Java like notation) 1.add(2).times(3) = 3.times(3) = 9

Table 3.2: Seaside violations for all rules.

Group	#violations	ULoC with violations
All generic	1,118	7.7%
All system-specific	312	2.1%

Top Rules. In this subsection we detail the approach used to determine the top rules. Table 3.3 shows all the rules that generate some violations and for which at least one violation coincide with a defect, that is to say the rule for which $TP > 0$. This result confirms previous work, in which a subset of rules performs better than others [KE07, BM09, BM08, BvdB06]. Rules prefixed by "GR" are system-specific rules and those prefixed by "RB" are generic rules.

Table 3.3: Rules with $TP > 0$. Rules in **bold** performed significantly better than a random predictor (top rules).

Rule	#violation	#TP
GRAnsiCollectionsRule	8	1
GRAnsiConditionalsRule	**118**	**18**
GRAnsiStreamsRule	11	1
GRAnsiStringsRule	**40**	**10**
GRDeprecatedApiProtocolRule	56	3
GRNotPortableCollectionsRule	**7**	**4**
RBBadMessageRule	16	1
RBGuardingClauseRule	19	2
RBIfTrueBlocksRule	**7**	**2**
RBIfTrueReturnsRule	**14**	**3**
RBLawOfDemeterRule	**224**	**18**
RBLiteralValuesSpellingRule	232	10
RBMethodCommentsSpellingRule	216	8
RBNotEliminationRule	58	1
RBReturnsIfTrueRule	72	3
RBTempsReadBeforeWrittenRule	**16**	**3**
RBToDoRule	**38**	**6**

In fact, any random predictor, marking random lines with violations, would, with a sufficient number of attempts, end up with a number of true positives higher than zero, but would not be very useful. Therefore, we can assess the significance of a rule by comparing it to a random predic-

tor [BM09]. As suggested by [BM09, BM08], this problem can be modeled as follows: the project is viewed as a large repository of lines, with a certain probability (p= #ULoC with defects / #ULoC) of those lines being defect related. A rule marks n lines with violations. A certain number of these violations (r) are successful defect predictions (*i.e.*, TPs).

This is compared with a random predictor, which selects n lines randomly from the repository. We can model the random predictor as a Bernoulli process (with probability p and n trials). The number of correctly predicted lines r has a binomial distribution; using the cumulative distribution function $P(TP \leq X \leq n)$ we compute the significance of the rule [BM08]. In conformance with our other statistical tests, we choose a 5% threshold[4]. When the random predictor has less than 5% probability to give a better result than the rule, we call this one a *top rule*. For example, for Seaside we have $14,416$ ULoCs and 664 with defects, so the probability of randomly picking a line with defect is $p = (664/14416) = 0.046$. For rule GRNotPortableCollectionsRule (Table 3.3), TP = 4, $n = 7$ and the cumulative distribution function $P(4 \leq X \leq 7)$ is 0.0001, therefore, we consider it a top rule. The top rules are presented in **bold** in Table 3.3.

Table 3.4 presents the number of violations raised by the top rules as well as the percentage of ULoC impacted.

Table 3.4: Seaside violations for top rules

Group	#violations	ULoC with violations
Top generic	299	2.0%
Top system-specific	165	1.1%

3.4 Experiment Results

In this section we present the results of our empirical study. We first present the results for all rules, then, we follow with the results for top rules.

3.4.1 Evaluating All Rules

The hypotheses for the Chi-squared test are derived from the one presented in Section 3.3.3.

RQ1 *Is there a relation between generic violations and defects?*

H$_0^1$ *Generic violations and defects are independent.*

---

[4]Note however that this is not a statistical test of significance.

$\mathbf{H}_a^1$ *Generic violations and defects are related.*

Table 3.5 shows the contingency table for generic violations. The Chi-squared test gives a *p-value* = 0.65 (> 0.05 significance level), therefore, we cannot reject the null hypothesis that generic violations and defects are independent, *i.e.*, the proportions of lines with and without generic violations are the same in lines with and without defects.

Table 3.5: Contingency table for generic violations (#ULoCs)

	with defect	without defect	total
with violation	55	1,063	1,118
without violation	609	12,689	13,298
total	664	13,752	14,416

Table 3.6 shows the residuals for generic violations, the values close to 0 indicate that there is no significant difference between the observed frequencies and the expected ones.

Table 3.6: Residuals for generic violations

	with defect	without defect
with violation	0.48	-0.10
without violation	-0.14	0.03

Next, we present the results for all system-specific rules.

RQ2 *Is there a relation between system-specific violations and defects?*

$\mathbf{H}_0^2$ *System-specific violations and defects are independent.*

$\mathbf{H}_a^2$ *System-specific violations and defects are related.*

Table 3.7 shows the contingency table for system-specific violations. The Chi-squared test gives *p-value* < 0.001. The effect size is 0.051. We can reject the null hypothesis with a very small probability of error, we conclude that system-specific violations and defects are related, *i.e.*, the proportions of lines with and without system-specific violations are not the same in lines with and without defects.

Table 3.8 shows the residuals for system-specific violations. One can see that condition *with violation and defect* is over-represented (> 2) and is the

Table 3.7: Contingency table for system-specific violations (#ULoCs)

	with defect	without defect	total
with violation	37	275	312
without violation	627	13,477	14,104
total	664	13,752	14,416

major contributor to the rejection of the null hypothesis. We further conclude that defects appear more frequently on lines with system-specific violations than on lines without violation.

Table 3.8: Residuals for system-specific violations

	with defect	without defect
with violation	5.97	-1.31
without violation	-0.88	0.19

We now test whether system-specific violations are better than generic violations with respect to defect prevention. An evidence to support such a direction is the fact that we answered negatively to RQ1 and positively to RQ2.

RQ3 *Are system-specific violations more likely to point to defects than generic violations?*

$\mathbf{H}_0^3$ *System-specific and generic violations have the same PPV.*

$\mathbf{H}_a^3$ *System-specific violations PPV is higher.*

From all method histories, 509 methods had at least one generic violation and 175 had at least one system-specific violation, these will be our two samples in this test. The other methods having no violation have undefined PPV (*i.e.*, $TP/(TP + FN) = 0 \ / \ 0$).

Applying the Mann-Whitney test for such samples, we have *p-value* $= 0.003$. The effect size is 0.1. We can reject the null hypothesis and say that system-specific PPV is higher than generic PPV, methods with system-specific violations have more chance to have defects on these lines than those with generic violation. We conclude that it is better to use system-specific violations to point to defects than generic violations.

3.4.2 Evaluating Top Rules

We also performed the same experiments on the top rules.

RQ4 *Is there a relation between top generic violations and defects?*

$\mathbf{H}_0^4$ *Top generic violations and defects are independent.*

$\mathbf{H}_a^4$ *Top generic violations and defects are related.*

RQ5 *Is there a relation between top system-specific violations and defects?*

$\mathbf{H}_0^5$ *Top system-specific violations and defects are independent.*

$\mathbf{H}_a^5$ *Top system-specific violations and defects are related.*

Table 3.9 and 3.10 show the contingency tables for top generic and system-specific violations, respectively. The Chi-squared tests give *p-value* < 0.001 for both. We can reject null hypotheses $\mathbf{H}_0^4$ and $\mathbf{H}_0^5$. We conclude that top generic violations and top system-specific violations are related to defects.

Table 3.9: Contingency table for top generic violations (#ULoCs)

	with defect	without defect	total
with violation	32	267	299
without violation	632	13,485	14,117
total	664	13,752	14,416

Table 3.10: Contingency table for top system-specific violations (#ULoCs)

	with defect	without defect	total
with violation	32	133	165
without violation	632	13,619	14,251
total	664	13,752	14,416

The effect size is 0.042 for top generic and 0.076 for top system-specific violations. Table 3.11 and 3.12 show the residuals for top generic and top system-specific violations. We see that category *with violation and defect* is over-represented (> 2) and is the major contributor to the rejection of the null hypotheses. We further conclude that defects appear more frequently on lines with top generic or top system-specific violations than on lines without such violations.

Finally, we test whether top system-specific violations are better than top generic violations with respect to defect prevention. An evidence to support such a direction is the fact that residual of the category *with violation and defect* is higher for the former than for the later.

Table 3.11: Residuals for top generic violations

	with defect	without defect
with violation	4.91	-1.07
without violation	-0.71	0.15

Table 3.12: Residuals for top system-specific violations

	with defect	without defect
with violation	8.85	-1.94
without violation	-0.95	0.20

RQ6 *Are top system-specific violations more likely to point to defects top generic violations?*

$\mathbf{H}_0^6$ *Top system-specific and top generic violations have the same PPV.*

$\mathbf{H}_a^6$ *Top system-specific violations PPV is higher.*

From all method histories, 77 methods had at least one top generic violation and 67 had at least one top system-specific violation. These will be our two samples in this test.

Applying the Mann-Whitney test we have *p-value* $= 0.047$. The effect size is 0.14. There is a significant difference between both samples and we can reject the null hypothesis. We conclude that it is better to use top system-specific violations to point to defects than top generic violations.

3.5   Discussion

3.5.1   Evaluating All Rules

We studied the relation between violations and defects. The outcome of our experiments is that generic violations are not efficient to identify lines with defects (RQ1). This is coherent with the conclusions of previously published results [KE07, BM09, BM08, BvdB06]. The fact that different bug-finding tools and programming languages were considered in other experiments and ours, reinforce the general validity of this conclusion.

This results is due to the great amount of false positives generated by generic rules. It hints at the importance of tailoring coding standard rules to a specific domain, which is confirmed by RQ2, showing that system-specific

violations and defects are dependent for the case study under analysis. In this case, we see a reduction of the amount of false positives.

Since RQ1 was rejected, and RQ2 accepted, the result provided by RQ3 is expected: PPV measured for system-specific rules is higher than for generic rules. We conclude that, for the case study under analysis, generic rules are not effective enough to be used for defect prevention. System-specific rules give more relevant information on how to avoid bugs and therefore they are effective to be used for defect prevention.

3.5.2 Evaluating Top Rules

We also studied the relation between top violations and defects. With this experiment, we are fairer to both groups of rules, since we select just the most effective rules for defect prevention, thus producing less false positives. The results of RQ4 and RQ5 show that both top generic and top system-specific violations are related to defects, and thus can be used for defect prevention. This result is also confirmed by previous work, in which a subset of rules performed better than others [KE07, BM09, BM08, BvdB06].

Contrary to RQ3, the result of RQ6 was not clear beforehand because both RQ4 and RQ5 were accepted. Testing RQ6 shows that top system-specific PPV is (statistically) significantly higher than top generic PPV. Therefore, we can say that it is better to use top system-specific violations to point to defects than top generic violations. We conclude that, for the case study under analysis, top system-specific rules are more effective to be used for defect prevention than top generic rules.

3.6 Threats to Validity

3.6.1 Construct Validity

The construct validity is related to whether the measurement in the study reflects real-world situations. In our study, the main threat is the quality of the analyzed rules.

We believe that this threat is alleviated because we adopted a set of rules coming from the most used static analysis tool in Smalltalk (SmallLint) and a set of rules created by the Seaside experts, as the generic and system-specific rules, respectively.

3.6.2 Internal Validity

The internal validity is related to uncontrolled aspects that may affect the experimental results.

The matching between violations and defects may be an underestimation: some bug-fixes only introduce new code, such as the addition of a previously forgotten check clauses.

Overestimation is less likely: although not all lines that are part of a bug-fix may be directly related to the bug, violations on such lines still point out the area in which the bug occurred. These possible problems are also pointed out by [BM08, BM09] since they also study defect prevention at the level of line of code.

Violations might point to defects that have not been found. The influence of such dormant defects is minimized in the case of a long-running project as the one analyzed, where most of the defects will have been found.

Finally, we have not tried to identify instances of method renaming to receive the propagation of defects and violations. If a method foo() had previously been named bar(), bar() will not receive the propagation of defects and violations from foo(). However, only 475 methods have been renamed during the experiment time frame, which includes 943 versions and $1,592$ methods per version on average, *i.e.*, 0.5 method renaming per version. Therefore, there is a very small amount of method renaming, which is hardly likely to impact on the validity of the results.

3.6.3 External Validity

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

We believe Seaside is a credible case study. It includes a large number of versions (943 collected over a time frame of almost four years), classes and methods representing real-world and non-trivial application, with a consolidated number of users and a relevant history of bugs. Despite this observation, our findings — as usual in empirical software engineering – cannot be directly generalized to other systems, specifically to systems implemented in other languages or to systems from different domains, even if a comparison with previous studies [KE07, CMSV12, BM09, BM08, BvdB06] (which analyze systems implemented in other languages and from different domains) yielded similar results.

There are two requirements for the used approach that should be considered when replicating this study. The first is the existence of system-specific rules, the second is the possibility to link software repository and issue database. Many studies have successfully extracted such links before [KE07, SZZ05, KZPW06, BM09, BM08], suggesting that there is a general habit of clearly identifying bug-fixes in many projects.

3.7   Summary

Software evolution is likely impact other systems, which may be aggravated in the context of ecosystems. To alleviate such impact, developers can make use of generic or system-specific rules. As these rules are intended to improve code quality, a solution to evaluate them is to verify their relationship with defects.

To the best of our knowledge, this work is the first to study the use of system-specific rules created by experts for defect prevention. It provides a systematic study to investigate the relation between generic or system-specific violations and observed defects. The study was performed on Seaside, a real world-system, that has been used and maintained for years and for which system-specific rules were created. Two groups of research questions were created to assess whether system-specific rules would be better bug predictors than generic rules. The first questions were about generic and system-specific rules, the second considered the top rules, a more focused set of generic and system-specific rules. All the results reported in this work were statistically significant, and not due to chance.

We conclude that, for the case study under analysis, generic rules were not effective enough to be used for defect prevention. This was also reported by previous work. In contrast, system-specific rules provide more relevant information on how to avoid bugs, and, therefore, they are more effective to be used for defect prevention. With the results reported in this work, we expect system-specific rules to be created and used by developers in complement to generic ones for defect prevention.

The results presented in this chapter show that system-specific rules created by experts are somehow better than generic ones. However, assessing experts is costly, they may not have the complete knowledge of the whole system or they may be no longer available. In that respect, the reduction of the dependence on experts would be relevant. The next two chapters present solutions to extract rules from source code history, reducing the involvement of the experts.

Supporting System-specific Conventions with History-based Rules

In the previous chapter we have seen that expert-based rules can be in fact worthwhile if system-specific. Yet, they are costly since they need to be manually created by experts on the system under analysis. In this chapter we propose to use the source code history as the basis to create rules related to system-specific conventions.

4.1   Introduction

As presented in Chapter 2 in Section 2.3.1, previous researchers took advantage of the fact that similar source code changes are recurrent to support bug-discovering (*e.g.*, [LZ05, WH05, KPW06, NNP+10, SSPR12]), or to filter rules provided by static analysis tools [KE07]. In both cases, related work does not focus on the detection of change conventions, and the variable *time* is never considered to produce rules.

In this chapter, we propose an approach that describes source code change conventions as rules [HADV13, HAE+15b]. These rules are created by mining incremental revisions in code repositories, based on predefined rule patterns that ensure their quality. In this process, we consider methods that were not removed from the system, and we take into account the variable *time*. This allows us to produce rules related to code conventions, which

can even help experts, since they may not have the complete knowledge of the whole system under analysis.

We focus on a category of rules that provides recommendations for developers, which is particularly important to keep consistency in large-scale systems and ecosystems. Such category can be found in static analysis tools, but, of course, in a generic form. For example, PMD contains a set of rules[1] to support porting systems from a Java version to another. FindBugs contains rules[2] that suggest the replacement of method calls to improve performance. In SmallLint, rules provide recommendations to improve code legibility, among others.

We validate our approach on open-source systems with the help of experts. For such experiment, we selected two real systems since we have access to their experts, which is fundamental to receive real assessment about the rules and their violations. The results show that many rules are in fact relevant for the experts and can be used to ensure that better rules are adopted by developers.

Moreover, we discuss the creation of rules according to their occurrence over time to check when a change is frequent enough to be considered as a rule. In addition, we compare the extracted rules with generic rules provided by a static analysis tool. The results show that the two sets are mutually exclusive.

The main contributions of this chapter can be summarized as follows:

1. We provide a novel approach to produce system-specific conventions from source code history.

2. We provide a qualitative and quantitative (both with the help of a system expert) evaluation of the rules extracted from real-world systems.

3. We provide an analysis about the overlap between our rules and rules provided by static analysis tools as well as about the ideal number of changes to create rules.

Structure of the Chapter

We present our approach in Section 4.2. We propose research questions to validate our approach and define our experiment setting in Section 4.3. We present the results of the experiments and discuss them in Section 4.4. We

---

[1]http://pmd.sourceforge.net/pmd-5.0.2/rules/java/migrating.html

[2]Rules in http://findbugs.sourceforge.net/bugDescriptions.html, *e.g.,* (1) "Method allocates a boxed primitive just to call toString", (2) "Method invokes inefficient Number constructor; use static valueOf instead", (3) "Method invokes inefficient Boolean constructor; use valueOf instead", (4) "Use the nextInt method of Random rather than nextDouble to generate a random integer".

present a complementary discussion in Section 4.5. In Section 4.6, we discuss the threats to the validity of our experiments. Finally, we conclude the chapter in Section 4.7.

4.2   Mining System-Specific Rules

In this section we present our approach, which extracts system-specific rules by monitoring API changes found in source code history of the system. Before detailing our approach, we present an overview about it.

4.2.1   Overview of the Approach

Consider the examples shown in Figure 4.1 that occurred in the Pharo[3] programming language. Figure 4.1-I shows a replacement of the static method call RPackageOrganizer.default() by RPackage.organizer(). Figure 4.1-II shows a replacement to improve legibility, *i.e.,* calls to Collection.at(3) are replaced by calls to Collection.third().

I. Diff between revisions 1 and 2 of method foo()
− rpackage = RPackageOrganizer.default();
+ rpackage = RPackage.organizer();

II. Diff between revisions 3 and 4 of method bar()
− if (collection.at(3) == myValue) { . . .
+ if (collection.third() == myValue) { . . .

Figure 4.1: Examples of changes in Pharo. "−" indicates the deleted line and "+" indicates the added line (code converted to Java-like syntax to ease understanding).

Notice that the replaced methods are not removed from the system. In Figure 4.1-I, the old method call, RPackageOrganizer.default(), should only be used in specific cases and by some classes. In Figure 4.1-II, the new method call, Collection.third(), improves code legibility when compared to the old one, Collection.at(3). These changes occurred several times in *different revisions*, and they are evidence of the effort to use of a better API. It is important to ensure that these changes are consistently applied over source code to avoid maintenance problems. Our approach is intended to ensure that such changes can be described as rules. That is to say, instead of using rules only provided by experts that come in static analysis tools, we want to learn from the various changes that occurred on the system over time and provide corre-

---

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

sponding rules. This will minimize the period in which these changes occur over time.

Figure 4.2 shows an overview of our approach. In the first step, we extract deltas from revisions in a system history (Subsection 4.2.2), which is done once for a system history under analysis (or can be done incrementaly after each commit). In the second step, we discover rules based on the extracted changes and the patterns. This step is performed for each pattern, and it is divided in two substeps: (i) it creates rules based on the provided pattern (Subsection 4.2.3) and (ii) it filters the created rules based on their occurrence over different revisions (Subsection 4.2.3).
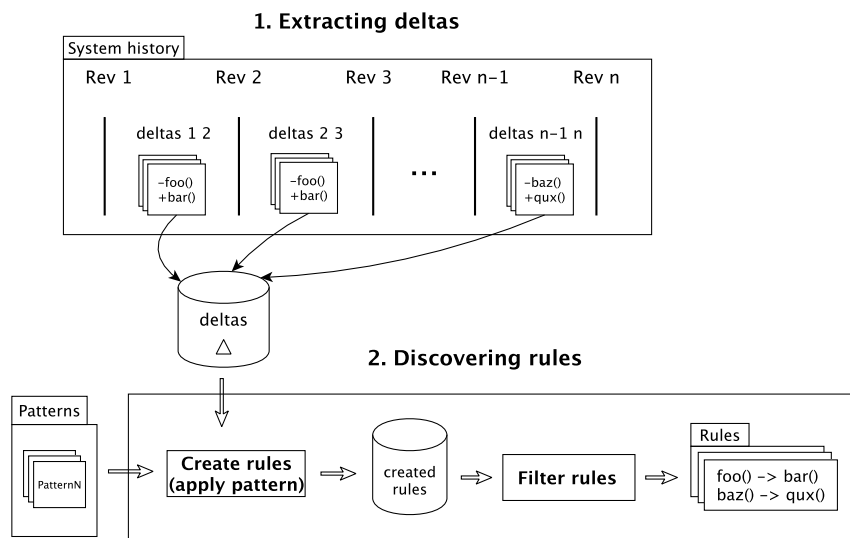
Figure 4.2: Overview of our approach.

4.2.2   Extracting Deltas from Revisions

The first step of our approach is to extract changes from the revisions. When comparing the differences between changed source code, one needs to define what should be analyzed in the code. For example, one can keep track of syntactical changes (*i.e.*, adding or removing conditional statements, modifying expressions), or structural changes (*i.e.*, adding or removing classes, methods, method invocations). While syntactical changes play important role in the context of bugs discovering [NNP+10,KPW06,SSPR12,WH05], they have a small role in discovering systematic changes, for which structural changes are better suited since they do not take into account low level changes [KN09,

LZ05].

In this work, we illustrate our approach by extracting rules from method call changes. In order to do that, we need first to extract the correct data from the source code history of a system.

Let a *delta* be a set of deleted and added invocations between two revisions of a method. We represent a delta with predicates that describe deleted or added method invocations:

deleted-invoc(context-id, receiver, signature, static)

added-invoc(context-id, receiver, signature, static)

where, the predicate *deleted-invoc(. . .)* represents a deleted invocation; the predicate *added-invoc(. . .)* represents an added invocation; *context-id* uniquely identifies a change context (*i.e.*, the changed method and the revision); *receiver* is the receiver of the invocation; *signature* is the signature of the invoked method (for the arguments, the value is presented if they are primitive types such as int, boolean or null, otherwise the type is presented; this is done to obtain more precise convention rules); and *static* is a keyword *isStatic* or *notStatic* that states if the invocation is static.

Small deltas between revisions are preferable to avoid the noise that can be found in large ones [LZ05, MWZ11]. Thus, to avoid the problem of large diff size between methods, making it difficult to extract relevant information, we always select deltas involved in less than five deleted or added invocations. Moreover, discover other type of rules may imply extracting other data (*e.g.*, inheritance), which is not in the scope of this work.

In Figure 4.3, we present the deltas generated by the changes in Figure 4.1.

I. Deltas between revisions 1 and 2 of method foo()
deleted-invoc("foo()-rev2", "RPackageOrganizer", "default()", "isStatic")
added-invoc("foo()-rev2", "RPackage", "organizer()", "isStatic")

II. Deltas between revisions 3 and 4 of method bar()
deleted-invoc("bar()-rev4", "Collection", "at(3)", "notStatic")
added-invoc("bar()-rev4", "Collection", "third()", "notStatic")

Figure 4.3: Deltas generated for the changes in Figure 4.1.

4.2.3 Discovering Rules

Creating Rules

This step creates rules from the extracted deltas. To improve the quality and relevance of the rules, we define patterns that the rules must follow. These patterns will limit the search space, and, thus, the extraction of noisy rules.

In particular, such patterns were inspired by existing change rules found in static analysis tools.

We illustrate our solution with rules related to method call replacement. As presented in the introduction, FindBugs, PMD and SmallLint already include rules to call some methods rather than others, and our goal is to expand this set of rules. In SmallLint, for example, a change rule states that invocations to Object.equals(nil) should be replaced by invocations to Object.isNil() to improve code legibility. Assume that such rule was applied in source code. So, for each replacement, our approach generates a delta similar to, each with their respective *context-id*:

deleted-invoc("mtd()-revX", "Object", "equals(nil)", "notStatic")

added-invoc("mtd()-revX', "Object", "isNil()", "notStatic")

The deltas are used as a database in which we want to find instances of predefined patterns. For example, the previous change follows the pattern where the receiver remains the same (*i.e.,*Object) while the method call changes (*i.e.*, from equals(nil) to isNil()). Thus, we query the following pattern in order to find such changes in our database:

pattern(deletedSignature, addedSignature) =

    deleted-invoc(id, receiver, deletedSignature, "notStatic") and

    added-invoc(id, receiver, addedSignature, "notStatic")

The variables *id*, *receiver*, *deletedSignature* and *addedSignature* are used to ensure the constraints of such pattern. We use the same variable *receiver* in both deleted and added predicates to ensure the same receiver, and we use different variable (*deletedSignature* and *addedSignature*, such that *deletedSignature $\neq$ addedSignature*) to ensure different signatures in both predicates. This pattern will output rules in the format: receiver.deletedSignature $\rightarrow$ receiver.addedSignature, where the left hand side (LHS) presents what should be deleted and the right hand side (RHS) presents what should be added. For example, the previous delta is represented as Object.equals(nil) $\rightarrow$ Object.isNil().

By studying change rules of static analysis tools, we want to abstract patterns that will allow us to automatically extract system-specific rules. Based on that, we define five patterns to abstract change rules. The following list of patterns is not exhaustive, and new ones can be included. Next, we present each pattern.

Pattern 1: Change receiver and invocation, static

pattern1(deletedReceiver, deletedSignature, addedReceiver, addedSignature) =
    deleted-invoc(id, deletedReceiver, deletedSignature, "isStatic") *and*
    added-invoc(id, addedReceiver, addedSignature, "isStatic")

Example

FileDirectory.default() → FileSystem.workingDirectory()

Pattern 2: Change invocation, same receiver, static

pattern2(receiver, deletedSignature, addedSignature) =
    deleted-invoc(id, receiver, deletedSignature, "isStatic") *and*
    added-invoc(id, receiver, addedSignature, "isStatic")

Example

SystemNavigation.default() → SystemNavigation.new()

Pattern 3: Change receiver, same invocation, static

pattern3(deletedReceiver, addedReceiver, signature) =
    deleted-invoc(id, deletedReceiver, signature, "isStatic") *and*
    added-invoc(id, addedReceiver, signature, "isStatic")

Example

SystemChangeNotifier.uniqueInstance() → SystemAnnouncer.uniqueInstance()

Pattern 4: Change invocation, same receiver, non-static

pattern4(deletedSignature, addedSignature) =
    deleted-invoc(id, receiver, deletedSignature, "notStatic") *and*
    added-invoc(id, receiver, addedSignature, "notStatic")

Example

Object.noMoreNotificationsFor() → Object.unsubscribe()

Pattern 5: Change double invocation, same receiver, non-static

pattern5(deletedSignature1,deletedSignature2,addedSignature1,addedSignature2)=
    deleted-invoc(id, receiver, deletedSignature1, "notStatic") *and*
    deleted-invoc(id, deletedSignature1, deletedSignature2, "notStatic") *and*
    added-invoc(id, receiver, addedSignature1, "notStatic") *and*
    added-invoc(id, addedSignature1, addedSignature2, "notStatic")

Example

Object.vm().getSystemAttribute(1001) → Object.platform().name()

Patterns 1, 2 and 3 represent the case where a static invocation is replaced by another. Patterns 4 and 5 cover the replacement of non-static invocations. These patterns were kept because they produced little noisy rules. Other patterns were revealed, for example, a variation of Pattern 4 accepting different receivers, however, this option did not seem promising since many noisy rules were produced. Similarly, a variation of Pattern 5 could be added, but it showed likewise to be related with noisy results.

From the delta shown in Figure 4.3-I, we find a rule based on Pattern

1: RPackageOrganizer.default() → RPackage.organizer(). From the delta shown
in Figure 4.3-II, we find a rule based on Pattern 4: Collection.at(3) → Collection.third().

Filtering Rules

In the source code history of real systems many rules may be found. Our goal
is to assess the rules which are likely to be system-specific conventions.

In order to do that, in this final step, we classify the rules according to
their *occurrence over different revisions*, and we only keep the rules that occur
in two or more revisions. Changes that occur in different revisions are those
being incrementally applied by developers, and we want to describe such
changes as rules (*cf.* Section 2.3.1).

In contrast, changes that occur in only one revision are likely to capture
method or class renaming done, for example, with the support of refactoring
tools provided by current IDEs. These cases are not in the scope of the work
presented in this chapter, but they are covered in the next chapter.

4.3   Validation Experiment

In this section we detail our research questions and the experiments that test
them. We first present the proposed research questions (Subsection 4.3.1).
Next, we present the context of our experiment detailing the case studies
and the evaluated change rules (Subsections 4.3.2 and 4.3.3). Finally, in Subsections 4.3.4 and 4.3.5, we formalize the experiment design used to answer
the research questions.

4.3.1   Research Questions

We propose research questions to assess the rules generated by the proposed
approach. We assess the change rule correctness and whether they are likely
to find violations in source code that developers would fix.
Assessing Rule Correctness. We evaluate whether the rules are correct according to the opinion of an expert:

RQ1 *Are the rules correct to the system expert?*

Assessing Rule Violations. The rules may be classified as correct but produce no violation when applied to source code. To complement RQ1, we
also evaluate whether violations identified by the change rules are likely to
be fixed, as suggested by the rule. This will also be validated with the help
of an expert. For example, a violation for the rule Collection.at(3) → Collection.third() is a piece of source code that still contains Collection.at(3); if the

expert decides that this piece of code should be fixed, he will remove Collection.at(3) and add Collection.third(). Thus, we propose another research question:

RQ2 *Are the violations likely to be fixed by the system expert?*

4.3.2 Case Studies

The *context* of the experiment is real systems for which source code history is available. We need real systems to ensure that our experiment is meaningful, and we need source code history to extract our change rules. Moreover, it is fundamental to have access to the experts of the systems to receive assessment.

In this work, we selected two open-source Smalltalk systems, Pharo and Moose to perform our empirical studies. They have the advantage of being large, real-world and non-trivial systems, with a relevant number of developers as well as relevant source code history. Also, they have different missions working in different domains.

Pharo[4] [BDN+09, BCDL13] is an open-source Smalltalk-inspired dynamically typed language and environment. It can be compared to the Java SDK and includes the implementation of all features inherent to an object-oriented language (collections, exceptions, primitive types, etc.) as well as an IDE and several tools. It is currently used in many industrial and research projects[5]. The analyzed version in this study has 374 KLOC, 3,246 classes, and it is supported by 37 developers.

Moose[6] [DAB+11, NDG05] is an open-source platform for software and data analysis written in Pharo. It is composed of several tools to deal with meta-modeling; frameworks to build visualizations, diagrams, interactive browsers; it also includes tools to support common software maintenance tasks such as code duplication detection, identifying dependency cycles, among others. It is currently supported by several research groups around the world[7], and also adopted in industrial projects[8]. The analyzed version in this study has 210 KLOC, 2,617 classes, and it is supported by 21 developers.

We extracted rules from source code changes that occurred during the evolution of Pharo 1.4 to 2.0 (435 revisions from April 2012 to March 2013). Then, we applied such rules in the last release of Pharo 2.0 itself and in Moose.

Furthermore, we have access to an expert of the Pharo language, which is fundamental to receive real assessment about the rules and their violations.

---

[4] http://www.pharo.org
[5] http://consortium.pharo.org
[6] http://www.moosetechnology.org
[7] http://www.moosetechnology.org/about/researchprojects
[8] http://www.moosetechnology.org/about/industrialprojects

The expert selected to support us validating and assessing the rules and the violations is a core developer and release master of Pharo.

Additionally, we also experimented with three Java systems: Ant[9], Tomcat[10] and Lucene[11]. Ant is a tool for automating software build processes, Tomcat is a web server and servlet container, and Lucene is an information retrieval software library. Such experiments are about mining and automatically validating rules (*i.e.,,* without the help of experts) from these Java systems; we discuss them in Section 4.4.

4.3.3 Detecting Rules

We obtain the rules by mining Pharo code changes which incrementally occurred in revisions between versions 1.4 and 2.0. As small deltas between revisions are preferable to avoid the noise that can be found in large ones [LZ05, MWZ11], we select deltas involved in less than five deleted or added invocations. In this process, changes are represented as the deltas described in Subsection 4.2.2 and stored in a database. From such database of deltas, we generated the rules as described in Subsection 4.2.3.

As shown in Table 4.1, this process generated a total of 426 rules considering all patterns. We ranked the rules generated by each pattern by the number of distinct revisions they appear in. Then, we only analyzed the top-15 (*i.e.,* the first 15 in the ranking) rules generated by each pattern; this was done to reduce the amount of rules to be manually analyzed by an expert. To detect the rules relevant for our study (*cf.* Subsection 4.2.3), we selected from the top-15 the ones that occurred in two or more revisions. In the case that the 15th and the 16th rules occurred in the same amount of revisions, they were not considered; this was done to ensure a maximum of 15 rules per pattern. This process generated at the end 45 relevant rules considering all patterns.

Table 4.1: Rules obtained from Pharo.

Pattern	1	2	3	4	5	Total
All Rules	25	31	49	304	17	426
Relevant rules ($\geq$ 2 revisions)	14	11	3	13	4	45

4.3.4 Experiment for RQ1: Assessing Rule Correctness

RQ1 *Are the rules correct to the system expert?*

---

[9]http://ant.apache.org
[10]http://tomcat.apache.org
[11]http://lucene.apache.org

With the help of an expert we validate the rules according to their correctness. We asked the expert to classify the rules as correct or incorrect, a *correct* rule being one that he believes would describe a good modification to apply in the source code.

4.3.5   Experiment for RQ2: Assessing Rule Violations

We detail this experiment following the methodology proposed by Wohlin *et al.* [WRH$^+$00].

Hypotheses Formulation

RQ2 *Are the violations likely to be fixed by the system expert?*

H_0^2 *Number of violations before and after fixing are the same.*

H_a^2 *Number of violations after fixing are smaller.*

Notice that we make a directional (one-tailed) hypothesis.

Variable and Subject Selection

The *subjects* for this experiment are the *violations* generated by rules. First, we take the last version of Pharo and Moose, and we compute the number of violations generated by each rule; this will generate a sample, namely *violations before fixing*. From such sample, we remove the violations that, according to the expert, should be fixed exactly as suggested by the rule; this will generate another sample, namely *violations after fixing*.

The *independent variable* is the rule. It is categorical and takes two values: before or after fixing the violations. The *dependent variable* (measured) is the number of violation for each rule.

Experiment Design

We want to compare the two generated samples. Thus, we use a paired setting, which means the rules composing one sample (before fixing the violations) are the same as those composing the other sample (after fixing the violations). We use the Wilcoxon test which is used for assessing whether one of two samples tends to have smaller/larger values than the other. It can be used when the participants are the same in each sample. The null hypothesis is that the median of violations is the same for both samples. If we cannot reject the null hypothesis, we conclude that there is no statistically significant difference between the number of violations in both samples. The tests will be performed at the 5% significance level. We will also report the *effect size*.

4.4   Experiment Results

In this section, we present the results of our empirical study and discuss
them.

4.4.1   Evaluating RQ1: Assessing Rule Correctness

RQ1 *Are the rules correct to the system expert?*

Table 4.2 shows that 62% (28 out of 45) of the analyzed rules were cor-
rect according to the expert. In Pattern 1, 79% (11 out of 14) of analyzed rules
were correct while in Pattern 3, 1 out of 3 were correct. The incorrect rules
were mostly noisy rules, which are likely to occur when they are extracted
from deltas not related to change conventions. In such deltas, method calls
not involved with change conventions tend to get intermingled with real
rules [LZ05]. The outcome of this experiment is that a relevant amount of
rules is correct according to the expert.

Table 4.2: RQ1: Assessing rule correctness.

Pattern	1	2	3	4	5	Total
Relevant rules	14	11	3	13	4	45
Correct rules	11 (79%)	6 (55%)	1 (33%)	7 (54%)	3 (75%)	28 (62%)

Table 4.3 presents some rules generated for Pharo. For instance, in the rule
RPackageOrganizer.default() → RPackage.organizer(), the method RPackageOr-
ganizer.default() should only be used in specific cases as stated in its com-
ments; the rule vm(). getSystemAttribute(1001) → platform().name() clearly
improves legibility.

All the correct rules were implemented in the static analysis tool Small-
Lint (the static analysis tool for Smalltalk), and they are publicly available[12] to
support developers. Furthermore, the discovering of 28 new system-specific
rules represents a significant addition to the set of rules provided by Small-
Lint, which originally includes only 19 generic change rules that were created
by experts.

4.4.2   Evaluating RQ2: Assessing Rule Violations

RQ2 *Are the violations likely to be fixed by the system expert?*

H_0^2 *Number of violations before and after fixing are the same.*

---

[12]http://www.smalltalkhub.com, Project: FindBugs, Package: MiningLintRules-
PharoMigration

Table 4.3: Examples of Pharo rules.

P. 1	FileDirectory.default() → FileSystem.workingDirectory() RPackageOrganizer.default() → RPackage.organizer()
P. 2	ZnCharacterEncoder.forEncoding(*) → ZnCharacterEncoder.newForEncoding(*) SystemAnnouncer.current() → SystemAnnouncer.uniqueInstance()
P. 3	DataStream.initialize() → MCDataStream.initialize() SystemChangeNotifier.uniqueInstance() → SystemAnnouncer.uniqueInstance()
P. 4	getSource() → sourceCode() noMoreNotificationsFor(*) → unsubscribe(*)
P. 5	vm().getSystemAttribute(1001) → platform().name() vm().getSystemAttribute(1003) → platform().subtype()

H_a^2 *Number of violations after fixing are smaller.*

We set two samples of violations of the rules for Pharo and Moose, the first sample before fixing the real violations, and the second sample after fixing the real violations. Table 4.4 shows the number of rules that produced at least one violation and the total number of violations in each sample. In the last analyzed Pharo release, 8 rules generated at least one violation, producing a total of 21 violations (*before fixing*). The expert pointed that, from such violations, 10 should be fixed, so only 11 violations remained (*after fixing*). In the last analyzed Moose release, 7 rules generated at least one violation, producing a total of 37 violations (*before fixing*). The expert pointed that all violations should be fixed, thus, 0 violations remained (*after fixing*).

Applying the Wilcoxon test in the samples gives a *p-value* < 0.01, then we reject the null hypothesis. Moreover, the *effect size* is $= 0.56$ (large effect). We conclude that the rules are pointing to violations in source code, and, these violations are likely to be fixed by the system expert.

Table 4.4: RQ2: Assessing violations before and after fixing the real ones. Rules refers to the number of rules that produced at least one violation.

System	Rules	Violations	
		Before fixing	After fixing
Pharo	8	21	11
Moose	7	37	0
Total	15	58	11

4.4.3 Java Case Studies

We also extracted and validated rules for the Java systems Ant, Tomcat and
Lucene, which produced 88 rules [HADV13]. The validation of those rules,
in contrast to the previous validation, were done automatically using the ap-
proach developed by [KPW06] on navigating through revisions and checking
if the learned rules were fixed over time. Next we discuss some change rules
generated to the Java systems.

An example of rule generated for Apache Ant is a convention to close
files, where calls to InputStream.close() should be replaced by calls to FileU-
tils.close(InputStream). This convention was introduced in the system in 2004,
but in practice it has never been fully adopted as, even six years later (2010),
the refactoring was still being applied.

In Tomcat, we detected rules defined by FindBugs such as "DM_NUM-
BER_CTOR: Method invokes inefficient Number constructor; use static val-
ueOf instead"[13]. This rule is intended to solve performance issues and it states
that using valueOf is approximately 3.5 times faster than using the construc-
tor. In fact, we detected such rules because Tomcat developers have been us-
ing FindBugs over time. Even if there was an effort to fix such violations, they
were not completely removed. This means that developers may not be aware
of common refactorings even when static analysis tools are adopted. Also,
the great amount of violations generated by such tools in real-world systems
is not easy to manage [RDGN10, FSV11, KE07].

In Lucene, rules were related, for example, to structural changes (*e.g.,*
replace Document.get() by StoredDocument.get()) and to internal guid-
ance to have better performance (*e.g.,* replace Analyzer.tokenStream() by
Analyzer.reusableTokenStream(), replace Random.nextInt() by SmartRan-
dom.nextInt()). Overall, the analysis also produced rules related to Java
API changes such as the replacement of calls from the classes Vector to
ArrayList, Hashtable to Map, and StringBuffer to StringBuilder, which were
incrementally fixed by developers, and, thus, also detected by our approach.

4.5 Complementary Discussion

In this section we discuss three topics complementary to our experiments.
First, we discuss the creation of rules with respect to its frequency over time,
then, we present the overlap of our rules with predefined generic change
rules. Finally, we discuss the creation of other types of system-specific rules
beyond of the ones adopted in this study.

---

[13]http://findbugs.sourceforge.net/bugDescriptions.html#DM_NUMBER_
CTOR

4.5.1 Assessing When Rules Should be Created

In our experiments we extracted rules from a specific timeframe, *i.e.*, the changes in Pharo from version 1.4 to 2.0. Then, we validated such rules in the latest Pharo and Moose releases. However, in practice, we may have no clear timeframe about what should be analyzed in the past code history, the code history may not be available, or the system may be in initial development stage. Therefore, to support these cases, in this subsection we discuss the creation of rules with respect to its frequency over time, *i.e.*, not according to a predefined timeframe.

The idea is to assess when a change is frequent enough to be considered as a rule. This frequency can impact the quality of the produced rules as well as the amount of generated rules. Thus, depending on the goal of the developer (*e.g.*, to produce rules with better precision or to produce more rules), different frequencies can be adopted. Next, we study the impact of such frequency to obtain rules.

Process to learn and evaluate rules

To assess when rules should be created, we need to learn the rules and evaluate them incrementally revision by revision. To learn the rules from source code history, we use the approach developed by [KPW06] on navigating through revisions to extract information. It suits well since it works by learning from changes in revisions. The idea is that we walk through the revision history of a project *learning* rules and *evaluating* at each revision how well our approach works when using only the information available for that revision. We learn a rule when it occurs in f different revisions. We evaluate at revision n the rules learned from revisions 1 to $n - 1$. If the fix in revision n matches the learned rule, *i.e.*, the modification to obtain revision n matches the LHS and RHS, we have a true positive (TP) violation. If the fix in revision n matches the LHS, but not the RHS, we have a false positive (FP) violation. We can measure the precision of a rule from the portion of violations predicted correctly over all violations, *i.e.*, *precision = TP/(TP + FP)*.

Discussion

Table 4.5 (top) shows the precision and the number of rules obtained over the frequency f such that $2 \le f \le 9$ for our case study[14]. For example, if we say that rules are created when the same change occurs over two different revisions (*i.e.*, $f = 2$), then 104 rules are generated and they have a preci-

---

[14]We empirically detected that $f \ge 9$ was not relevant to be studied because not so many rules were generated.

sion of 25%. As expected, the greater the frequency, the more precise are the generated rules, but the smaller the number of generated rules.

Table 4.5 (bottom) shows the delta for the precision and the number of rules. For example, moving from $f = 2$ to $f = 3$ (*i.e.,* Δ_2^3) improves the precision by 36% but reduces the number of generated rules by 58%, which is the greatest loss in number of rules. Moving from $f = 3$ to $f = 4$ (*i.e.,* Δ_3^4) improves the precision by 85% (which is the greatest gain in precision) but reduces the number of generated rules by 46%. Also, moving from $f = 6$ to $f = 7$ (*i.e.,* Δ_6^7) changes neither the precision nor the number of generated rules.

Table 4.5: Top: evaluation of the frequency (f) to create rules. Bottom: delta between frequencies (Δ_{f-1}^f) of precision and number of rules.

Frequency (f)	2	3	4	5	6	7	8	9
Precision	25%	34%	63%	71%	72%	72%	75%	75%
Rules	104	43	23	12	8	8	6	6

Δ Frequency	Δ_2^3	Δ_3^4	Δ_4^5	Δ_5^6	Δ_6^7	Δ_7^8	Δ_8^9
Δ Precision	+36%	+85%	+12%	+1%	0%	+4%	0%
Δ Rules	-58%	-46%	-47%	-33%	0%	-25%	0%

Figure 4.4 shows the curves of precision and number of rules over the frequency. We observe that the curves are clearly inversely proportional, *i.e.,* the precision tend to be greater and the number of generated rules tend to be smaller. *If the number of generated rules is an important goal, then we should not choose a large f.*
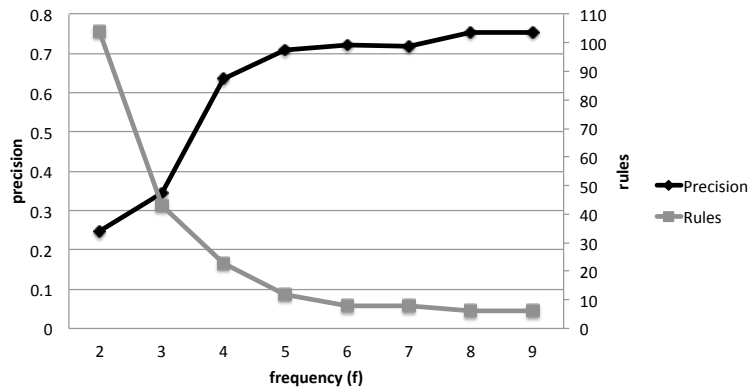
Figure 4.4: Evaluation of the frequency (f) over precision and number of rules.

In Table 4.6 we compare the precisions of the rules generated in each

frequency. This comparison is done by applying the Mann-Whitney test in each pair of rule samples. For example, the *p-value* 0.01 (first line and last column) means that the precision of the rules obtained with $f = 9$ is statistically greater at the 1% level than the precision of the rules obtained with $f = 2$. Also, the *p-value* 0.43 (last line and last column) means that we cannot conclude that the precision for $f = 9$ is greater than the precision for $f = 8$. We marked with * results that are significant at the 10% level and with ** results that are significant at the 5% level.

By analyzing the first line in the table, we see that the precision of rules obtained with $f = 2$ is smaller than any other sample (they are all marked with * or **). When analyzing the second line, we see that the precision of rules obtained with $f = 3$ is smaller than the ones obtained with $f = 8$ and $f = 9$. The same occurs in the third line: the precision of rules obtained with $f = 4$ is smaller than the ones obtained with $f = 8$ and $f = 9$. However, when analyzing the fourth line (*i.e.*, $f = 5$ in **bold**), we see that no other sample is greater than the one obtained with $f = 5$ (none of them are marked with * or **). The same occurs for the subsequent lines. Therefore, it means that the precision of the rules obtained from $f = 5$ to 9 are statistically equivalent. In such samples, the precision remains between 71% and 75% (as we see in Table 4.5). In other words, choosing between $f = 5$ to 9 would bring equivalent results. *If precision is an important goal, we should choose a large f, but only up to a certain point (f=5 in our experiments).*

Table 4.6: *p-values* comparing the precisions of the created rules. ** *p-value* $<$ 0.05, * *p-value* $<$ 0.10.

Frequency (f)	3	4	5	6	7	8	9
2	0.07*	0.05*	0.07*	0.02**	0.02**	0.02**	0.01**
3	-	0.36	0.23	0.11	0.11	0.07*	0.05*
4	-	-	0.25	0.11	0.14	0.05*	0.04**
5	-	-	-	**0.36**	**0.43**	**0.22**	**0.17**
6	-	-	-	-	0.60	0.39	0.43
7	-	-	-	-	-	0.37	0.27
8	-	-	-	-	-	-	0.43

4.5.2 Overlap with Generic Change Rules

As stated in the introduction, the static analysis tool SmallLint also provides generic change rules that were manually created by experts. It contains 19 generic change rules that details how source code should be updated to im-

prove code maintainability.

We found that there is no overlap between our rules and the generic change rules provided by SmallLint. This was also similar to the results provided by [KPW06] in which the intersection of bug-detection rules generated by their approach and rules found in the Java static analysis tool PMD were mostly exclusive. One reason for this result is that the generic change rules were not applied in the analyzed case study over different revisions or they were applied in large commits, so our approach was not able to detect them.

To better understand the adoption of generic rules by developers, we investigated their usage in our case study, Pharo. We detected that 17,821 (27%) methods in version 2.0 were new ones. From such methods, we found that 1,064 (6%) used the suggested RHS of the generic change rules while only 129 (1%) used the non-suggested LHS of the generic rule. It means that new methods were mostly created using (when necessary) the suggested RHS. This shows that these rules are in fact relevant for developers; the rules discovered in our study can improve such set of rules with new system-specific ones.

4.5.3   Creating Other Types of System-Specific Rules

In this subsection we briefly discuss the creation of other types of system-specific rules beyond the one presented in this study. Next, we present three types of system-specific rules that can also be extracted from code history with the support of a slightly different dataset and patterns.

Simply-deleted calls. This type of rule catches methods that in general should not be called. For example, in FindBugs some rules present this idea: "Don't use removeAll() to clear a collection", "Invokes dangerous method System.exit()" and "Invokes dangerous method runFinalizersOnExit()". Similarly, in SmallLint, some rules state: "Debugging code left in methods (avoid using halt(), haltOnce(), etc.)" and "Calls questionable message (avoid using become(), primitiveChangeClassTo(), etc.)". In order to extract such types of rules from code history and expand the set of rules of current static analysis tools, one can use a dataset similar to the one adopted in this study. The patterns would detect deltas that frequently deleted but did not add calls. In this context, some studies (*e.g.*, [WGAK10, MWZM12]) propose the extraction of *simply-deleted* rules, but focusing on API migration, *i.e.*, not adopting the *time* variable to filter out rules and not focusing on system-specific rules.

Method pairs. This type of rule catches methods that normally should appear together in a class. For example, FindBugs, PMD and SmallLint present the rules "Class defines equals() but not hashCode()" and "Class defines hashCode() but not equals()", meaning that the methods equals() and

hashCode() should be defined together. A dataset similar to ours but with the data grouped by signature of methods (instead of method calls) of a class can be adopted to extract this type of rule. The patterns would detect two or more methods that are likely to appear together. Notice that in this case, a data-mining approach (*e.g.*, Apriori [ZJ12]) can also be used to detect the co-occurrence of methods in the dataset.

Method call pairs. This type of rule catches methods that normally should be called together. For example, co-occurring method calls (*e.g.*,open()/close() and lock()/unlock()) are likely to exist in common software development. A dataset similar to ours (but not focusing on the deltas since the *method call pair* rules rely in a single version to be detected) can be adopted to extract this type of rule. The patterns would detect two or more method calls that are likely to appear together. Livshits and Zimmermann [LZ05] propose to extract such type of rule from a single version of a system using a data-mining approach to detect co-occurring method call pairs.

4.6   Threats to Validity

4.6.1   Construct Validity

The construct validity is related to whether the measurement in the study reflects real-world situations. In our study, the main threat is the validation of the rules.

As an error in this process would bias the results, our rules were manually validated with the help of an expert (which is hard to have for real-world systems) to decrease the possibility of bias.

4.6.2   Internal Validity

The internal validity is related to uncontrolled aspects that may affect the experimental results. In our study, the main threat is the possible errors in the implementation of our approach causing the generation of wrong rules.

Apart from the validation by the expert presented in this chapter, the rules generated by our approach have been (i) used by several members of our laboratory in different systems and (ii) divulged in an open-source software reengineering mailing list[15] such that developers of this community can use it, thus, we believe that the risks of this threat are reduced.

Moreover, the patterns used to extract rules from source code history may be an underestimation of the real changes occurring in commits: some changes are more complex, only introducing new code or only removing old

---

[15]http://www.moosetechnology.org

code. We do not extract rules from such cases, and they might also represent relevant source of information. However, the patterns do reflect generic change rules found in static analysis tools.

4.6.3   External Validity

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

Pharo and Moose are credible case studies as they are open-source, real-world, and non-trivial systems with a consolidated number of developers and users. They also come from different domains and include a large number of revisions. Despite this observation, our findings — as usual in empirical software engineering -– cannot be directly generalized to other systems, specifically to systems implemented in other languages or to systems from different domains. Closed-source systems, due to differences in the internal processes, might also have different properties in their commits. Finally, small systems or systems in initial stage may not produce data sufficient to generate rules.

4.7   Summary

In this study, we proposed to automatically extract system-specific conventions from source code history. In this process, we extract information from incremental revisions in source code history and the rules are based on pre-defined patterns that ensure their quality.

We validated our approach on open-source systems with the help of an expert, which was important to provide real assessment about the change rules. In addition, we discussed the creation of rules with respect to their frequency over time and we compared the extracted rules with predefined generic change rules provided by a static analysis tool. We reiterate here the most interesting conclusions we derived from our study:

1. A relevant amount of the change rules (62%, 28 out of 45) were correct to the expert in our Smalltalk case study. We discovered 28 new system-specific rules, which represents a significant addition to the set of rules provided by the static analysis tool SmallLint since it contains only 19 generic change rules.

2. Rules pointed to real violations in source code. In total, 15 rules generated violations, producing a total of 58 violations from which 47 (81%) were real ones.

3. We also extracted and automatically validated rules for three Java systems. Thus, with the Smalltalk experiment, case studies written in two programming languages were covered.

4. There was no overlap between our rules and the generic change rules provided by SmallLint.

5. Different frequencies can be adopted, depending on the goal of the developer, to assess when rules should be created.

The rules generated by the proposed approach are related to system-specific conventions. Yet, there is a lack as to better support client systems as we have seen in Chapter 2. Moreover, the rules are based on predefined patterns; while simple to be created, the patterns will generate rules restricted to them. In addition, as pointed by our discussion on Section 4.5.1, it is important to discover thresholds to assess when the rules should be created. Thus, it would be relevant to produce more flexible rules, *i.e.,* independent of predefined patterns and less dependent on thresholds.

The next chapter presents an approach to mine rules from source code history with the goal to support client systems and without the need of using predefined patterns.

Supporting Client Systems with History-based Rules

5

Contents

In the previous chapter we presented an approach to extract system-specific conventions from source code history based on predefined patterns. In this chapter we complement the previous one: we propose an approach to mine rules from source code history by focusing on better supporting client systems (rather than system-specific conventions) and by using data-mining (rather than predefined patterns).

5.1   Introduction

As described in Chapter 2 in Section 2.3.2, frameworks often evolve by breaking compatibility with clients [BTF05,DJ06,WGAK10]. At the ecosystem level, this problem may be even more complex and risky [RLR12]. To deal with such issues, approaches have been developed to support the evolution of frameworks, and reduce the efforts of client developers. In this context, the identification of evolution rules has become a research focus: some researchers demonstrated the need to automatically identify evolution rules of the types *one-to-one* [DR08,SJM08], *one-to-many*, *many-to-one* [WGAK10], and *many-to-many* [MWZM12].

In these rules, the replaced method can be either absent or present in the new release of the framework. Rules that involve methods still present in the new release of the framework are relevant, for example, to the detection of better APIs, missing deprecations, or any replacement involving methods

not necessarily removed during the framework evolution, *i.e.*, they are about method suggestion. However, currently, only the *one-to-one* approaches generate evolution rules involving method suggestion. While the other approaches produce more types of rules, they rely on deletion/addition of methods as the basis for detecting rules, so they are able to produce rules about method replacement (*e.g.*, method renaming) but they cannot produce rules about method suggestion.

In this chapter, we propose to extract evolution rules by monitoring changes applied in source code during framework or library evolution using a data-mining technique [HEA+14, HAE+15a]. Similarly to the previous chapter, we focus on mining method call changes taking into account the changes between revisions. However, in this chapter, our approach is able to produce more flexible rules, in the formats *one-to-one*, *one-to-many*, *many-to-one* and *many-to-many*, involving method replacement and suggestion.

The proposed approach is validated on five open-source systems, and the validity of our rules is assessed with the help of experts. The results show that rules can be extracted from source code changes in code history, and that they are likely to enforce the correct changes to be applied. We detected a relevant amount of rules related to method suggestion, thus, our approach can improve the spectrum of rules generated by pre-existing solutions. Furthermore, we provide a complementary discussion on the effort to select correct rules and the gain provided by the rules in the case studies themselves.

The main contributions of this chapter can be summarized as follows:

1. We provide a novel approach to support the evolution of frameworks and libraries through evolution rules.

2. We provide a qualitative evaluation of the rules in real-world systems to assess whether they are valid according to the assessment of experts.

3. We provide a comparison of the rules with related work.

4. We provide a discussion around the effort to select correct evolution rules and the gain provided by them.

Structure of the Chapter

We present our approach in Section 5.2. We propose research questions and present case studies in Section 5.3. We present the experiment design and answer the research questions in Sections 5.4 and 5.5. In Section 5.6, we present concrete examples of rules found by our approach and discuss additional issues regarding effort and gain. We present the threats to the validity of our experiments in Section 5.7. Finally, we conclude the chapter in Section 5.8.

5.2 Mining API Change Rules

In this section we present our approach to extract rules. Before detailing our solution, we show an overview of it.

5.2.1 Overview of the Approach

Consider the two examples shown in Figure 5.1 that occurred in Pharo and Moose. Figure 5.1-I shows an example of replacement of the call ClassOrganizer.default() by the call Protocol.unclassified() due to a class and method renaming. Figure 5.1-II shows an example of the replacement of the calls MooseModel.root() *and* MooseModel.add(MooseModel) by the call MooseModel.install() to improve maintenance and clean the code. These two changes occurred several times in different revisions of these systems. Our goal is to ensure that such changes are not lost, and can be described as rules to support clients of these systems (*i.e.*, users of the API).

I. Replace ClassOrganizer.default() by Protocol.unclassified()
Diff between revisions 1 and 2 of method foo()
− if (method.protocol() == ClassOrganizer.default()) {...
+ if (method.protocol() == Protocol.unclassified()) {...
II. Replace MooseModel.root().add(...) by install()
Diff between revisions 3 and 4 of method bar()
− self.add(MooseModel.root().add(model));
+ self.add(model.install());

Figure 5.1: Examples of changes in Pharo and Moose. "−" indicates the deleted lines and "+" indicates the added lines. Bold indicate the changed method calls (code is converted to Java-like syntax to ease understanding).

Our approach produces rules that indicate how method calls should be replaced. There are two ways to produce rules:

1. On request rules: After a framework upgrade there are compilation errors because some methods were removed. The developer asks to our tool what to do with the offending method call. He receives on request rules about how this particular method call should be replaced.

2. List of rules: Method calls that should be replaced are automatically extracted from code history. The developer will receive automatically created rules about the overall API evolution. For example, rules to ensure the changes shown in Figure 5.1. This scenario includes cases of method replacement.

Figure 5.2 shows an overview of our approach (it shares some similarities with the approach described in the previous chapter, so a comparison between both approaches is presented in Subsection 5.6.4). In the first step, we extract changes from revisions in a system history (Subsection 5.2.2). In the second step we discover rules based on the extracted changes and the requests. This step is performed each time a rule is requested and it is divided in two substeps: (i) it selects changes related to the provided request (Subsection 5.2.3) and (ii) it creates rules based on the selected changes (Subsection 5.2.3). We first explain these steps in the context of on request rules, then in Section 5.2.4 we explain how to produce a list of rules. Finally, in Subsection 5.2.5 we briefly describe our tool APIEvolutionMiner, which implements the proposed approach.
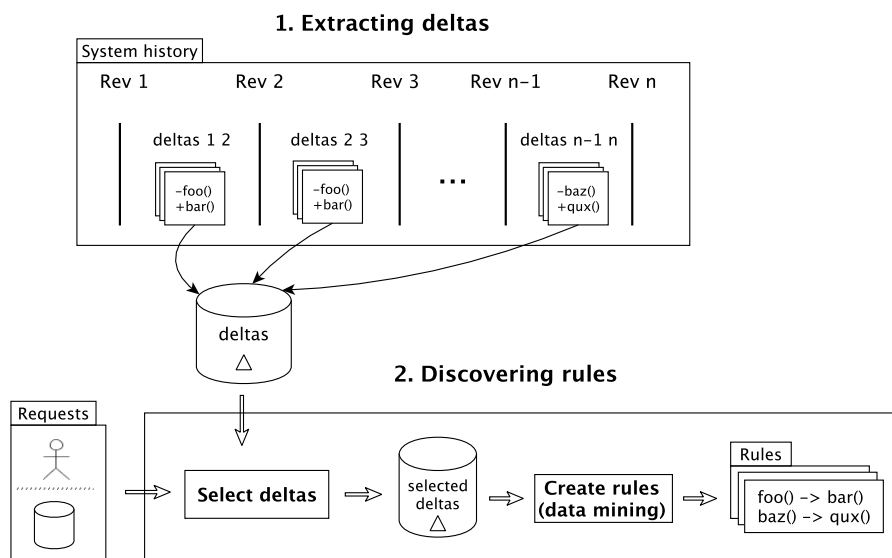
Figure 5.2: Overview of our approach.

5.2.2   Extracting Deltas from Revisions

Similarly to the previous chapter, the first step is to extract deltas from the revisions that will be used in the rule discovering process. We represent a delta with predicates that describe deleted or added method calls:

deleted-call(context-id, signature)

added-call(context-id, signature)

where the predicate *deleted-call(...)* represents a deleted call; the predicate *added-call(...)* represents an added call; *context-id* uniquely identifies the change to save its context; and *signature* is the signature of the method with declaring module, name, and parameter list.

Notice that the delta presented in this approach is slightly different from the one presented in the previous chapter, *i.e.*, here, the *signature* includes the declaring module, and there is no *receiver* and *static* variables. Such changes are needed to facilitate the input of the data-mining algorithm, presented in the next subsection. In fact, the delta adopted in this chapter can be derived from the delta presented in the previous chapter.

Figure 5.3 shows the deltas generated for the changes in Figure 5.1. For example, the change in Figure 5.1-II has two deleted calls (MooseModel.root() and MooseModel.add(MooseModel)) and one added, MooseModel.install(). Thus, two deleted predicates and one added predicate are generated as shown in Figure 5.3-II.

I. Replace ClassOrganizer.default() by Protocol.unclassified()

Deltas between revisions 1 and 2 of method foo()
    deleted-call("foo()-rev2", "ClassOrganizer.default()")
    added-call("foo()-rev2", "Protocol.unclassified()")

II. Replace MooseModel.root().add(...) by install()

Deltas between revisions 3 and 4 of method bar()
    deleted-call("bar()-rev4", "MooseModel.root()")
    deleted-call("bar()-rev4", "MooseModel.add(MooseModel)")
    added-call("bar()-rev4", "MooseModel.install()")

Figure 5.3: Deltas generated for the changes in Figure 5.1.

5.2.3 Discovering Rules

The process to produce rules starts with a method call that needs to be replaced, for example, because it does not compile anymore. Given this call, we will extract rules that suggest how to replace it. Next, we describe the steps to discover rules.

Selecting Deltas

We name as *request* the method call that should be replaced. Given a request, we select from all deltas: (i) the deleted calls that match the request, and (ii) the added calls:

select-deltas(request) ⇒ select all deltas where deleted calls include the requested call

For example, for the request ClassOrganizer.default(), the delta shown in Figure 5.3-I is returned:

select-deltas({ClassOrganizer.default()}) $\Rightarrow$
    deleted-call("foo()-rev2", "ClassOrganizer.default()")
    added-call("foo()-rev2", "Protocol.unclassified()")

Furthermore, a request can contain more than one method. For example, for a request with MooseModel.root() and MooseModel.add(MooseModel), the delta shown in Figure 5.3-II is returned:

select-deltas({MooseModel.root(), MooseModel.add(MooseModel)}) $\Rightarrow$
    deleted-call("bar()-rev4", "MooseModel.root()")
    deleted-call("bar()-rev4", "MooseModel.add(MooseModel)")
    added-call("bar()-rev4", "MooseModel.install()")

Creating Rules

In this step we mine the selected deltas in order to produce rules. In this study we adopt the Apriori algorithm [ZJ12]. This algorithm has been adopted because it is intended to produce association rules (which highlight general trends in a database), and our goal is to produce rules about method call replacement.

Before mining the rules, we represent the selected deltas as transactions[1]. For example, the two selections shown in the previous subsection are simply represented as:

T_1: *"deleted ClassOrganizer.default()", "added Protocol.unclassified()"*
T_2: *"deleted MooseModel.root()", "deleted MooseModel.add(MooseModel)", "added MooseModel.install()"*

The transactions are first analyzed using the data-mining technique *frequent itemset mining*. Given a set of transactions, this technique identifies the itemsets which are subsets of at least n transactions. It defines *support* as the number of occurrences of an itemset. An itemset is considered frequent if its support is greater than or equal to a specified threshold called *minimum support* (for short, we use *min-supp*):

find-frequent-itemsets(transactions, min-supp) $\Rightarrow$ returns the itemsets in *transactions* with support $\geq$ *min-supp*

For transaction T_1, for example, *find-frequent-itemsets({T_1}, 1)* produces three frequent itemsets, each one with *support*[2] *= 1*, as shown in Table 5.1.

Traditionally, from the frequent itemsets, *association rules* [ZJ12] are computed. An association rule is defined as $L \rightarrow R$, where L and R are itemsets.

---

[1]A transaction contains all the method calls of the selected delta identified by the predicate type (deleted or added). The *context-id* is not presented in the transactions.

[2]It means a relative support of 100%, *i.e.*, each itemset occurs in all transactions.

Table 5.1: Frequent itemsets for transaction T_1.

Id	Frequent itemset	Support
I_1	*"deleted ClassOrganizer.default()"*	1
I_2	*"added Protocol.unclassified()"*	1
I_3	*"deleted ClassOrganizer.default()", "added Protocol.unclassified()"*	1

Also, $R = X - L$, where X is a frequent itemset, then, an association rule can be written as $L \to X - L$. For each frequent itemset X and non-empty subset $L \subset X$, an association rule is generated. Moreover, an association rule has a *confidence*[3], which is the probability of finding the R in transactions under the condition these transactions also contain L. The *confidence* is calculated as *confidence(L $\to$ R) = support(L $\cup$ R) / support(L)*. Given a set of itemsets and a *minimum confidence* (for short, we call *min-conf*) indicating the minimum accepted confidence, association rules are produced:

find-assoc-rules(itemsets, min-conf) $\Rightarrow$ returns the association rules in *itemsets* with confidence $\geq$ *min-conf*

In our approach, we want to produce rules in the specific format *Requests $\to$ Replacement*, where *Requests* must include only itemsets with deleted calls, and *Replacement* must include only itemsets with added calls. According to the association rule definition in the previous paragraph, we can consider that *Replacement = X $-$ Requests*, thus, in our approach, X must include only frequent itemsets with both deleted and added calls. Finally, for the generated association rules, we select the ones in the format of interest: *Requests $\to$ Replacement*. If more than one replacement is suggested for a request, we present the one with greatest support and confidence, and which is not a subset in the suggested replacement.

For example, in Table 5.1, only I_3 is considered relevant to our approach, since it satisfies the condition to include both deleted and added calls. Thus, if X is the frequent itemset I_3, *find-assoc-rules({I_3}, 1)* produces the following rule with *confidence*[4] *= 1*:

"deleted ClassOrganizer.default()" $\to$ *"added Protocol.unclassified()"*

The overall process for generating rules in the format *Requests $\to$ Replacement* is sketched in Figure 5.4.

Usually the Apriori can generate a lot of association rules, many of them

---

[3]Other metrics such as lift could have been adopted, *lift(L $\to$ R) = confidence(L $\to$ R) / confidence($\emptyset$ $\to$ R)*. As we are not interested in every association rule, but only in the ones where the antecedent is known, lift will be always 1, thus, not relevant in our study.

[4]*i.e.*, 100% of the transactions that contain *deleted ClassOrganizer.default()* also contain *added Protocol.unclassified()*.

relevant-assoc-rules(requests, min-supp, min-conf) {
    transactions = ***select-deltas****(requests)*;
    frequent-itemsets = ***find-frequent-itemsets****(transactions, min-supp)*;
    relevant-itemsets = select the itemsets in *frequent-itemsets* that includes
                      both deleted and added calls;
    rules = ***find-assoc-rules****(relevant-itemsets, min-conf)*;
    return *rules* }

Figure 5.4: Overall process to generate rules *Requests → Replacement*.

confliting ($A \rightarrow B$ and $B \rightarrow A$) or specialization/generalization of each other
($A, C \rightarrow B$ and $A \rightarrow B$). By adding constraints on the format of the rules (*Requests → Replacement*) we can eliminate many of these unwanted rules.

The rules may have very different support values since they are computed only from the selected deltas, *i.e.*, not directly from changes in the entire system. This happens because the *minimum support* will be relative to the selected deltas. Rules with small support (that can still be relevant to developers) would not be detected if they were computed directly from changes in the entire system.

Furthermore, by adopting association rules, in particular, in the format *Requests → Replacement*, our approach can produce *one-to-one*, *one-to-many*, *many-to-one* and *many-to-many* rules, involving method replacement and suggestion since it does not use deletion/addition of methods as the basis for detecting rules, *i.e.*, we focus on the addition and deletion of methods calls.

5.2.4   Creating a List of Rules

As stated in the begining of this section, there are two ways to generate rules: on request and by listing the rules. On request rules are produced when requests are provided to relevant-assoc-rules(. . .) by the developer.

In contrast, a list of rules is produced when the requests are automatically extracted from code history and provided to relevant-assoc-rules(. . .) without involving the developer. In this case, we consider as *request* the deleted method calls from all deltas. Each request will have a *number of occurrences*, which is the number of times it occurred over the deltas.

For example, from the deltas in Figure 5.3, two requests can be automatically extracted, both with *number of occurrences = 1*:

R_1: {*"ClassOrganizer.default()"*}

R_2: {*MooseModel.root()"*, *"MooseModel.add(MooseModel)"*}

When provided to relevant-assoc-rules(. . .), the requests produce the following rules, which in fact describe their real changes in source code:

$Rule_{R1}$: *"deleted ClassOrganizer.default()"* → *"added Protocol.unclassified()"*

$Rule_{R2}$: *"deleted MooseModel.root()"*, *"deleted MooseModel.add(MooseModel)"* →
"added MooseModel.install()"

In real systems, the number of requests (and consequently the number of rules) generated may be large. Thus, to filter the most relevant requests, we can rank them by the decreasing order of *number of occurrences*, and produce rules for the most frequent requests. Notice that the *number of occurrences* here is not related to the *support* described in the previous subsection.

5.2.5 Tool Support

Our approach is supported by the tool APIEvolutionMiner [HEA$^+$14] shown in Figure 5.5. The developer can (i) provide requests (1. Input pane) in order to receive as output an association rule (2. Association rule pane), or (ii) access a list of automatically generated rules. In both cases, by selecting the association rule, the developer is able too see a list with all the deltas in which the rule was found (3. Delta pane). Each delta is linked to a source code change. Thus, by selecting a delta, the tool displays source code examples (4. Example helper pane) that clarify how the framework adapted to its own changes or clients adapted to the framework changes as well as a description of the delta with the class, method, author and date of the change.
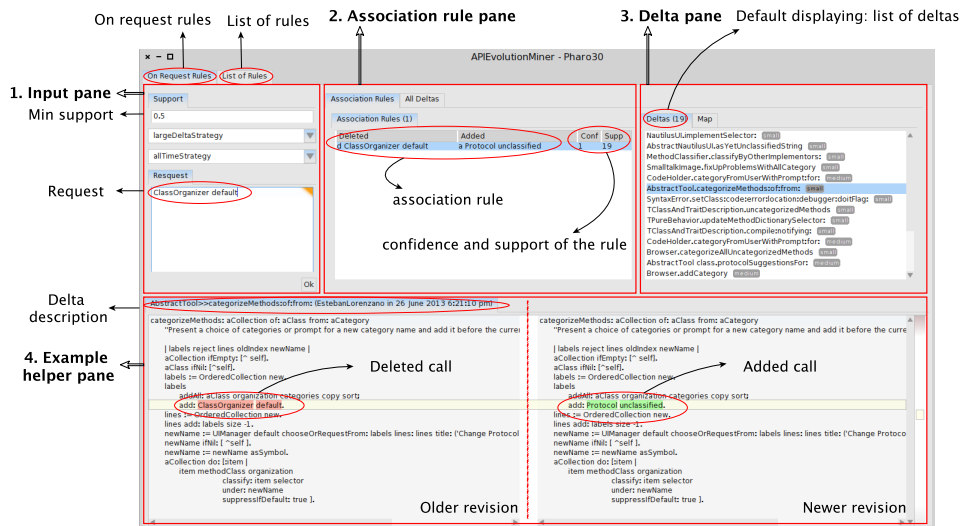
Figure 5.5: Tool support with APIEvolutionMiner.

5.3 Research Questions and Case Studies

In this section we present our research questions (Subsection 5.3.1) and the context of our experiment, detailing our case studies (Subsection 5.3.2).

5.3.1   Research Questions

In the experiments we validate the list of rules automatically generated by our approach. We assess the rules validity according to assessment of experts, and whether our approach is able to produce rules *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many* involving method replacement and suggestion, *i.e.,* the types of rules generated. Therefore, we propose two research questions:

RQ1 *Are the generated rules valid to experts?*

RQ2 *Which types of rules are generated?*

5.3.2   Case Studies

The *context* of the experiment is real systems for which source code history is available. We need real systems to ensure that our experiment is meaningful, and we need source code history to extract our rules. Moreover, it is fundamental to have access to the experts of the systems under analysis to receive real assessment about the rules.

We selected five open-source Smalltalk systems to perform our empirical studies: Pharo [BCDL13, BDN+09], Moose [DAB+11, NDG05], Glamour [Bun09][5], Roassal, and Seaside [DLR07]. They are large, real-world and non-trivial systems, with relevant number of developers as well as relevant source code history. Moreover, they have different missions working in different domains:

Pharo is an open-source Smalltalk-inspired dynamically language and environment (it is described in the Chapter 4 in Subsection 4.3.2). We study two evolutions of this system, one from version 1.4 to 2.0 (for short, we call Pharo2), and the other from version 2.0 to 3.0 (for short, we call Pharo3).

Moose is an open-source platform for software and data analysis (it is described in Chapter 4 in Subsection 4.3.2).

Glamour is an engine for building dedicated browsers.

Roassal is an agile visualization engine which graphically renders objects with interaction facilities.

Seaside is an open-source framework for developing web applications (it is described in Chapter 3 in Subsection 3.3.2).

Table 5.2 shows an overview about the six case studies. It shows the number of classes and lines of code in the last version, the number of contributors (commiters), dates of the first and the last analyzed revisions, the number of revisions, and the number of deltas, *i.e.,* method deltas in which at least one call is deleted and one is added.

---

[5]http://www.moosetechnology.org/tools/glamour

Table 5.2: Case studies overview. *Cont.* represents the number of contributors.

System	Classes	KLOC	Cont.	First rev.	Last rev.	Revisions	Deltas
Pharo2	3,246	374	37	12/2011	03/2013	435	3,196
Pharo3	3,844	412	37	09/2012	11/2013	483	4,729
Moose	2,617	210	21	05/2008	11/2013	512	2,848
Glamour	1,117	81	24	02/2009	11/2013	598	1,807
Roassal	411	29	6	07/2011	11/2013	379	1,559
Seaside	1,122	97	18	11/2007	11/2013	862	3,962

Furthermore, we have access to experts of most of these systems, which is hard to find for real-world systems and is fundamental to receive real assessment about the rules to answer research question RQ1. The experts selected to help us assessing the rules are all core developers or creators of the systems under analysis. We had the help of 2 experts for Pharo2, 4 for Pharo3, 4 for Moose, 1 for Glamour, and 1 for Roassal. We have no close access to Seaside experts, so the rules of this system were validated by the authors of this study.

5.4 RQ1. Are the generated rules valid to experts?

5.4.1 Experiment Design

In this experiment we will evaluate the validity of the rules produced as described in Subsection 5.2.4 (list of rules) based on the expert assessment. We attempt to produce rules for each request with *number of occurrences > 5*, setting a mininum support and confidence of 50%. Thus, we asked the experts to classify the rules on: *valid*, *don't know* or *invalid*. The correctness of the rules are then assessed using *precision*, *i.e.,* the percentage of valid rules over all generated ones, which is commonly adopted by related studies. Rules classified as *don't know* are not used to compute the precision. For the Seaside case study, the rules will be validated only by the authors of this study with the support of code examples and documentation.

Notice that, in previous related studies (*e.g.,* [MWZM12, WGAK10]) the validation is always done by the authors of the paper themselves. We believe that a validation by experts, which is harder to be done, increases the confidence of the results.

5.4.2   Experiment Results

Table 5.3 shows the number of *valid*, *invalid*, and *don't know* rules as classified by experts of the systems under analysis. We see that the precision remained between 46% (for Pharo2) and 86% (for Moose).

We note that for Pharo2 and Pharo3, 25 and 3 rules, respectively, were classified as *don't know* by the experts. This reinforce the importance of the validation by experts: some rules are hard to be evaluated and involve specific knowledge about some parts of the system. In fact, Pharo2 is a legacy system, so the experts were not confident to point rules as *valid* or *invalid*; as a result, the precision for this system was low (46%). Pharo3 is a very large system, and, naturally, some rules were classified as *don't know*; for this system the precison is 74%. In Moose, Glamour and Roassal, the experts were confident to point the *valid* and *invalid* rules; in these cases the precision remained between 65% and 86%. Seaside was the only system in which the rules were evaluated by the authors of the study and it has a precision of 80%.

Table 5.3: RQ1. Evaluation of the rules.

System	Requests	Rules	Valid	Invalid	Don't know	Precision
Pharo2	77	62	17	20	25	46%
Pharo3	114	95	68	24	3	74%
Moose	54	50	43	7	0	86%
Glamour	31	23	15	8	0	65%
Roassal	43	36	28	8	0	78%
Seaside	*93*	*76*	*61*	*15*	*-*	*80%*

The invalid rules were mostly noisy rules. These rules occur when they are extracted from large deltas (*i.e.*, noisy deltas which include large number of deleted and added calls). Smaller deltas tend to produce less noise [LZ05, MWZ11]. Thus, a solution to decrease the amount of invalid rules is to constraint the size of the deltas. This has the drawback of producing less rules.

For the next experiment, we do not take into account the rules classified as invalid.

5.5   RQ2. Which types of rules are generated?

5.5.1   Experiment Design

In this experiment we will verify which types of rules are generated by our approach in order to classify them in *one-to-one*, *one-to-many*, *many-to-one*, and

many-to-many. We also verify whether the rules involve (i) method replacement, or (ii) method suggestion. A rule is about suggestion methods when the methods in both the right and the left side are still present in the last analyzed version of the case study. This will allow us to discuss the types of rules generated by our approach with the ones generated by related studies.

5.5.2   Experiment Results

Table 5.4 shows the number of rules in each category. Our approach is the first to generate rules for the four types of rules.

More specifically, 53% of the rules are concentrated in the category 1 (*i.e.,* method replacement *one-to-one*), which is covered by previous studies. Categories 1 and 2 (*i.e., one-to-one*) represent 84% while categories 1 and 3 (*i.e.,* rules about method suggestion) represent 56% of the rules. Category 3, which represents 3%, is the lowest in number of rules; this result is comparable to the results provided by [MWZM12] in which this category represented only 1% of their rules. Each of these three categories are *separately* covered by some previous studies, but none of them cover categories 1, 2 and 3 *together* (*cf.* Section 2.3.2). Furthermore, category 4 (*i.e.,* method suggestion *one-to-many, many-to-one, many-to-many*) represents 13% of the rules; this category is not covered by any previous study.

Table 5.4: RQ2. Number of rules in each category.

System	Rules	one-one		one-to-many, many-to-one many-to-many	
		(1) suggestion	(2) replac.	(3) suggestion	(4) replac.
Pharo2	42	17 (40%)	16 (38%)	1 (2%)	8 (19%)
Pharo3	71	23 (32%)	26 (37%)	3 (4%)	19 (27%)
Seaside	61	41 (67%)	15 (25%)	3 (5%)	2 (3%)
Moose	43	27 (63%)	14 (33%)	0 (0%)	2 (5%)
Glamour	15	7 (47%)	7 (47%)	0 (0%)	1 (7%)
Roassal	28	23 (82%)	2 (7%)	0 (0%)	3 (11%)
Total	260	138 (53%)	80 (31%)	7 (3%)	35 (13%)

5.6   Complementary Discussion

In this section we discuss three complementary topics. First we discuss and show examples of rules about method replacement and suggestion in order to better understand the differences between them (Subsection 5.6.1). Second, we discuss the effort to produce valid rules with a classical data-mining ap-

proach (Subsection 5.6.2). Third, we discuss how distributed are the changes (which could be described as rules) over different revisions, and whether rules based on these changes would support developers finding violations (*i.e.*, inconsistencies in source code that should be fixed as stated by the rule) in the case studies themselves (Subsection 5.6.3). Finally, we compare the approach presented in this chapter with the one presented in the previous chapter (Subsection 5.6.4).

5.6.1   Rules about Method Replacement and Suggestion

Method replacement rules. These rules involve removed, renamed or explicitly deprecated methods. In general, the left side of the rule is likely to be invalid or will be invalid soon (in the case of explicitly deprecated methods and classes). Note that, to discover rules related to explicitly deprecated methods, one could think about interpreting their deprecation messages. These annotations can include a deprecation message that gives a recommendation for the replacement. However, in practice, recommendations are often missing or unclear on how to replace the deprecated methods or classes [RLR12]. This highlights the importance of extracting rules from code history. There is also cases in which methods are simply renamed without deprecation. For example, in the Moose migration to Pharo 3.0, a client developer noticed this issue and commented[6]: "In FileSystem, ensureDirectory() was renamed to ensureCreateDirectory() without a deprecation", the language developer then answered: "Fill up a bug entry and we will add this deprecation. Good catch". In fact, in some case, asking in mailing lists is the only alternative for client developers[7]. Our approach described such case as the rule ensureDirectory() → ensureCreateDirectory(), so we are able to help developers in similar cases.

Method suggestion rules. These rules are related to how systems use some APIs either due to usage convention or to adapt to a better API. In general, these rules ensure consistency in source code or the right side of the rule is simply a better option (*e.g.*, improving performance, code legibility, portability). Again, currently, client developers may use mailing lists for coordination[8]. Next, we discuss some examples, which are shown in Table 5.5.

Pharo2. Rule 1, intersect(*) → intersectIfNone(*,*), replaces a method with a more robust one, that allows one to provide a different behavior when the intersection is empty.

Pharo3. Rules 2 and 3 improve code legibility, as it replaces two method

---

[6] http://goo.gl/jKrHPb

[7] Examples of mailing coordination involving API changes about method replacement: http://goo.gl/k9Fl0K and http://goo.gl/UVO910.

[8] Examples of mailing list coordination involving API changes about method suggestion: http://goo.gl/50q2yZ, http://goo.gl/SkMORX and http://goo.gl/ZgvsVF.

Table 5.5: Examples of rules about method suggestion.

System	Rule
Pharo2	1 intersect(*) → intersectIfNone(*,*)
Pharo3	2 isNil().ifTrue(*) → ifNil(*)
	3 Scanner.new().scanTokens(*) → parseAsLiteralToken()
	4 UserManager.default().currentUser() → Smalltalk.tools().userManager()
	5 keys().do(*) → keysDo(*)
Seaside	6 beginsWith(*) → beginsWithSubCollection(*)
	7 registerAsApplication(*) → WAAdmin.registerAsApplicationAt(*,*)
Moose	8 MooseModel.root().add(*) → install()
Glamour	9 assert(equals(*,*)) → assertEquals(*,*)
Roassal	10 ifEmpty(*) → isEmpty().ifTrue(*)
	11 Character.cr() → ROPlatform.current().newLine()

calls by a single, clearer one. Rule 4 is about using the class Smalltalk as a central entry point to the system; in fact, Smalltalk class acts as a facade to access the system. In Rule 5, keys().do(*) → keysDo(*), the use of keysDo(*) avoids creating an extra collection in the iterating process (thus, saving memory) as pointed by an expert.

Seaside. Rule 6, beginsWith(*) → beginsWithSubCollection(*), is a convention to support Seaside and its clients to be platform-independent. Rule 7 is a convention to make the call more explicit by using a static method.

Moose. Rule 8, MooseModel.root().add(*) → install(), represents an evolution of the API to deal with the addition of models. Even if both options are valid, the latter is clearer and improves code legibility.

Glamour. Rule 9, assert(equals(*,*)) → assertEquals(*,*), represents the use of a better suited unit test API.

Roassal. In the Pharo language, one expects the rule isEmpty().ifTrue(*) → ifEmpty(*) to test empty objects in conditional statements. However, in Roassal, the convention is the opposite, so it generated the Rule 10: ifEmpty(*) → isEmpty().ifTrue(*). The expert pointed that Roassal adopts this usage convention because the system (and its clients) should be portable over different Smalltalk platforms, and as ifEmpty(*) is Pharo-specific, it should not be used. Finally, Rule 11, Character.cr() → ROPlatform.current().newLine(), is about a new way to represent the *carriage return* also to ensure portability of platform.

5.6.2 Effort to Produce Rules

In this subsection we discuss the effort to produce valid rules with our approach and with a classical data-mining approach. In our approach, we apply

two steps: first *frequent itemset mining*, and second *association rules generation* on some selected itemsets during the rule discovering process (see Subsection 5.2.3). In contrast, using the classical data-mining approach to discover rules would mean to apply these steps on all itemsets.

In order to generate rules using the classical approach, we compute association rules directly from the changes of the entire system. In this case, we only considered associations rules in which the left side contained deleted calls and the right side contained added calls since this is the format used in our work (*i.e.,* Evidences → Replacement). If all formats were considered, very few or even no valid rules would be produced in this approach given the great amount of rules generated by large datasets. Thus, we ranked the rules in decreasing order of *support* and *confidence* values.

For each case study, we evaluated the same number of rules of our experiment (see Table 5.3). For example, for the Pharo2 case study, there are 62 rules, thus, we evaluated for the classical approach the first 62 association rules in the ranking, *i.e.,* the top-62. We reused the validation of the experts to validate the rules obtained with the classical approach; for the cases in which the classical approach produced a rule not previously validated by the experts, the authors of the study validated them with the support of code examples and documentation.

Table 5.6 shows the number of valid rules for each approach[9]. We see that our approach generated more valid rules for all the case studies. Overall, the classical approach performs worse because, for large frequent changes (*i.e.,* changes that have more than one deleted or added calls), many rules are created that are permutations or subsets of the elements of the itemset, thus, generating noisy rules. On the other hand, for the case of large frequent changes, our approach filters noisy rules keeping only the largest change itself (*cf.* Subsection 5.2.3).

When there is no large frequent changes, both approaches can be equivalent in terms of rules they can find (recall that we applied an extra filter to the classical approach to avoid producing rules in the wrong format "Replacement → Evidences", which would be meaningless). For example, for Moose, even if our approach generated more rules (*i.e.,* 43 against 39), the difference is small. However, this was not the case for the other five case studies; *e.g.,* only 4 rules were found in Glamour for the classical approach while our approach found 15.

In Table 5.6 we also see in parentheses the number of rules only detected by the classical approach. It shows that the rules generated by the classical

---

[9]In this experiment, we also consider as valid the rules classified as *dont't know* by the experts. Notice that it does not impact the results of this experiment since this consideration is done for both approaches.

Table 5.6: Number of valid rules both classical data-mining and our approach. In parentheses the number of rules only detected by the classical approach.

	Pharo2	Pharo3	Seaside	Moose	Glamour	Roassal
Rules	62	95	76	50	23	36
Classical approach	23 (0)	28 (1)	45 (5)	39 (0)	4 (0)	19 (2)
Our approach	42	71	61	43	15	28

approach were mostly the ones found by our approach. For example, from the 45 rules detected using the classical approach in Seaside, 5 rules were not detected by our approach. In Pharo2, Moose and Roassal there were no rule only detected by the classical approach while in Pharo3 and Roassal there were 1 and 2 rules, respectively.

Finally, Figure 5.6 compares the evolution of the precision in the top-x rules. Overall, for our approach, precision remains mostly constant (with variations when very few rules are considered). Furthermore, it clearly confirms that the precision of our approach is better than that of the classical approach.

5.6.3 Change Analysis

In this subsection we verify how distributed are the changes, which could be described as rules, over different revisions. Then, we verify whether rules created from these changes would support developers in the task of finding violations (*i.e.*, inconsistencies in source code that should be fixed as stated by the rule) in the case studies themselves.

Change Distribution Analysis

Figure 5.7 shows the distribution of the changes over different revisions. More specifically, we see the number of revisions versus the number of changes in two levels of granularity. In the outer chart, changes are shown by revisions while in the inner chart, we consider two categories: changes occurring in a single revision and changes occurring in several revisions.

The outer chart shows that a great amount of changes occur in only one revision. In fact, this happens thanks to refactoring tools found in current IDEs. However, it is also true that many changes are spread over time in different revisions. For instance, in Pharo3, we see a change that occurs over 13 revisions, and in Moose another change occurs over 17 revisions. The inner chart shows that when revisions are grouped by one vs. several (*i.e.*, >1) revisions, the latter becomes more noticeable. More specifically, the group with
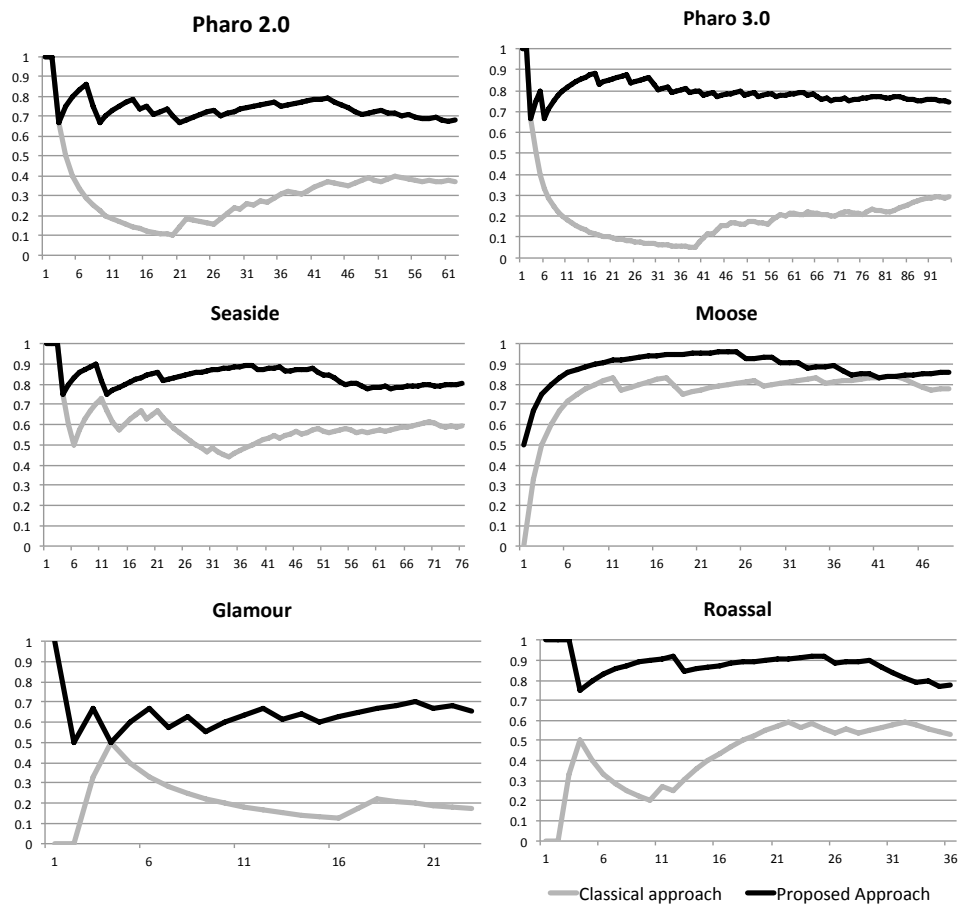
Figure 5.6: Precision (y-axis) in the top-x rules (x axis) for both classical data-mining and proposed approach.

several revisions represents 45% of the rules for Pharo2, 42% for Pharo3, 39% for Seaside, 60% for Moose, 47% for Glamour, and 57% for Roassal.
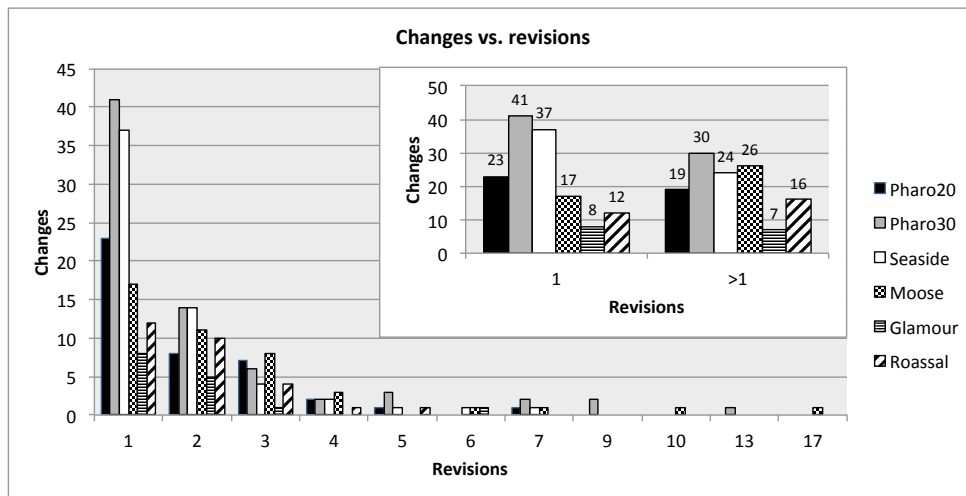
Figure 5.7: Revisions and changes in two levels of granularity. Outer chart: revisions are shown individually. Inner chart: revisions are grouped in 1 or >1.

Figure 5.8 details the data shown in Figure 5.7 with a box plot[10] of the changes distribution over the revisions. In the box plot, we identify the Pharo3 change over 13 revisions and the Moose change over 17 revisions depicted as the dot at the top (outliers are shown as dots in a box plot). The third quartile (top of the box in the box plot) is 3 for Pharo2 and Moose, and 2 for the other systems, *i.e.*, 25% of the same changes occurred in 3 or more revisions for Pharo and Moose while in 2 or more revisions for the other systems. The top whisker (which marks the highest number of changes in different revisions that is not considered an outlier) is 5 revisions for Pharo2, 6 for Moose and 3 for the other systems.

Thus, we verify that the same changes are found in different revisions, meaning that they are incrementally fixed by developers. Next, we see how our approach can alleviate this issue for the case studies themselves, and avoid the spread of changes.

Gain Analysis

A rule would support a developer finding violations when, from the moment it is created, the change proposed by the rule continues to occur in different revisions of the system. Thus, a rule could have been used to support the developers in the task of finding such violations, and avoided the spread over revisions.

---

[10]Computed with R.
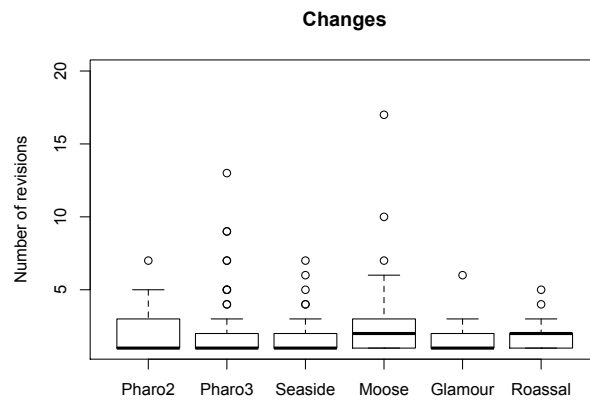
Figure 5.8: Box plot of rule fixes over revisions.

In order to investigate that, let us consider that a change is frequent enough to be considered as a rule when it occurs more than 5 times. After that, when a rule is created, it is able to detect inconsistencies in source code that should be fixed as stated by such rule.

The box plot of Figure 5.9[11] shows the distribution of (a) the number of methods for which the rules detected violations, and (b) the number of days they took to be fixed for all the case studies. The top whisker is 5 for the impacted methods and 12 days for the adaptation time. The third quartile (top of the box in the box plot) is 2 for the impacted methods, and 5.5 days for the adaptation time. In other words, 25% of the same rule violations occurred in 2 or more methods, and they took 5.5 or more days to be fixed. Thus, our approach can alleviate this problem, and avoid similar large spread in both number of impacted methods and adaptation time.

Notice that we investigated the gain that our rules could have provided in the case studies themselves. As the analyzed case studies are large, there was a relevant gain. In this context, in the case a similar analysis is performed in client systems, we expect an even more noticeable gain in terms of impacted methods and adaptation time (since, for example, client developers may be not aware or do not closely follow the frameworks and libraries evolution). Such investigation, at a large-scale ecosystem level, is covered by the next chapter.

5.6.4   Comparison with the Previous Approach

In this chapter we focus on the extraction of rules to support client systems while in the previous chapter the goal is to detect system-specific conven-

---

[11]For legibility, we filter outliers.
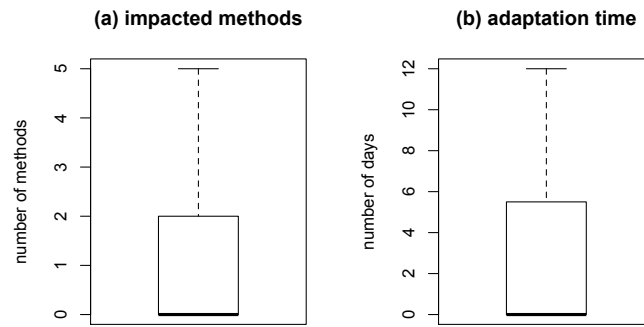
Figure 5.9: Box plots for (a) impacted methods and (b) adaptation time of rule violations.

tions.

On the one hand, system-specific conventions are produced because we only take into account rules that occur in two or more revisions (*cf.* Subsection 4.2.3). In addition, another characteristic that favors the creation of system-specific conventions is the presence of the argument value for primitive types in the deltas (*cf.* Subsection 4.2.2).

On the other hand, we support clients because we consider all rules extracted from frameworks/libraries as proposed by previous studies in this research area [DR08, SJM08, WGAK10, MWZM12], independently of the number of revisions the rules occur.

Notice that, by definition, the approach presented in this chapter is more flexible (because it relies on data-mining) than the one presented in the previous chapter (because it is based on patterns). Thus, the approach presented here is able to produce the rules of the previous chapter, at the cost of producing more noise, since it neither filters rules over revisions nor uses the argument value for primitive types in the deltas. A comparison experiment between both approaches with respect to the rules they generate remains future work.

5.7 Threats to Validity

5.7.1 Construct Validity

The construct validity is related to whether the measurement in the study reflects real-world situations. In our study, the main threat is the validation of the generated rules.

As an error in this process would bias the results, the rules were mostly

validated by experts of the systems under analysis, which is hard to have for real-world systems, ensuring more confidence to our results.

In two cases (*i.e.,* the rules generated for Seaside and the ones only produced by the classical approach in Subsection 5.6.2), the rules were manually validated by the authors of the study with the support of documentation and code examples. Previous related studies (*e.g.,* [SJM08, MWZM12, WGAK10]) do *not* adopt the validation with experts, *i.e.,* the authors of the studies validate the rules themselves.

5.7.2   Internal Validity

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

Our tool has been (i) used by several members of our laboratory to support their own problems with frameworks evolution, and (ii) divulged in an open-source software reengineering mailing list[12] such that developers of this community can use it. Thus, we believe that these tasks reduce risks of this threat.

5.7.3   External Validity

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

We studied six case studies of different size. They are credible case studies as they are open-source, real-world and non-trivial systems with relevant number of developers. They also come from different domains and include a large number of revisions and deltas. Despite this observation, our findings — as usual in empirical software engineering — cannot be directly generalized to other systems, specifically implemented in other languages. Moreover, closed-source systems, due to differences in the internal processes, might have different properties in their commits.

5.8   Summary

In this study, we proposed a novel approach to generate evolution rules by monitoring changes applied in source code during the framework or library evolution. Rules are mined from source code history taking into account the changes between revisions, and they follow the types *one-to-one*, *one-to-many*, *many-to-one* and *many-to-many*. Furthermore, our approach provides either on request rules or automatically generated rules, and our tool is able

---

[12]http://www.moosetechnology.org

to show code examples of the rules that clarify how the framework/library adapted to its own changes.

Our approach was evaluated on five open-source systems, and the rules were assessed with the help of experts with respect to their validity. The results showed that valid rules were extracted from source code history and many were related to method suggestion. Moreover, we provided two complementary discussions around the effort to select correct rules and the gain provided by them. We reiterate here the most interesting conclusions we derived from our study:

1. In the evaluation by the experts the precision remained between 46% and 86%. For two case studies, the experts were not confident to evaluate some rules (thus, this decreased the precision). This suggests the importance of the validation by experts: some rules are hard to be evaluated and involve very specific knowledge. For the cases were the experts were confident the precision remained between 65% and 86%.

2. Our approach is the first to produce rules *one-to-one*, *one-to-many*, *many-to-one* and *many-to-many* involving method replacement and suggestion, thus, improving the spectrum of rules generated by current studies.

3. Using the classical approach to discover rules in changes of the entire system was not as effective as using our approach. Our study was able to produce more valid rules and a better precision, thus, reducing the effort to find valid ones.

4. The same changes were found in different revisions, meaning that in many cases they are incrementally fixed by developers. Our approach can alleviate similar issues by describing changes as rules, thus, avoiding the spread of the changes over time.

The validation presented in this chapter was performed in the systems under analysis themselves, such as existing work in this research area (*e.g.*, [DR08, SJM08, WGAK10, MWZM12]). In practice, a software system is frequently part of a bigger software ecosystem [Lun09]. The analysis of software evolution and its impact at large-scale level can help developers to better understand its real extension and what can be done to alleviate such impact. In the next chapter we verify the impact of the rules extracted with the proposed approach in a large-scale software ecosystem.

IMPACT OF SOFTWARE EVOLUTION ON ECOSYSTEMS

Contents

6.1 Introduction

As frameworks evolve, client systems often need to adapt their source code to use the updated API. To facilitate this time-consuming task, approaches have been developed to help client developers. This can be done, for example, with the support of specialized IDEs [HD05], the help of framework developers [CN96], or with evolution rules [DR08,SJM08,WGAK10,MWZM12, HEA^{+}14], as described in the previous chapter.

Commonly, these approaches are evaluated on small case studies such as individual systems. In practice, a software system is frequently part of a bigger software ecosystem, which often exists in large companies, organizations, or open-source communities [Lun09]. Software ecosystems consist of multiple systems, often interrelated between each other. In this context, it is hard to predict how systems are used by their clients. For example, developers of a large corporation pointed to us that sometimes changes in the API of their core systems would break other systems that they were not expecting. In another case, the evolution of an API affected thousand of clients, but only a minority of these clients were aware and updated their source code [RLR12].

These scenarios suggest that the impact of API evolution on large-scale software ecosystems may be large and sometimes unknown; managing API evolution is, in this context, a complex and risky task [BCP+13].

To support developers to better understand the real impact of API evolution and how it could be alleviated, researchers should also support developers working in software ecosystems. In that respect, a large-scale study was performed by Robbes *et al.* [RLR12] to verify the impact of deprecated APIs on a software ecosystem.

Naturally, API evolution is not restricted to method deprecation. It may state the use of a better API to improve, for example, code legibility, portability, performance, etc., or any other replacement not necessarily involving API deprecation, as we have seen in the previous chapter. But are client developers aware about such API replacements? How often and how broad is the impact on clients? While studies (*e.g.*, [WGAK10,DJ05]) state that some framework changes may have consequences (*e.g.*, compile errors) on clients, the real extension is not known. The awareness of clients with respect to new/better APIs needs therefore to be investigated; an ecosystem analysis allows us to verify on the actual clients.

In this chapter, we analyze the impact of API evolution, not related to API deprecation, of frameworks and libraries on their actual client systems [HRA+15]. We set out to discover (i) whether API changes cause propagation in their client systems, and (ii) whether the clients are aware about these API changes. Our goal is to better understand, at the ecosystem level, to which extent client developers are affected by the evolution of frameworks and libraries, and to reason about how it could be alleviated. Thus, we investigate the following research questions to support our study:

- **Frequency**. *RQ1*: How often do API changes cause propagation in the ecosystem?

- **Magnitude**. *RQ2*: How many systems react to the API changes in the ecosystem and how many developers are involved?

- **Duration**. *RQ3*: How long does it take for systems to react and adapt to the API changes?

- **Extension**. *RQ4*: Do all the systems in the ecosystem react to the API changes?

- **Consistency**. *RQ5*: Do the systems react to an API change in the same way? *RQ6*: How followed are the API changes by the ecosystem?

In this study we cover the *Pharo software ecosystem*, which is composed by systems implemented with the Pharo programming language. It has about 3,600 distinct systems, more than 2,800 contributors, and six years of evolution. We analyzed 118 important API changes and we detected that: (i) more

than half of the API changes impacted client systems; (ii) many API changes impacted the ecosystem in terms of systems, packages, classes, methods, and developers; (iii) client developers need some time to discover and apply the new API; (iv) the number of affected systems were much higher than those that actually react to API changes; (v) replacements were not resolved in an uniform manner in the ecosystem.

This work extends the study of Robbes *et al.* [RLR12] that focused on the analysis of methods marked as deprecated. Thus, this will allow us to compare our results with the ones provided by that study to better understand how two distinct types of API evolution affect their actual clients. We detected, for example, that the API changes analyzed in our study take more time to be adopted by clients, and that clients are less aware about them.

The main contributions of this chapter can be summarized as follows:

1. We provide a large-scale study, at the ecosystem level, to better understand to which extent client developers are impacted by API changes of frameworks and libraries.

2. We provide a comparison between our results and the ones of a previous work on API deprecation.

Structure of the Chapter

We introduce our ecosystem in Section 6.2. We describe our methodology in Section 6.3. We present and discuss the results of the experiments in Sections 6.4 to 6.9. We present the implications of our study in Section 6.10. Finally, we discuss the threats to the validity of our experiments in Section 6.11, and we conclude the chapter in Section 6.12.

6.2 The Pharo Ecosystem

6.2.1 Choice of the Pharo Ecosystem, and Alternatives

We need to select an adequate ecosystem that is relevant and provide support for answering our research questions. This step is critical, since determining which system is part of the ecosystem can be a challenge, if there is no clear inclusion criterion. For instance, some communities may be spread over multiple websites that need to be individually analyzed [RLR12]; the initial step of gathering the list of projects being part of the ecosystem in that case may be difficult in itself – independently of the subsequent task of gathering the actual data.

For this study, we select the ecosystem built around the Pharo open-

source development community. It is hosted by the SqueakSource[1] and by the SmalltalkHub[2] source code repositories, which store a large number of distinct systems. Our analysis included six years of evolution (from 2008 to 2013) with 3,588 systems and 2,874 contributors. There are two factors influencing this choice. First, the ecosystem is concentrated on two repositories, SqueakSource and SmalltalkHub, which gives us a clear inclusion criterion. Second, we are interested in comparing our results with the work of [RLR12]; using the same ecosystem facilitates this comparison.

As an alternative to our choice of ecosystem, we could have selected a development community based on a more popular language such as Java or C++. However, this would have presented several disadvantages. First, deciding which systems to include or exclude would have been much more challenging. Second, the potentially very large size of the ecosystem could prove impractical. We consider the size of the Pharo ecosystem as a "sweet spot": with about 3,600 distinct systems and more than 2,800 contributors, it is large enough to be relevant, but does not necesitate a large amount of processing power as the Java Maven ecosystem would [RvDV13].

6.2.2   Pharo, and the Pharo Ecosystem

Pharo is an open-source, Smalltalk-inspired language and environment. Pharo includes the implementation of all features inherent to an object-oriented language (collections, exceptions, primitive types, etc.). It provides a distribution of Smalltalk with a large set of frameworks and libraries as well as an IDE and several tools.

Pharo is currently used in many industrial and research projects[3]. For example, the Seaside web-development framework[4], a competitor for Ruby on Rails as the framework of choice for rapid web prototyping, is developed by the Pharo community. Moose, an open-source platform for software and data analysis, is also based on Pharo; this platform is currently supported by several research groups around the world and adopted in industrial projects[5]. Phratch[6], a visual and educational programming language, is a port of Scratch to the Pharo platform. Many other projects are developed in Pharo and hosted in SqueakSource or SmalltalkHub.

---

[1]http://www.squeaksource.com
[2]http://www.smalltalkhub.com
[3]http://consortium.pharo.org
[4]http://www.seaside.st
[5]http://www.moosetechnology.org/docs/publications
[6]http://www.phratch.com

6.2.3 The SqueakSource and SmalltalkHub Large-scale Repositories

SqueakSource and SmalltalkHub repositories are the basis for the software ecosystem that the Pharo community have built over the years. They are the *de facto* platform for sharing open-source code for this community. Therefore, the large majority of Pharo developers use SqueakSource or SmalltalkHub as their source code repository. This means that these repositories offer a nearly complete view of the Pharo software ecosystem.

The SqueakSource repository is also partially used by the Squeak[7] open-source development community. SmalltalkHub was created after SqueakSource by the Pharo community to be a more scalable and stable repository. As a consequence, many Pharo projects migrated from Squeak-Source to SmalltalkHub, and nowadays, new Pharo projects are concentrated in SmalltalkHub. We address this transition between the repositories in our experiments.

6.3 Methodology

In this section we present the methodology to generate and validate the list of API changes used in our study.

6.3.1 Selecting frameworks and libraries

The frameworks and libraries from which we extract the API changes come from the Pharo language itself. They provide the set of APIs that come with Pharo by default, including collections, files, sockets, unit tests, streams, exceptions, graphical interfaces, etc.; they are Pharo's equivalent to Java's SDK.

We took into account all the versions of Pharo since its initial release, *i.e.*, versions 1.0, 1.4, 2.0, and 3.0. Table 6.1 shows the number of classes and lines of code in each version. The major development effort between versions 1.0 and 1.4 was focused on removing outdated code that came from Squeak, the Smalltalk dialect Pharo is a fork of, explaining the drop in number of classes and lines of code.

Table 6.1: Pharo versions size.

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

---

[7]http://www.squeak.org

6.3.2   Generating a list of API changes

We adopted our previous approach [HEA⁺14], described in Section **??**, to generate a list of API changes. We set out to produce rules with a minimum support of 5, and a minimum confidence of 50%. The minimum support at 5 states that a rule has a relevant amount of occurrences in the framework or library, and the minimum confidence at 50% yields a good level of confidence (as an example, [SJM08] use a confidence of 33% in their approach to detect evolution rules). Moreover, the thresholds reduce the number of rules to be manually analyzed.

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

6.3.3   Filtering the list of API changes by removing API deprecation

Naturally, some of the API changes generated by our approach are related to API deprecation. As in this work we do not take into account API changes related to API deprecation, they need to be removed from our list of rules. To do that, we extracted all methods marked as deprecated found in the analyzed evolution of Pharo; this produced 1,015 API deprecation. We detected that from our list of 148 API changes, 30 were about API deprecation. For example, the API change FileReference.asReference() → FileReference.asFileReference() is about API deprecation because FileReference.asReference() was marked as deprecated, so this rule was discarded. By discarding the API changes related to API deprecation, *our final list includes 118 API changes, which are adopted in this work*.

From these API changes, 59 are about method suggestion (*i.e.*, both methods are available to be used by the client; *cf.* Subsection **??**) and 59 are about method replacement (*i.e.*, the old method is removed, so it is not available to be used by the client).

Furthermore, 10 out of the 118 API changes involved the evolution of internal APIs of the frameworks and libraries which, in theory, should not affect client systems. By internal API, we mean a public component that should only be used internally by the framework or library, *i.e.*, not by client systems. For instance, in Eclipse, the packages named with *internal* include public classes that is not part of the API provided to the clients [DJ05].

Table 6.2 resumes the types of API changes. In our experiments, we assess

the use of API changes about method replacement and suggestion as well as about internal APIs.

Table 6.2: Types of API changes.

Total API changes	118		
Replacement	59	**Internal**	10
Suggestion	59	**Public**	108

In Table 6.3, we present some examples of the API changes. The first API change improves code legibility, as it replaces two method calls by a single, clearer one. The second example replaces a method with a more robust one, that allows one to provide a different behavior when the intersection is empty. The third is a usage convention: Pharo advises not to use Object.log() methods, to avoid problems with the *log* function. Finally, the fourth one represents a class and method replacement due to a large refactoring: ClassOrganizer.default() does not exist anymore; ideally, it should have been marked as deprecated.

Table 6.3: Example of API changes.

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

6.3.4   Assessing reactions of API changes in the ecosystem

When analyzing the reaction of the API changes in the ecosystem, we do not consider the frameworks and libraries from which we discovered the API changes.

To detect a reaction to API change in client systems, we analyze every source code commit in which the same method removes method calls to the old API and adds method calls to the new API, according to one of the rules selected above. For example, Figure 6.1 shows an example of a client (Seaside) commit reacting to the API change ProtoObject.isNil() *and* Boolean.ifTrue(*) $\rightarrow$ ProtoObject.ifNil(*), *i.e.*, the method calls ProtoObject.isNil() and Boolean.ifTrue(*) are removed while the method call

ProtoObject.ifNil(*) is added.

> Diff of method RRRssRoot.styles() between two revisions:
> − if (styles.isNil().ifTrue()) {...
> + if (styles.ifNil()) {...

Figure 6.1: Example of a client (Seaside) reaction to the API change ProtoObject.isNil() *and* Boolean.ifTrue(*) → ProtoObject.ifNil(*).

6.3.5 Addressing the transition between SqueakSource and SmalltalkHub

As stated in the previous section, there was a transition of some projects from SqueakSource to SmalltalkHub. We detected that 211 projects migrated from SqueakSource to SmalltalkHub while keeping the same name and copying the full source code history. We count these projects only once: we only kept the projects hosted in SmalltalkHub, which hosts the version under development and also the full source code history.

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

6.3.6 Presenting the results

Sections 6.4 to 6.9 present the results of our experiments. Each section is organized in three parts:

1. *General results.* It answers the research questions and discusses the results.

2. *Time-based results.* It analyzes the most important results taking into account the age of the API changes, where *earlier API changes* are the older ones and *later API changes* are the recent ones when they are sorted by their creation date. This is done because we want to verify whether the age of an API change influences the analysis. For such comparison, we use the Mann-Whitney test for assessing whether one of two samples (earlier and later API changes, in our case) of independent observations tends to have larger values than the other. This test is used when the distribution of the data is not normal and there is different participants (not matched) in each sample. As is customary, the tests will be performed at the 5% significance level. We also report *effect size* which indicates the magnitude of the effect and is independent of sample size. Effect size value 0.1, 0.3 and 0.5 are considered small, medium and large effects, respectively.

3. *Comparison with API deprecation.* It compares our results with the ones provided by [RLR12] on API deprecation. This is done in order to better characterize the phenomenon of change propagation at the ecosystem level. The comparison is possible because, in most of the cases, the research questions are equivalent for both studies.

Results will be presented, when it is possible, using box plots in conformance with the previous study on API deprecation [RLR12].

6.3.7 Experiment replication

The empirical study presented in this chapter is a replication of the study provided by Robbes *et al.* [RLR12]. In our study, we perform an internal replication [SSS08], which is undertaken by one of the original experimenters, *i.e.*, we repeated our own experiments. This study can also be classified as a close replication [SSS08].

Regarding the method of analysis, our study analyzed rules coming from frameworks and libraries while the original study analyzed rules coming from deprecated methods.

Due to such fact, there is also a difference on the task of our replication. We assess commits in the ecosystem that applied the prescribed API change (*i.e.*, the removals and additions of method call according to the rule we inferred). In the original study, the authors were primarily interested in the removals of the deprecated methods, but did not specify their replacement. Thus, by definition, we are more rigorous when assessing a reaction to an API change. This comes at the cost of not considering alternative reactions to the API change, *i.e.*, reactions different of the ones extracted from the analyzed frameworks and libraries.

Finally, there is also a small difference on the subject under analysis: we studied a larger ecosystem (SmalltalkHub and SqueakSource) while the original work studied only SqueakSource.

6.4   Frequency of Change Propagation

Next, we present the results of our first research question about the frequency of change propagation.

RQ1. How often do API changes cause propagation in the ecosystem?

6.4.1   General results

From the 118 API changes, 62 (53%) caused reactions that impacted at least one system in the ecosystem while 56 (47%) did not cause any reaction. This means that when doing any change on such APIs, the developers should be aware that there is roughly 50% chances that this change will impact the clients.

Moreover, from the API changes that caused reactions, 5 are internal, meaning client developers also use internal parts of frameworks and libraries to access functionalities not available in the public interfaces [DR08, BR06].

The other API changes that did not cause reactions could only be used internally by the frameworks and libraries; alternatively, the clients could still be unaware of the API change. In fact, we see in the next research questions that many systems take time to react to API changes. Hence, some API changes may not have been applied by all systems yet.

6.4.2   Time-based results

The proportion of reactions in the ecosystem may be also influenced by the age of the API changes. We may expect recent API changes to cause less reactions than older ones. In that respect, we verify whether there is a higher proportion of earlier API changes that cause propagation.

Considering the 1/3 earlier API changes (*i.e.*, 39 out of 118), 21 (54%) caused reactions while 18 (46%) did not cause any reaction. In contrast, considering the 1/3 later API changes, 16 (40%) caused reactions while 24 (60%) did not cause any reaction. This result confirms that earlier API changes caused more reactions than later ones, thus, clients that did not react may do later.

6.4.3   Comparison with API deprecation

In the API deprecation study, 12% (93 out of 762) of the deprecated entities caused reactions in the ecosystem. We cannot make a direct comparison with

our results because such study considers all possible API deprecation while in our study we do not analyze all possible API changes (*i.e.*, we set out a threshold and analyze the most important ones).

However, both studies demonstrate that there are reactions at the ecosystem level. The low reaction in the API deprecation work suggests that framework developers follow defensive practices by deprecating even entities used only internally and that clients are unaware [RLR12], similarly to what happens in our study. Furthermore, even if both results are not directly comparable, one could try to investigate these two results to confirm whether there is such a difference.

6.5 Magnitude of Change Propagation

In this section we present the results of our second research question about the magnitude of change propagation.
RQ2. How many systems react to the API changes in the ecosystem and how many developers are involved?

6.5.1 General results

In this research question we quantify the reactions to the API changes in number of systems, packages, classes, methods and developers.

To determine the magnitude of a change propagation, we need to detect when the change was available to be used by client systems. We consider that an API change is available from the first moment it was used internally in the framework or library. *All commits after this moment that remove a method call to the old API and add a method call to the new API are considered to be reactions to the API change.*

Table 6.4: Magnitude of change propagation.

	Systems	Packages	Classes	Methods	Developers
Total	178	252	503	796	134
Avg.	4.9	5.6	9	13.4	4.4

As shown in Table 6.4, 178 client systems reacted to API changes and on average 4.9 systems reacted to a change causing propagation; these changes involved 134 distinct developers (*i.e.,* distinct commit authors). We analyze the distribution of such data (*i.e.,* the API changes that caused change propagation) in the box plots shown in Figures 6.2 and 6.3. Moreover, in research question 4, we discuss the relative distribution of such data.

Reacting systems and packages

Figure 6.2a shows the distribution of reacting systems: the 1st quartile is 1 (bottom of the box), the median is 2 (middle of the box), the 3rd quartile is 5 (*i.e.*, 25% of the API changes cause reactions in 5 or more systems, forming the top of the box in the box plot), and the maximum[8] is 11 (*i.e.*, it marks the highest number of reacting systems that is not considered an outlier, forming the top whisker of the box). The API change isNil().ifTrue(*) → ifNil(*), for example, caused the largest reaction, 41 systems reacted; this change is depicted as the dot at the top of the box plot in Figure 6.2a (in a box plot all outliers are shown as dots).

For packages (Figure 6.2b), the 1st quartile is 1, the median is 2, the 3rd quartile is 7, and the maximum is 16.
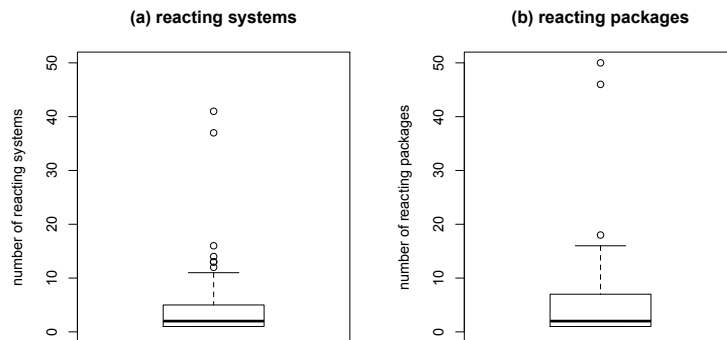
Figure 6.2: Box plots for (a) systems and (b) packages reacting to API changes.

Reacting classes and methods

The figures at the level of classes and methods also follow similar distributions. For classes (Figure 6.3a), the 1st quartile is 1, the median is 3, the 3rd quartile is 13, and the maximum is 23. Finally, for methods (Figure 6.3b), the 1st quartile is 2, the median is 6, the 3rd quartile is 17, and the maximum is 39.

These results show that some systems reacted several times to the same API change: the median system reaction is 2 while the median method reaction is 6. Overall, the impact is localized per system (few classes reacted for each system) whereas it is spread over several systems. For example, the API change isNil().ifTrue(*) → ifNil(*) caused reaction in 41 systems, 46 packages, 69 classes, and 89 methods.

---

[8]We adhere to the naming used in [RLR12].

Figure 6.3: Box plots for (a) classes and (b) methods reacting to API changes, and (c) affected developers.

Reacting developers

The number of developers involved by the change propagation is shown in Figure 6.3c, as the number of commit authors that react to the API changes. In this case, the 1st quartile is 1, the median is 2, the 3rd quartile is 5, and the maximum is 11. The median at 2 shows that many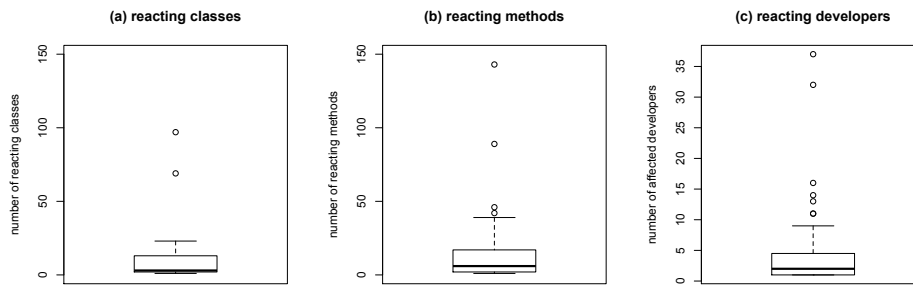 change propagations involve few developers while the 3rd quartile at 5 shows that some of them involve several developers. The API change isNil().ifTrue(*) $\rightarrow$ ifNil(*), for example, involved a large number of developers: 37.

Overall, the distribution of developers involved in the change propagation is similar to the number of systems, implying that it is common that only one developer involved in the same system reacts to the API changes.

6.5.2 Time-based results

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

Figure 6.4 shows the reacting systems with the API changes that caused change propagation separated in two groups: earlier changes and later changes. For the earlier changes, the median is 2, the 3rd quartile is 6.5, and the maximum is 13 whereas for the later changes, the median is 2, the 3rd quartile is 4, and the maximum is 8. Comparing both earlier and later give a *p-value* > 0.05 and effect size = 0.01. While the median is equal for both groups of changes, the 3rd quartile and maximum show that earlier API changes have more reactions. Consequently, reactions to more recent API changes may be yet to come; we investigate this issue in the next research question.
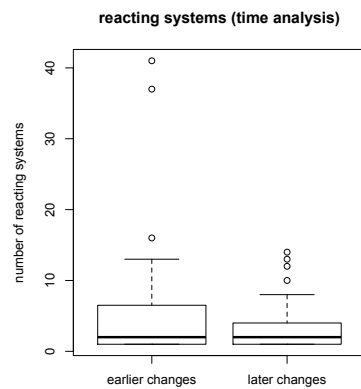
Figure 6.4: Box plots for the reacting systems separated by earlier and later API changes.

6.5.3   Comparison with API deprecation

Our result is different when we compare to API deprecation. In our study there are 62 API changes reacting while in the API deprecation case there are 93 deprecated entities reacting, *i.e.*, 50% more. However, the median of reactions in our study in 2, whereas it is 5 to the API deprecation, meaning 150% more reactions. It means that there are more reactions to the API deprecation. As expected, this is facilitated since deprecated methods produce warnings to developers with recommendation replacements while in our API changes no warning is produced; the problem is only fixed by a developer aware about the API change.

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

6.6   Duration of Change Propagation

Next, we present the results of our third research question about the duration of change propagation.

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

6.6.1 General results

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

Reaction time

We calculate the reaction time to an API change as the number of days between the starting time and the first reaction in the ecosystem to the API change. As shown in Figure 6.5a, the minimum is 0, the 1st quartile is 5, the median is 34, the 3rd quartile is 110, and the maximum is 255. The 1st quartile at 5 days shows that some API changes see a reaction in only some days: this is possible if developers, work both on frameworks and on client systems, or coordinate API evolution via mailing lists[9], as suggested by [?]. In contrast, the median at about 34 days and the 3rd quartile at 110 days indicate that some API changes take a long time to be applied. In fact, as Pharo is a dynamically typed language, some API changes will only appear for developers at runtime, not in compile time, which can explain the long timeframe.

Figure 6.5: Box plots for reaction time of (a) all API changes and (b) separated by method suggestion and replacement, both in number of days.

In addition, we analyze the reaction time considering the two categories of API changes, method suggestion and replacement, as shown in Figure 6.5b. For the API changes about method suggestion, the 1st quartile is 10, the median is 47, the 3rd quartile is 255, and the maximum is 351. In

---

[9]Examples of API change coordination via mailing lists: `http://goo.gl/50q2yZ`, `http://goo.gl/SkMORX`, `http://goo.gl/ZgvsVF`.

contrast, for the API changes about method replacement, the 1st quartile is
1, the median is 20, the 3rd quartile is 59, and the maximum is 110.

We notice that the reaction time for the API changes about method sug-
gestion is longer than the ones about replacement, implying that the former
is harder to be detected by client developers. This is explained by the fact
that in the API changes about method suggestion, even if the new method
uses a better API, the old method is still valid, so client developers are not
forced to update their code. Ideally, client developers would benefit if such
API changes were presented to them beforehand. In practice, many develop-
ers are simply not aware about them or see no reason to change their running
code.

Adaptation time

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

We detected that 86% of the adaptations occur in 0 days, indicating that
the majority of the systems fix an API change extremely quickly. This may
occur with the help of refactoring tools found in current IDEs. Figure 6.6a
shows the distribution for the other 14% of adaptations: the 1st quartile is 12
days, the median is 71, the 3rd quartile is 284, and the maximum is 662 (we
filter outliers for legibility). The median at 71 days shows that 50% of these
systems take more than two months to adapt to certain API changes; some
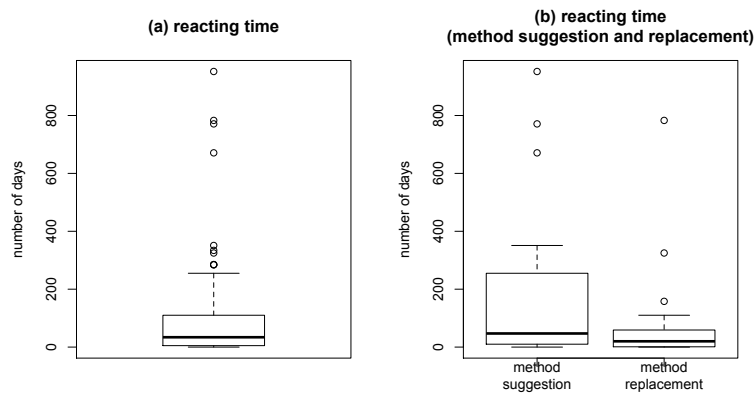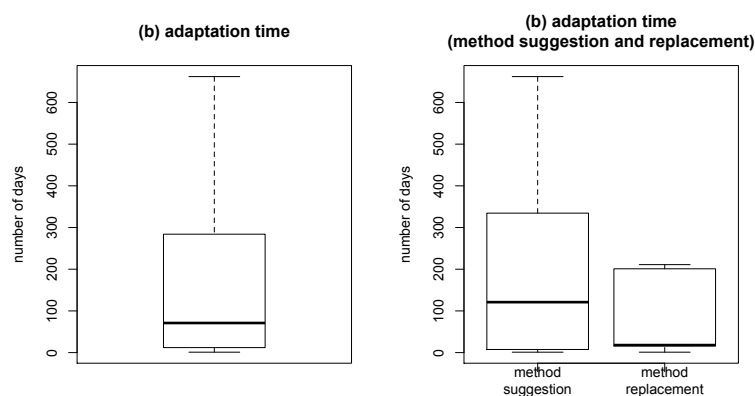systems may even take years.

Figure 6.6: Box plots for adaptation time of (a) all API changes and (b) sepa-
rated by method suggestion and replacement, both in number of days.

In Figure 6.6b we analyze the adaptation time considering the two cate-

gories of API changes. For the API changes about method suggestion, the 1st quartile is 7.5, the median is 121, the 3rd quartile is 334.5, and the maximum is 662. For the API changes about method replacement, the 1st quartile is 16, the median is 18, the 3rd quartile is 201, and the maximum is 211. Again, similarly to the reaction time, the adaptation time for the API changes about method suggestion is longer than the ones about replacement. This suggests that the former takes more time to be adopted in the same system by client developers.

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

6.6.2 Time-based results

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

Figure 6.7 shows the adaptation time (for the 14% of the adaptations that occur in more than 0 days) separated in the groups earlier and later API changes. For the earlier changes, the 1st quartile is 32, the median is 284, and the 3rd quartile is 454. For the later changes, the 1st quartile is 5, the median is 18, and the 3rd quartile is 133. Comparing both earlier and later give a *p-value* < 0.01 and effect size $= 0.39$. Thus, we confirm that earlier API changes have a longer adaptation time, and that the variable time also plays an important role in the adaptation of a system.
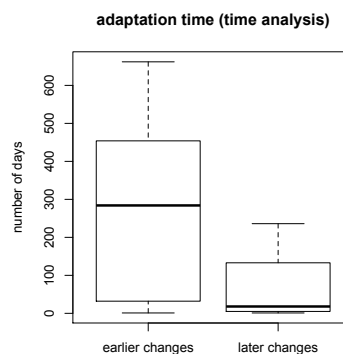

Figure 6.7: Box plots for the adaptation time separated by earlier and later API changes.

6.6.3 Comparison with API deprecation

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

The adaptation time of the API changes reported in our study is comparable to the adaptation time of API deprecation in which most of the systems adapt quickly while some systems may take a long time. For both studies, the same explanation can be considered. First, some of the largest systems on SqueakSource and SmalltalkHub are software distributions, where the large size of the system and the amount of developers make it comparable to a small-scale ecosystem. Second, due to the fact that Pharo is a dynamically typed language, some large and even small systems may miss to update the API, and keep in such state for a long time.

6.7 Extension of Change Propagation

In this section we present the results of our fourth research question about the extension of change propagation.
RQ4. Do all the systems in the ecosystem react to the API changes?

6.7.1 General results

In the previous subsection we have seen that some systems take a long time to react to an API change. We see in this subsection that other systems do not react at all. *To determine whether all systems react to the API changes, we investigate all the systems that are potentially affected by them, i.e., that feature calls to the old API.*

Table 6.5 shows that 2,188 systems are potentially affected by the API changes and on average 127 systems are potentially affected by a API change.

Moreover, we detected that 112 API changes (from the 118 analyzed API changes), including the 10 internal API changes, potentially affected systems in the ecosystem. In the rest of this subsection, we analyze the distribution of such data.

Affected systems and packages

Figures 6.8 (a) and (b) show the distribution of systems and packages affected by API changes in the ecosystem. We note that the number of affected systems

Table 6.5: Extension of change propagation.

	Systems	Packages	Classes	Methods	Developers
Total	2,188	6,322	45,186	107,549	1,579
Avg.	127	220.7	582.5	1,020.6	100.5

and packages are much higher than those that actually react to API changes (as shown in Figure 6.2). The 1st quartile of affected systems is 15 compared to only 1 system reacting (packages: 19.5 compared to 1). The median of affected systems by an API change is 56.5 compared to only 2 systems reacting to it (packages: 82 compared to 2). The 3rd quartile of affected systems is 154.5 compared to 5 systems reacting (packages: 244.5 compared to 7).
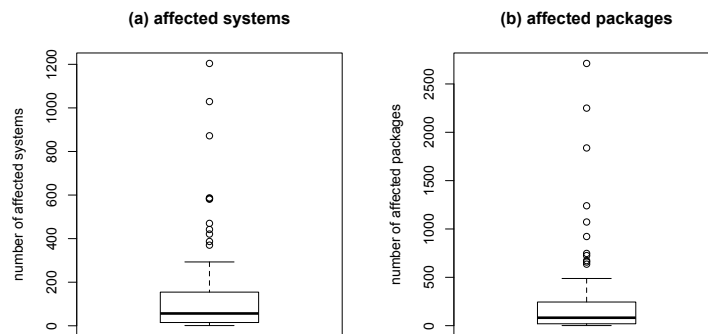
Figure 6.8: Box plots for (a) systems and (b) packages affected by API changes.

Affected classes, methods and developers

We also analyze the distribution of affected classes, methods as well as developers. For classes (Figure 6.9a), the 1st quartile is 29.5, the median is 176, the 3rd quartile is 473, and the maximum is 1,029 (we filter outliers for legibility). For methods (Figure 6.9b), the 1st quartile is 59, the median is 253, the 3rd quartile is 744.5, and the maximum is 1,650 (we filter outliers for legibility). For the developers (Figure 6.9c), the 1st quartile is 13.5, the median is 49, the 3rd quartile is 125.5, and the maximum is 291. Note again that the number of affected classes, methods and developers as presented in Figure 6.9 are much higher than those that actually react to API changes (as shown in Figure 6.3). In order to better understand this difference, next we analyze the reactions in the context of the affected systems.
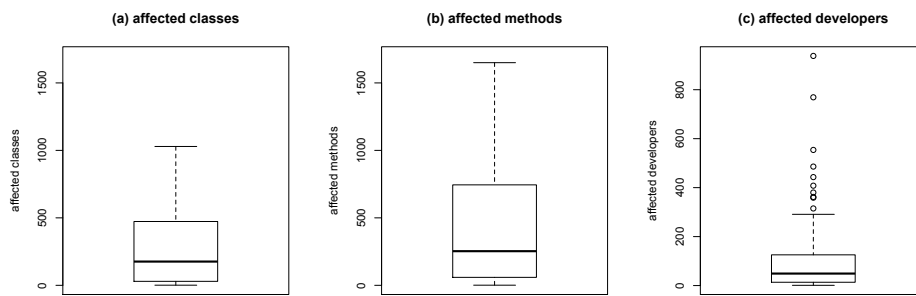
Figure 6.9: Box plots for affected (a) classes and (b) methods, and (c) developers.

Relative analysis

The relative analysis of reacting and affected systems produce a better overview of the impact. In that respect, comparing the ratio of reacting systems to the ratio of affected systems gives the distribution shown in Figure 6.10a. It shows that a very low number of systems react: the median is 0%, the 3rd quartile is 3%, the maximum is 7%. We investigate possible reasons for this low amount of reactions.

Figure 6.10: Box plots for ratios of: (a) reacting affected systems; (b) reacting alive systems; and (c) reacting alive systems, removing counter reactions.

In a software ecosystem, a possibly large amount of the systems may be stagnant, or even dead systems [RLR12]. Thus, we first investigate the hypothesis in which systems that did not react either died before the change propagation started or were stagnant. A system is dead if there are no commits to its repository after the API change that triggered the change propagation. A system is stagnant if a minimal number of commits (less than 10) have been performed after the API change. Thus, removing dead or stagnant systems (*i.e.*, keeping alive systems only) produces the distribution shown in Figure 6.10b: 1st quartile is 0%, the median is 2.5%, the 3rd quartile is 12%,

and the maximum is 27%.

A second reason why a system would not react to a change is when it is using another version of the library or framework changing its API, one in which the API did not change. This may occur when a system does not have the manpower to keep up-to-date with the evolution and freezes its relationship with a version that is sufficient for it [RLR12]. To estimate this effect, we measure the number of systems that actually add more calls to the old API change, *i.e.*, they are counter reacting to the API evolution. Thus, removing these systems from the alive ones gives the distribution shown in Figure 6.10c: the 1st quartile is 0%, the median is 9%, the 3rd quartile is 44%, and the maximum is 100%.

To further detect inactive systems we analyze each repository, Squeak-Source and SmalltalkHub, separately. Since SmalltalkHub repository was created by the Pharo community, we expect that developers are more active in such repository in the recent years. Thus, in Figure 6.11 we perform, for each repository, the three ratio comparisons shown in Figure 6.10. Firstly, in the ratio of reacting and affected systems, shown in Figure 6.11a, the median is 0%/0% (for SqueakSource/SmalltalkHub, respectively), the 3rd quartile is 1%/7%, and the maximum is 2%/16%. Secondly, in the ratio of reacting and alive systems, shown in Figure 6.11b, the median is 0%/0%, the 3rd quartile is 7%/22%, and the maximum is 16%/50%. Finally, in the ratio of reacting and alive without counter reacting systems, shown in Figure 6.11c, the median is 0%/0%, the 3rd quartile is 22%/50%, and the maximum is 55%/100%. These results show that the community in SmalltalkHub is in fact more active, so reactions are more common in such repository, but, relatively, they are still low.

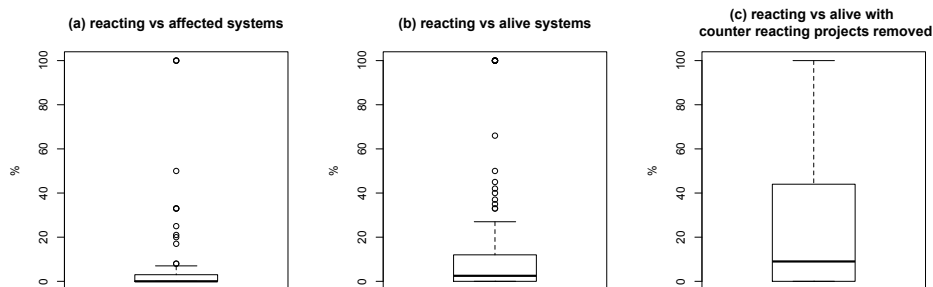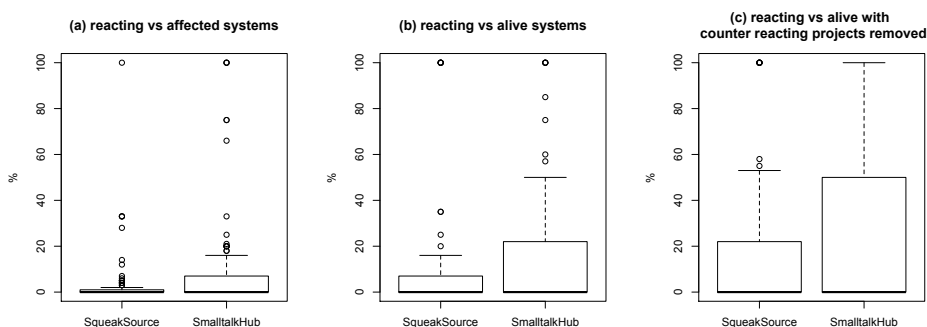Figure 6.11: Box plots, separated by repository, for ratios of: (a) reacting affected systems; (b) reacting alive systems; and (c) reacting alive systems, removing counter reactions.

6.7.2   Time-based results

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

Figure 6.12 shows the number of affected systems separated in the groups earlier and later API changes. For the earlier changes, the 1st quartile is 18.5, the median is 55.5, and the 3rd quartile is 212 while for the later changes, the 1st quartile is 5.5, the median is 59, and the 3rd quartile is 130. Comparing both earlier and later give a *p-value* $= 0.14$ and effect size $= 0.13$. Earlier API changes affect slightly more systems. Even if the difference between both groups is small, the number of potentially affected client systems by the API changes tends to increase over time. Consequently, as time passes, it will be more complicated for these clients to migrate to new/better APIs.
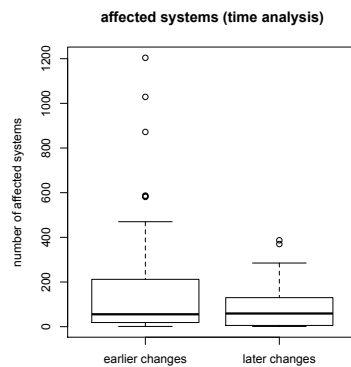
Figure 6.12: Box plots for the affected systems separated by earlier and later API changes.

6.7.3   Comparison with API deprecation

The presented ratio comparison is very different when compared to the API deprecation study. For the *ratio of reacting and affected systems*, the 1st quartile is 13%, the median is 20%, and the 3rd quartile is 31% in the API deprecation case (compared to 0%, 3% and 7%, respectively, in our API changes), which confirms the difference between both types of API evolution. These percentages increase in the other ratio comparisons. For the *ratio of reacting and alive without counter reacting systems*, the 1st quartile is 50%, the median is 66%, and the 3rd quartile is 75% for API deprecation (compared to 0%, 9% and 44%, respectively, in our API changes). Overall, even if the majority of the systems affected does not react in both our API changes and in API deprecation, the latter clearly presents more reactions.

In summary, a minority of systems that are moderately active update their

source code to keep up-to-date with the API evolution. In contrast with the API deprecation case, a majority of systems that are moderately active update their source code. Moreover, in both studies many systems do not update either because they are dead or stagnant systems, or because they froze their dependency to an old version of the library or framework. As a consequence, the effort of migrating to newer versions becomes more expensive over time due to change accumulation.

6.8 Consistency of Change Propagation (1)

Next, we present the results of our fifth research question about the consistency of change propagation.

RQ5. Do the systems react to an API change in the same way?

6.8.1 General results

Ideally, an API change should provide a single replacement, making the adaptation simpler for client developers. In that respect, the API changes analyzed in the previous research questions described the main way the analyzed frameworks and libraries evolved. However, some API changes may have more complex solutions, allowing multiple replacements [RLR12]. For example, Table 6.6 shows three examples of API changes extracted from the analyzed frameworks and libraries, and their reactions by the ecosystem.

Table 6.6: Examples of API changes; the numbers show the confidence of the replacement in the ecosystem.

Old call	New call	
	Framework/library	**Ecosystem reaction**
doSilently()	suspendAllWhile()	80% suspendAllWhile()
Pref.standardMenuFont()	StandardFonts.menuFont()	40% StandardFonts.menuFont() 40% ECPreferences.menuFont()
SecHashAlgorithm.new()	SHA1.new()	63% HashFunction.new() 30% SHA1.new()

    The first API change, doSilently() → suspendAllWhile(), is mostly followed by the ecosystem, presenting a confidence of 80% (*i.e.*, 80% of the commits that removed the old call also added the new call). For the second API change, Preferences.standardMenuFont() → StandardFonts.menuFont(), the ecosystem reacts with two possible replacements, both with confidence of 40%. For the third API change, SecureHashAlgorithm.new() → SHA1.new(), the ecosystem also reacts with two possible replacements: a main one with confidence of

63% and an alternative[10] one with 30%. Notice that, in this case, the main replacement is not the one extracted from the analyzed framework/library, *i.e.*, it is HashFunction.new() instead of SHA1.new().

To better understand such cases, we analyze the consistency of the API changes by verifying the reactions of the ecosystem.

Consistency of main and alternative replacements in the ecosystem

Figure 6.13a presents the confidence distribution of the main and alternative replacements in the ecosystem. For the main replacement, the 1st quartile is 36%, the median is 60%, and the 3rd quartile is 100%. For the alternative replacement, the 1st quartile is 20%, the median is 25%, and the 3rd quartile is 31%.

This results show that alternative replacements can be found in the ecosystem (such as the second and third examples in Table 6.6), but with less confidence than the main ones. Therefore, alternative replacements explain a minority of the cases where affected systems do not react to the prescribed API changes.
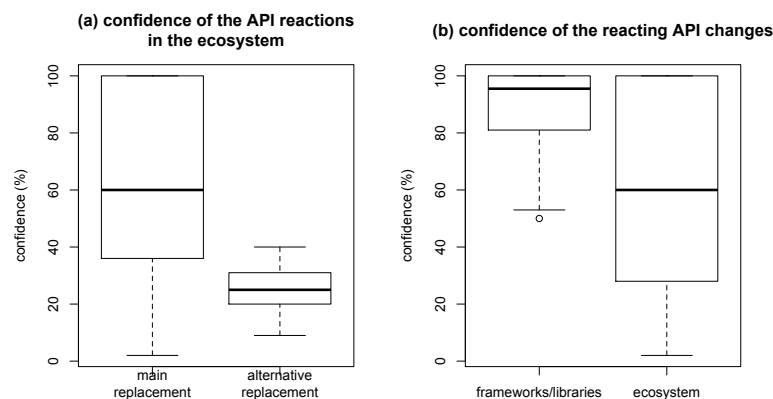
Figure 6.13: Box plots for the confidence of (a) the reaction in the ecosystem (main and alternative replacements) and (b) the reacting API changes (frameworks/libraries and ecosystem).

Consistency of API changes in the frameworks/libraries and in the ecosystem

Figure 6.13b compares the confidence distribution of the 62 reacting API changes both in the analyzed frameworks/libraries and in the ecosystem.

---

[10]Main and alternative replacements of API changes in the ecosystem are determined by verifying how the ecosystem replaces the old calls. This is done by applying our approach described in Section **??** in the ecosystem itself instead of the frameworks/libraries.

In the analyzed frameworks/libraries, the minimum is 53%, the 1st quartile is 81%, the median is 95%, and the 3rd quartile is 100% (recall that a minimum confidence of 50% was adopted to generate the rules from the analyzed frameworks/libraries as described in Subsection 6.3.2). In the ecosystem, for the *same* API changes, the minimum is 2%, the 1st quartile is 28%, the median is 60%, and the 3rd quartile is 100%. Two observations can be derived from such distributions.

First, there is clearly a difference in the confidence: the API changes are more consistently followed by the frameworks/libraries than by the ecosystem. This suggests that many replacements are not resolved in a uniform manner in the ecosystem: client developers may adopt other replacements in addition to the prescribed ones (such as the second and third examples in Table 6.6); method calls may be simply dropped, so they disappear without replacements; and developers may replace the old call by local solutions. Thus, this result provides evidence that API changes can be more confidently extracted from frameworks/libraries than from clients (*i.e.*, the ecosystem).

Second, confidence of the reacting API changes in the ecosystem (Figure 6.13b right side) is mostly equivalent to the main replacement reactions (Figure 6.13a left side). This implies that, in the majority of the cases, the API changes proposed by the frameworks/libraries are the main replacement in the ecosystem. In other words, it means that API changes such as the first one in Table 6.6 (*i.e.*, where the ecosystem follows the framework/library) represent the majority of the cases. In contrast, API changes such as the third one in Table 6.6 (*i.e.*, where the ecosystem does not follow the framework/library) represent the minority of the cases.

6.8.2   Time-based results

The age of the API changes may also influence the consistency of reactions. We investigate whether earlier API changes are more heterogeneous in their reactions (more reactions, more opportunity to diverge). Figure 6.14 presents the distribution shown in Figure 6.13b separated by earlier and later API changes.

In the frameworks and libraries (Figure 6.14a), the median is 95% for the earlier changes and 100% for the later changes. Comparing both earlier and later give a *p-value* > 0.05 and effect size $= 0.06$. Even though, the difference between the median is small, earlier API changes present overall less confidence, implying that their reactions are slightly more heterogeneous than the later ones.

In the ecosystem (Figure 6.14b), the difference between earlier and later API changes are clearer. In this case, for the earlier changes, the 1st quartile

is 21%, the median is 35%, and the 3rd quartile is 60%. For the later changes, the 1st quartile is 61%, the median is 85%, and the 3rd quartile is 100%. Comparing both earlier and later give a *p-value* < 0.01 and effect size = 0.46. This confirms that, in the ecosystem, earlier API changes are more heterogeneous in their reactions. We can conclude that as old API changes produce more reactions over time (as shown in the time-analysis of RQ3), such reactions are more likely to diverge.
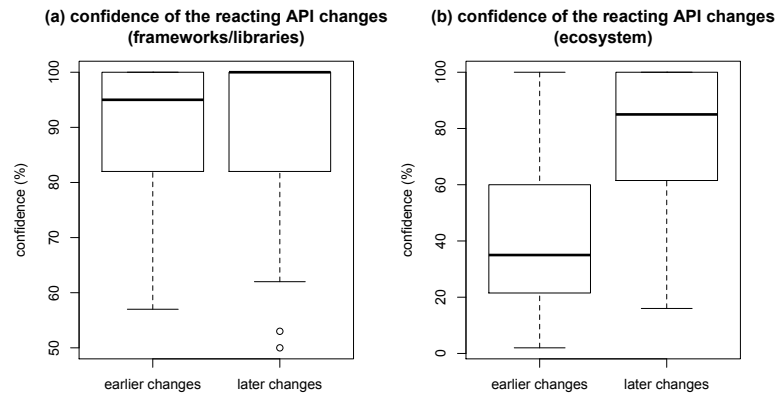
Figure 6.14: Box plots for the confidence of earlier and later reacting API changes in the (a) frameworks/libraries and (b) ecosystem.

6.8.3 Comparison with API deprecation

For the main replacement in the API deprecation study, the 1st quartile is 46%, the median is 60%, and the 3rd quartile is 80% (compared to 36%, 60%, and 100%, respectively, in our study). Even if the API changes do not produce warnings to alert the developer, the median at 60% shows that the distribution of the main replacement is mostly equivalent for API changes and deprecation.

Alternative replacements are not analyzed in the API deprecation study, so we cannot compare with our results.

6.9 Consistency of Change Propagation (2)

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

RQ6. How followed are the API changes by the ecosystem?

6.9.1 General results

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

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

Figure 6.15a shows the distribution of the classification for the 62 reacting API changes. The minority of the API changes, 4 (6%) are rarely followed; 21 (34%) are somewhat followed; and 37 (60%) are mostly followed — from such, 18 (29%) are totally followed with a confidence of 100%.

Figure 6.15b presents that only 13 (21%) API changes have multiple replacements in the ecosystem. Thus, this explains roughly half of the cases where the API changes are not consistently followed by the ecosystem (*i.e.*, the *rarely* and *somewhat* categories). The other half of the cases where the API changes are not consistently followed is due, for example, the drop of method calls with no replacement, or the use of local solutions by the client developers. This investigation remains future work.
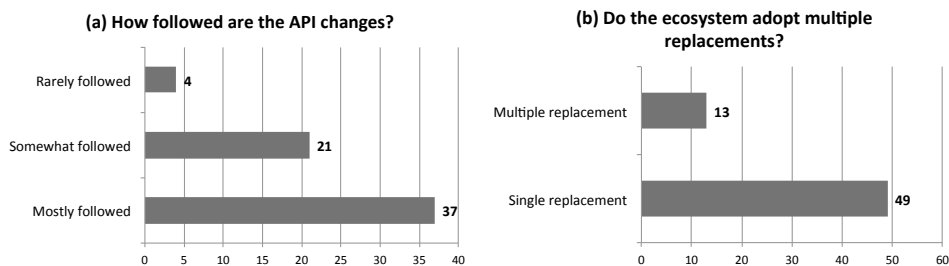
Figure 6.15: (a) How client developers follow the API changes, and (b) number of single and multiple replacements in the ecosystem.

6.9.2 Time-based results

In this subsection we investigate whether earlier API changes are more or less followed. Figure 6.16a shows that, for the earlier API changes, 4 (13%) are rarely followed; 16 (52%) are somewhat followed; and 11 (35%) are mostly followed. In contrast, for the later API changes, 0 are rarely followed; 5 (16%) are somewhat followed; and 26 (84%) are mostly followed. Figure 6.16b shows that 11 (35%) earlier API changes and 2 (6%) later API changes have multiple replacements.

In summary, earlier API changes are less followed than the later ones. Moreover, as API changes become old, other replacements are adopted by the ecosystem than the ones prescribed by the frameworks and libraries.
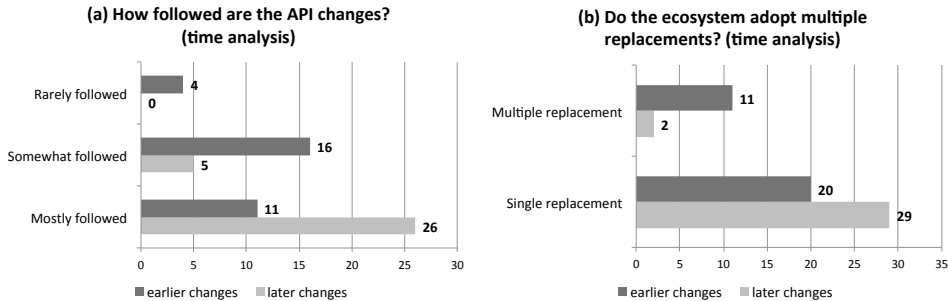
Figure 6.16: Time analysis for (a) how client developers follow the API changes, and (b) number of single and multiple replacements in the ecosystem.

6.9.3   Comparison with API deprecation

Such experiment was not performed by the API deprecation study.

6.10   Implications

The previous analysis allowed us to better understand, at the ecosystem level, to which extent client developers were impacted by API evolution of frameworks and libraries. Thus, the answers to our research questions allow us to formulate the following implications of our study.

API changes have a large impact on the ecosystem

Our study shows that 53% (62 out of 118) of the analyzed API changes caused reaction in 178 systems and affected 134 developers. A single API change impacted 41 systems and 37 developers. Overall, the reaction time of API changes is not quick. Client developers, naturally, need some time to discover and apply the new API; this time is even longer for the API changes about method suggestion. In contrast, the adaptation of the API changes in most of the systems occurs quickly. However, large systems may take a very long time to completely adapt.

A large amount of systems are potentially affected by the API changes. We show that 2,188 systems may be affected by the API changes with an average of 127 systems per API change. In fact, the number of affected systems, packages, classes, methods, and developers are much higher than those that actually react to API changes, *i.e.*, the majority of the systems do not react at

all. This could be due to several reasons: either because they are unaware or dormant systems, or because they follow a specific framework/library version. A minority of this lack of reactions is explained by client developers reacting in a way different of the one proposed by our API changes (*i.e.*, they are not following the main recommendation of the frameworks and libraries).

Time plays an important role on the analysis of API evolution

Our time-based results show that the age of an API change should be taken into account in the analysis. For all research questions, we observed distinct distributions between earlier and later API changes. Overall, earlier API changes have more propagation, cause slightly more reactions, have longer adaptation time, and affect slightly more systems. Moreover, earlier API changes cause more heterogeneous replacements, and are less followed. Therefore, such results show in large-scale level that as time passes: (i) the frequency, magnitude, duration and extension of the reactions tend to increase, and (ii) the reactions tend to be less consistent.

Deprecation mechanisms should be more adopted

Half of the API changes analyzed in this work (59 out of 118) were about method replacement. It means that such API changes were probably missing to use explicit deprecation mechanisms. Ideally, they should have been marked as deprecated by the framework and library developers. In fact, in large frameworks, developers may not know whether their code is used by clients: this may cause a growth [RLR12] or a lack in the use of deprecation [WGAK10, DJ05].

In our study, this lack of deprecation was mainly due to large refactorings in the frameworks and libraries. For instance, the library for dealing with files completely changed[11] after Pharo 1.4. As a result, some APIs missed to be marked as deprecated; *e.g.*, in the Moose migration to Pharo 3.0, a developer noticed this issue and commented[12]: "In FileSystem, ensureDirectory() was renamed to ensureCreateDirectory() without a deprecation", the library developer then answered: "Fill up a bug entry and we will add this deprecation. Good catch". In fact, for such cases, asking in mailing lists is the current alternative for client developers[13], confirming that mailing lists are popular tools developers use to satisfy their ecosystem-related information needs [?].

---

[11]http://stackoverflow.com/questions/15757529/porting-code-to-pharo-2-0

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

[13]Examples of mailing coordination involving API changes about method replacement found in our work: http://goo.gl/k9Fl0K, http://goo.gl/UVO910.

Another reason to the lack of deprecation is the adoption of naming conventions (*e.g.*, moving methods to a package named "deprecated") instead of adopting explicit deprecation mechanisms, which is a better solution for clients. An in-depth investigation to detect and avoid forgotten API deprecation remains future work.

Client developers use internal parts of frameworks or libraries

All the internal API changes (*i.e.*, 10) analyzed in this work affected client systems. From such internal API changes, 5 caused the clients to react as in the frameworks or libraries. Thus, our results reinforce (at large-scale and ecosystem level) previous studies [DR08, BR06, Bus13, BSvdB13] in the sense that even if client systems should only use public APIs, they also use internal parts of frameworks or libraries to access functionalities not available in the public interfaces for a variety of reasons.

For example, some internal APIs used in the ecosystem are related to internal parts of the Pharo language such as its graphical interface, AST engine, hash algorithm implementation, versioning system, among others, which, ideally, should not be used by clients.

Replacements are not resolved in a uniform manner

Many replacements are not resolved in a uniform manner in the ecosystem. Client developers may adopt other replacements in addition to the prescribed ones; method calls may be simply dropped; and developers may replace the old call by local solutions. As a result, in our case study, API changes can be more confidently extracted from frameworks/libraries than from clients. In fact, there is no clear agreement on this topic: while some studies propose the extraction of API changes from frameworks/libraries (*e.g.*, [DR08]) other adopt the extraction from clients (*e.g.*, [SJM08]). This study reinforces frameworks/libraries as a more reliable source to extract API changes.

Reactions to API changes can be partially automated

As we observed, many systems do not react to the API changes because they are not aware. Moreover, in the case of large client systems, the adaptation may take a large time and is costly if done manually. In practice, most of the API changes that we found in this work can be implemented as rules in current static analysis tools such as FindBugs [HP04], PMD [Cop05], and SmallLint [RBJ97]. Thus, these rules (attached to a confidence level) could help client developers to keep their source code up-to-date with the new APIs. Other API changes are more complex [SJM08] than the ones investi-

gated in this work, involve complex refactorings, and are not easy to auto-mate [CW12].

In addition, the comparison with API deprecation allow us to formulate the next implications:

API changes are less visible than API deprecation

Our comparisons show that API deprecation cause more reactions in the ecosystem than API changes. Also, in the API deprecation two or more developers involved in the same systems normally react while in the API changes only one developer is involved in the reaction. This shows that the knowledge about our API changes is more concentrated, so spreading it among developers is important.

Moreover, the reaction to deprecated APIs is faster than to the API changes. This occurs because deprecated methods produce warnings to developers with recommendation replacements while, in the API change case, developers need time to discover and apply the new API.

API changes are not as followed as API deprecation

Our comparisons show that the majority of the affected systems do not react neither in our API changes nor in API deprecation. However, the former clearly presents less reaction. When considering only systems that are moderately active, a majority updates their source code in the API deprecation case while only a minority updates their source code in the API change case.

As a result, the effort of porting to newer versions becomes more expensive in both approaches due to change accumulation, especially in the API change case. This reinforces the need of an automated approach to keep client developers aware about API changes of frameworks and libraries.

6.11   Threats to Validity

6.11.1   Construct Validity

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

Software ecosystems present some instances of duplication (around 15% of the code [SLR12]), where packages are copied from a repository to another (*e.g.,* a developer keeping a copy of a specific library version). This may over-estimate the number of systems reacting to an API change.

With respect to the API changes used in our study, they were manually validated by the authors of thesis with the support of documentation and code examples to eliminate incorrect and noisy ones.

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

Another factor that alleviates this threat is our focus on specific evolution rules (*i.e.,* a specific replacement of one or more calls by one or more calls). For research questions 1 to 3, we include only commits that are removing an old API *and* adding a new API to detect an API reaction. Requiring these two conditions to be achieved, decreases—or in some cases eliminates—the possibility of noise. For research question 4, we require the presence of the methods that contain a call to the old API. In this case, the noise could have been an issue, however, this threat is reduced since we discarded the API changes involved with common methods, *i.e.,* the noisy ones.

6.11.2 Internal Validity

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

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

6.11.3 External Validity

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

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

The Pharo ecosystem is a Smalltalk ecosystem, a dynamically typed programming language. Ecosystems in a statically typed programming lan-

guage may present differences. In particular, we expect static type checking to reduce the problem of noisy API changes for such ecosystems.

Our study considers API changes that were not marked as deprecated. In the previous work of [RLR12], the authors consider only changes that were marked as deprecated. Thus, there is no overlap between the changes investigated in this work and the ones investigated by that study. In fact, these studies complement each other in order to better characterize the phenomenon of change propagation at the ecosystem level.

6.12 Summary

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

- Many API changes can have a large impact on the ecosystem in terms of projects, packages, classes and methods affected as well as developers. Moreover, client developers need some time to discover and apply the new API, and the majority of the systems do not react at all. Such analysis can be influenced by the age of the API change.

- API changes can not be marked as deprecated because framework developers are not aware of their use by clients, which may be aggravated by large refactorings in the frameworks. Moreover, client developers can use internal parts of frameworks to access functionalities not available in the public interfaces.

- Replacements can not be resolved in a uniform manner in the ecosystem. Thus, API changes can be more confidently extracted from frameworks/libraries than from clients.

- Most of the analyzed API changes can be implemented as rules in static analysis tools in order to reduce the adaptation time or the amount of projects that are not aware about a new/better API.

- Information about API changes can be concentrated in a small amount of developers. It is important to share it among more developers, similarly to what happens for API deprecation. Overall, API changes and deprecation can present different characteristics, for example, reaction to API changes is slower and less clients adapts.

Conclusion

Contents

7.1 As a Conclusion

Software evolution is naturally a complex task. The impact of software evolution may be large and sometimes unknown. We need to ensure that changes are consistently propagated to dependent systems.

We reviewed several approaches to support software evolution. We organized them in three parts: (i) analysis of generic and expert-based rules, (ii) extraction of history-based rules, and (iii) analysis of software ecosystems. Existing approaches lack of a deep understanding of the benefits provided by expert-based rules, a better use of source code history to extract history-based rules, and an analyze of the impact of source code changes in the actual clients.

In this thesis, we argue for the need to analyze and improve rules as to better support developers keeping track of source code changes. We cover three aspects: (i) benefits of expert-based rules, (ii) improvement of history-based rules, and (iii) the impact of source code changes in a software ecosystem.

We provide (i) new experiments on the relationship between violations generated by expert-based rules and defects [HADA12], (ii) two novel approaches to extract history-based rules from code repository to better support evolving systems and their clients [HADV13,HEA$^+$14,HAE$^+$15b,HAE$^+$15a], and (iii) a large-scale study, at the ecosystem level, to understand to which extent client developers are impacted by source code evolution [HRA$^+$15].

We evaluated each approach with the use of research questions that were answered when necessary with the use of statistical tests. The experts also played an important role in our validation, assessing the extracted history-based rules.

Next, we present a summary and we reiterate the most interesting conclusions we derived from our study.

Benefits of Expert-based Rules

This work reported on a systematic study to investigate the relation between generic or system-specific violations and observed defects. The study was performed on Seaside, a real world-system, that has been used and maintained for years, and for which system-specific rules were created by experts. To the best of our knowledge, this study is the first to use system-specific rules created by experts to defect prevention. For the case study under analysis, generic rules were not effective enough to be used for defect prevention, confirming results of previous publications. In contrast, system-specific rules provide more relevant information on how to avoid defects, and, therefore, they are more effective to be used for defect prevention. We expect system-specific rules to be created and used by developers in complement to generic ones for defect prevention [HADA12].

Supporting System-specific Conventions with History-based Rules

In this study, we proposed to automatically extract system-specific conventions from source code history. In this process, we extract data from incremental revisions in source code history, and the rules are based on predefined patterns and filtered by their occurrence over different revisions. We validated our approach on open-source systems with the help of an expert, which was very valuable to provide assessment about the change rules. A relevant amount of rules (62%, 28 out of 45) were correct to the expert in our case study, pointing to real violations in source code. In total, 15 rules generated violations, producing a total of 58 violations from which 47 (81%) were real ones. The discovering of 28 new system-specific rules represents a significant addition to the set of rules provided by the static analysis tool SmallLint since it contains only 19 generic change rules [HADV13, HAE+15b].

Supporting Client Systems with History-based Rules

This study provided a novel approach to generate evolution rules by monitoring changes applied in source code during the framework or library evolution. Rules are also mined from source code history taking into account the changes between revisions, and they follow four types of method replacements. Furthermore, our approach provides either on request rules or automatically generated rules, and our tool is able to show code examples of the rules that clarify how the framework/library adapted to its own changes. This study was evaluated on five open-source systems, and the rules were assessed with the help of experts with respect to their validity. For the cases were the experts were confident the precision remained between 65% and 86%. This work is the first to produce rules *one-to-one*,

one-to-many, *many-to-one* and *many-to-many*, involving method replacement and suggestion, thus, improving the spectrum of rules generated by current studies [HEA$^+$14, HAE$^+$15a].

Impact of Software Evolution on Ecosystems

This work reported on an empirical study about the impact of API evolution, in the specific case of methods unrelated to API deprecation. The study was done in the context of a large-scale ecosystem, Pharo. We analyzed 118 API changes, and we found that many impacted other systems. Our findings can be summarized as follows: (i) API changes from frameworks can have a large impact on the ecosystem in terms of projects, packages, classes and methods affected as well as developers; (ii) time variable plays an important role on the analysis of API evolution; (iii) deprecation mechanisms should be more adopted; (iv) client developers can use internal parts of frameworks to access functionalities not available in the public interfaces; (v) replacements are not resolved in an uniform manner; (vi) reactions to API changes can be partially automated; and (vii) API changes and deprecation can present different characteristics, for example, reaction to API changes is slower with less clients adapting. The results of this study help to characterize the impact of API evolution in large software ecosystems [HRA$^+$15].

7.2 Future Work

There are some open issues that were not addressed in this thesis, but should be explored in future work.

Extracting Other Types of Rules

In this thesis, we focus on the extraction of rules that involve the replacement of methods. In Section 4.5.3, we pointed some solutions to extract other types of rules from source code. For example, rules can also be related to simply-deleted calls (methods that in general should not be called), method pairs (methods that normally should appear together in a class) or method call pairs (methods that normally should be called together).

Even if previous studies address some of these types, there is no effort in the literature to provide them as a set of rules to developers. Based on that, as a future work it is interesting an approach to unify such types of rules as well as the one adopted in this thesis to better support developers.

Better Characterization of API Evolution

Many approaches are proposed to deal with API evolution. In order to produce rules, these approaches need to mine code repositories. In this context, while some studies, including ours, are dedicated to mine commits from source code history (*e.g.*, [DR08,MWZM12]) others mine releases of software systems (*e.g.*, [SJM08, WGAK10]). The mining at the commit level seems to be more adopted by the literature, however, the exact gain provided by this alternative is not clear.

The same analogy can be performed at other levels, for example, is it better to extract data from frameworks or clients? Is it better to compare versions of classes or methods? The benefit provided by each alternative should be better investigated in order to help researchers in the task of choosing the best suited level of analysis.

Ecosystem Analysis

In this thesis, we focused on the analysis of a dynamic ecosystem, Pharo. This was done due to many reasons such as the fact that this ecosystem is concentrated in two repositories and we were interested in comparing our results with the work of Robbes *et al.* [RLR12] (as described in Section 6.2). Still, other ecosystems should be tested, for example, the ones provided by statically typed programming languages.

It is particularly relevant to compare dynamically and statically typed ecosystems. Due to the own nature of dynamic ecosystems, it is expected that many problems related to API evolution is detected in runtime instead of compile time, then increasing the time projects remain inconsistent. However, it is not clear whether this issue comes more from the nature of the language itself or from the fact that dealing with API evolution is a complex task. In other words, by comparing both type of ecosystems, one could verify to which extent the dynamic nature of the ecosystem influences on the API evolution reaction.

7.3   Collaboration

During the Ph.D, I had the opportunity to collaborate with the ASERG/UFMG group (Belo Horizonte, Brazil) and with the Siemens Research & Technology Center (Erlangen, Germany).

ASERG/UFMG group

In this collaboration I visited such group two times (December/2011 and December/2012). We have mainly worked in the context of bug-detection and

prevention as well as architecture conformance. From from collaboration, seven papers were published in conferences and journals [MVT+15, CVP+14, CPV+13, MVB+13a, MVB+13b, HAD+12, AAHD12].

Siemens Research & Technology Center

In this collaboration I visited such center one time (October and November/2014). We have mainly worked in the context of API evolution, by applying the approach described in Chapter 5 in C# Siemens systems. Moreover, we have also worked with the evolution of other software artifacts (*e.g.*, XML files). The goal was to support automatic API migration and documentation generation.

Bibliography

[AAHD12]   Simon Allier, Nicolas Anquetil, André Hora, and Stephane Ducasse. A Framework to Compare Alert Ranking Algorithms. In *Working Conference on Reverse Engineering*, pages 277–285, 2012. 123

[BCDL13]   Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep into Pharo*. Square Bracket Associates, 2013. accessed: 15 May 2014. 47, 70

[BCP⁺13]   Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. The evolution of project inter-dependencies in a software ecosystem: the case of Apache. In *International Conference on Software Maintenance*, pages 280–289. IEEE, 2013. 1, 16, 86

[BDN⁺09]   Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009. 47, 70

[BM08]   Cathal Boogerd and Leon Moonen. Assessing the Value of Coding Standards: An Empirical Study. In *International Conference on Software Maintenance*, pages 277 –286, 2008. 3, 8, 19, 20, 21, 22, 23, 30, 31, 35, 36, 37

[BM09]   Cathal Boogerd and Leon Moonen. Evaluating the Relation Between Coding Standard Violations and Faults Within and Across Software Versions. In *Working Conference on Mining Software Repositories*, pages 41–50, 2009. 3, 8, 19, 20, 21, 22, 23, 30, 31, 35, 36, 37

[BR06]   Jean-Sebastien Boulanger and Martin P. Robillar. Managing concern interfaces. In *International Conference on Software Maintenance*, pages 14–23, 2006. 2, 94, 114

[BSvdB13] John Businge, Alexander Serebrenik, and Mark GJ van den Brand. Eclipse api usage: the good and the bad. *Software Quality Journal*, pages 1–35, 2013. 2, 114

[BTF05]   Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. *SIGPLAN Not.*, 40(10):265–279, 2005. 2, 61

[Bun09]   Philipp Bunge. Scripting browsers with Glamour. Master's thesis, University of Bern, 2009. 70

[Bus13]   John Businge. Co-evolution of the eclipse framework and its third-party plug-ins. *Faculty of Mathematics and Computer Science, TU/e*, 9, 2013. 2, 114

[BvdB06]   Wojciech Basalaj and Frank van den Beuken. Correlation Between Coding Standards Compliance and Software Quality. Technical report, Programming Research, 2006. 3, 8, 19, 23, 30, 35, 36, 37

[CMG14]   Maëlick Claes, Tom Mens, and Philippe Grosjean. On the maintainability of cran packages. In *Software Evolution Week (European Conference on Software Maintenance and Working Conference on Reverse Engineering)*, 2014. 16

[CMSV12]   Cesar Couto, Joao Eduardo Montandon, Christofer Silva, and Marco Tulio Valente. Static Correspondence and Correlation Between Field Defects and Warnings Reported by a Bug Finding Tool. *Software Quality Journal*, pages 1–17, 2012. 3, 8, 19, 20, 23, 37

[CN96]   Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *International Conference on Software Maintenance*, pages 359–368, 1996. 12, 85

[Cop05]   Tom Copeland. *PMD Applied*. Centennial Books, 2005. 3, 7, 25, 114

[CPV+13]   César Couto, Pedro Pires, Marco Tulio Valente, Roberto Bigonha, Andre Hora, Nicolas Anquetil, et al. Bugmaps-granger: A tool for causality analysis between source code metrics and bugs. In *Brazilian Conference on Software: Theory and Practice (CBSoft'13)*, 2013. 123

[CVP+14]   Cesar Couto, Marco T Valente, Pedro Pires, Andre Hora, Nicolas Anquetil, and Roberto S Bigonha. Bugmaps-granger: a tool for visualizing and predicting bugs using granger causality tests. *Journal of Software Engineering Research and Development*, 2(1):1, 2014. 123

[CW12]   Bradley E. Cossette and Robert J. Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *International Symposium on the Foundations of Software Engineering*, page 55. ACM, 2012. 12, 115

[DAB+11]   Stéphane Ducasse, Nicolas Anquetil, Muhammad Usman Bhatti, André Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Technical report, 2011. 47, 70

[DJ05]   Danny Dig and Ralph Johnson. The role of refactorings in api
             evolution. In *International Conference on Software Maintenance*,
             pages 389–398, 2005. 12, 86, 90, 113

[DJ06]   Danny Dig and Ralph Johnson. How do APIs evolve? a story
             of refactoring: Research articles. *Journal of software maintenance
             and evolution: research and practice*, 18(2):83–107, 2006. 2, 61

[DLR07]   Stephane Ducasse, Adrian Lienhard, and Lukas Renggli. Sea-
             side: A Flexible Environment for Building Dynamic Web Appli-
             cations. *IEEE Software*, 24:56–63, 2007. 25, 70

[DR08]   Barthelemy Dagenais and Martin P. Robillard. Recommending
             adaptive changes for framework evolution. In *International Con-
             ference on Software engineering*, pages 481–490, 2008. 2, 3, 13, 14,
             15, 61, 81, 83, 85, 94, 114, 122

[Erl00]   Len Erlikh. Leveraging legacy system dollars for e-business. *IT
             professional*, 2(3):17–23, 2000. 1

[FSV11]   Joao Araujo Filho, Silvio Souza, and Marco Tulio Valente. Study
             on the Relevance of the Warnings Reported by Java Bug-Finding
             Tools. *Software, IET*, 5(4):366–374, 2011. 10, 52

[GAH13]   Daniel M German, Bram Adams, and Ahmed E Hassan. The
             evolution of the R software ecosystem. In *European Conference
             on Software Maintenance and Reengineering*, pages 243–252. IEEE,
             2013. 16

[GBRM+09]   Jesus M. Gonzalez-Barahona, Gregorio Robles, Martin
             Michlmayr, Juan Jose Amor, and Daniel M. German. Macro-
             level software evolution: a case study of a large software
             compilation. *Empirical Software Engineering*, 14(3):262–285,
             2009. 16

[GGP13]   Amruta Gokhale, Vinod Ganapathy, and Yogesh Padmanaban.
             Inferring likely mappings between apis. In *International Confer-
             ence on Software Engineering*, pages 82–91, 2013. 12

[HAD+12]   Andre Hora, Nicolas Anquetil, Stephane Ducasse, Muhammad
             Bhatti, Cesar Couto, Marco Tulio Valente, and Julio Martins.
             Bug maps: A tool for the visual exploration and analysis of bugs.
             In *European Conference on Software Maintenance and Reengineer-
             ing*, pages 523–526. IEEE, 2012. 123

[HADA12]   André Hora, Nicolas Anquetil, Stephane Ducasse, and Simon
             Allier. Domain Specific Warnings: Are They Any Better? In
             International Conference on Software Maintenance, 2012. 4, 20, 119,
             120

[HADV13]   André Hora, Nicolas Anquetil, Stéphane Ducasse, and
            Marco Tulio Valente.   Mining System Specific Rules from
            Change Patterns. In *Working Conference on Reverse Engineering*,
            2013. 5, 10, 14, 39, 52, 119, 120

[HAE⁺15a]   André Hora, Nicolas Anquetil, Anne Etien, Stephane Ducasse,
            and Marco Tulio Valente. Mining api change rules. *Journal of
            Software: Evolution and Process*, 2015. *In progress.* 5, 62, 119, 121

[HAE⁺15b]   André Hora, Nicolas Anquetil, Anne Etien, Stephane Ducasse,
            and Marco Tulio Valente. Mining system-specific rules to im-
            prove static analysis. *Software Quality Journal*, 2015. *Under sub-
            mission.* 5, 39, 119, 120

[HD05]   Johannes Henkel and Amer Diwan. Catchup!: Capturing and
            replaying refactorings to support api evolution. In *International
            Conference on Software Engineering*, pages 274–283, 2005. 12, 85

[HEA⁺14]   André Hora, Anne Etien, Nicolas Anquetil, Stéphane Ducasse,
            and Marco Tulio Valente.   APIEvolutionMiner: Keeping API
            Evolution under Control. In *Software Evolution Week (European
            Conference on Software Maintenance and Working Conference on Re-
            verse Engineering)*, pages 420–424, 2014. 5, 62, 69, 85, 90, 119, 121

[HP04]   David Hovemeyer and William Pugh. Finding Bugs is Easy. In
            Object Oriented Programming Systems Languages and Applications,
            pages 132–136, 2004. 3, 7, 25, 114

[HRA⁺15]   André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien,
            and Stephane Ducasse. How do developers react to api evo-
            lution? the case of the pharo ecosystem. *Empirical Software En-
            gineering*, 2015. *Under submission.* 2, 5, 86, 119, 121

[JBC13]   Slinger Jansen, Sjaak Brinkkemper, and Michael Cusumano.
            *Software Ecosystems: Analyzing and Managing Business Networks
            in the Software Industry*. Edward Elgar Pub, 2013. 15

[JSW11]   Corey Jergensen, Anita Sarma, and Patrick Wagstrom.   The
            onion patch: migration in open source ecosystems. In *European
            Software Engineering Conference and International Symposium on
            the Foundations of Software Engineering*, pages 70–80. ACM, 2011.
            15, 16

[KAYE04]   Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson En-
            gler. Correlation Exploitation in Error Ranking. In *International
            Symposium on the Foundations of Software Engineering*, pages 83–
            93, 2004. 19

[KE07]   Sunghun Kim and Michael D. Ernst. Which Warnings Should
             I Fix First? In *European Software Engineering Conference and In-
             ternational Symposium on the Foundations of Software Engineering*,
             pages 45–54, 2007. 10, 11, 19, 20, 21, 23, 30, 35, 36, 37, 39, 52

[KN09]   Miryung Kim and David Notkin. Discovering and Representing
             Systematic Code Changes. In *International Conference on Software
             Engineering*, pages 309–319, 2009. 12, 42

[KNG07]   Miryung Kim, David Notkin, and Dan Grossman. Automatic
             inference of structural changes for matching across program
             versions. In *International Conference on Software Engineering*,
             pages 333–343. IEEE, 2007. 12

[KPW06]   Sunghun Kim, Kai Pan, and E. James Whitehead. Memories
             of Bug Fixes. In *International Symposium on the Foundations of
             Software Engineering*, pages 35–45, 2006. 3, 9, 11, 19, 39, 42, 52,
             53, 56

[KZPW06]   Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James
             Whitehead. Automatic Identification of Bug-Introducing
             Changes. In *International Conference on Automated Software En-
             gineering*, pages 81–90, 2006. 22, 37

[Leh96]   Manny M. Lehman. Laws of software evolution revisited. In
             Software process technology, pages 108–124. Springer, 1996. 1

[LKL08]   David Lo, Siau-Cheng Khoo, and Chao Liu. Mining Temporal
             Rules for Software Maintenance. *Journal of Software Maintenance
             and Evolution: Research and Practice*, 20(4):227–247, 2008. 10

[LLGR10]   Mircea Lungu, Michele Lanza, Tudor Gîrba, and Romain
             Robbes. The small project observatory: Visualizing software
             ecosystems. *Science of Computer Programming*, 75(4):264–275,
             2010. 16

[LRL10]   Mircea Lungu, Romain Robbes, and Michele Lanza. Recover-
             ing inter-project dependencies in software ecosystems. In *In-
             ternational Conference on Automated Software Engineering*, pages
             309–312. ACM, 2010. 16

[LRRV12]   David Lo, G. Ramalingam, Venkatesh-Prasad Ranganath, and
             Kapil Vaswani. Mining Quantified Temporal Rules: Formalism,
             Algorithms, and Evaluation. *Science of Computer Programming*,
             77(6):743–759, 2012. 10

[LTW⁺06]   Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou,
             and Chengxiang Zhai. Have Things Changed Now? An Empir-
             ical Study of Bug Characteristics in Modern Open Source Soft-

ware. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, 2006. 19

[Lun09]   Mircea Lungu. Reverse Engineering Software Ecosystems. *PhD thesis, University of Lugano, Switzerland (October 2009)*, 2009. 1, 15, 16, 83, 85

[LZ05]   Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 296–305, 2005. 3, 10, 11, 39, 42, 43, 48, 50, 57, 72

[MCG14]   Tom Mens, Maelick Claes, and Philippe Grosjean. Ecos: Ecological studies of open source software ecosystems. In *Software Evolution Week (European Conference on Software Maintenance and Working Conference on Reverse Engineering)*, 2014. 16

[MCGS14]   Tom Mens, Maálick Claes, Philippe Grosjean, and Alexander Serebrenik. Studying evolving software ecosystems based on ecological models. In Tom Mens, Alexander Serebrenik, and Anthony Cleve, editors, *Evolving Software Systems*, pages 297–326. Springer Berlin Heidelberg, 2014. 16

[MH13]   Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems - a systematic literature review. *Journal of Systems and Software*, 86(5):1294 – 1306, 2013. 16

[MRK13]   Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *International Conference on Software Maintenance*, pages 70–79. IEEE, 2013. 3, 16

[MS05]   David G. Messerschmitt and Clemens Szyperski. Software ecosystem: understanding an indispensable technology and industry. *MIT Press Books*, 1, 2005. 15

[MV00]   Audris Mockus and Lawrence G. Votta. Identifying Reasons for Software Changes using Historic Databases. In *International Conference on Software Maintenance*, pages 120–130, 2000. 21

[MVB⁺13a]   Cristiano Maffort, Marco Tulio Valente, Mariza Bigonha, Nicolas Anquetil, and Andre Hora. Heuristics for discovering architectural violations. In *Working Conference on Reverse Engineering*, pages 222–231. IEEE, 2013. 123

[MVB⁺13b]   Cristiano Maffort, Marco Tulio Valente, Mariza Bigonha, Andre Hora, Nicolas Anquetil, Jonata Menezes, et al. Mining architec-

tural patterns using association rules. In *International Conference on Software Engineering and Knowledge Engineering*, 2013. 123

[MVT⁺15] Cristiano Maffort, Marco Tulio Valente, Ricardo Terra, Mariza Bigonha, Nicolas Anquetil, and Andre Hora. Mining architectural violations from version history. *Empirical Software Engineering Journal*, 2015. 123

[MWZ11] Yana Momchilova Mileva, Andrzej Wasylkowski, and Andreas Zeller. Mining Evolution of Object Usage. In *European Conference on Object-Oriented Programming*, pages 105–129, 2011. 1, 9, 43, 48, 72

[MWZM12] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *International Conference on Software Engineering*, pages 353–363, 2012. 3, 13, 14, 15, 56, 61, 71, 73, 81, 82, 83, 85, 122

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of moose: an agile reengineering environment. In *European Software Engineering Conference and International Symposium on the Foundations of Software Engineering*, pages 1–10, 2005. 47, 70

[NNP⁺10] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. Recurring Bug Fixes in Object-Oriented Programs. In *International Conference on Software Engineering*, pages 315–324, 2010. 3, 9, 11, 39, 42

[NNW⁺10] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to api usage adaptation. In *International Conference on Object Oriented Programming Systems Languages and Applications*, pages 302–321, 2010. 12

[OWB04] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the Bugs Are. In *International Symposium on Software Testing and Analysis*, pages 86–96, 2004. 20

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3:253–263, 1997. 3, 7, 25, 114

[RDGN10] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Domain-Specific Program Checking. In *Objects, Models, Components, Patterns*, pages 213–232, 2010. 3, 8, 10, 20, 25, 29, 52

[RLR12]   Romain Robbes, Mircea Lungu, and David Röthlisberger. How
           do developers react to API deprecation? The case of a smalltalk
           ecosystem. In *International Symposium on the Foundations of Soft-
           ware Engineering*, pages 56:1–56:11. ACM, 2012. 1, 2, 3, 15, 16, 19,
           61, 74, 85, 86, 87, 88, 93, 95, 96, 104, 105, 107, 113, 117, 122

[RvDV13]   Steven Raemaekers, Arie van Deursen, and Joost Visser. The
           maven repository dataset of metrics, changes, and dependen-
           cies. In *International Conference on Mining Software Repositories*,
           pages 221–224, 2013. 88

[SJM08]   Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining frame-
           work usage changes from instantiation code. In *International
           Conference on Software engineering*, pages 471–480, 2008. 3, 13,
           14, 15, 61, 81, 82, 83, 85, 90, 114, 122

[SLR12]   Niko Schwarz, Mircea Lungu, and Romain Robbes. On how of-
           ten code is cloned across repositories. In *International Conference
           on Software Engineering*, pages 1289–1292. IEEE, 2012. 115

[Som10]   Ian Sommerville. *Software Engineering*. Addison-Wesley, 2010.
           1

[SSPR12]   Boya Sun, Gang Shu, Andy Podgurski, and Brian Robinson. Ex-
           tending static analysis by mining project-specific rules. In *In-
           ternational Conference on Software Engineering*, pages 1054–1063,
           2012. 3, 9, 11, 39, 42

[SSS08]   Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to ad-
           vanced empirical software engineering*, volume 5. Springer, 2008.
           93

[SZZ05]   Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller.
           When do Changes Induce Fixes? In *International Workshop on
           Mining Software Repositories*, pages 1–5, 2005. 21, 37

[WDA⁺08]   Stefan Wagner, Florian Deissenboeck, Michael Aichner, Johann
           Wimmer, and Markus Schwalb. An Evaluation of Two Bug Pat-
           tern Tools for Java. In *International Conference on Software Testing,
           Verification, and Validation*, pages 248–257, 2008. 3, 8

[WGAK10]   Wei Wu, Yann-gael Gueheneuc, Giuliano Antoniol, and
           Miryung Kim. Aura: a hybrid approach to identify framework
           evolution. In *International Conference on Software Engineering*,
           pages 325–334, 2010. 2, 3, 13, 14, 15, 56, 61, 71, 81, 82, 83, 85,
           86, 113, 122

[WH05]   Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic
           Mining of Source Code Repositories to Improve Bug Finding

Techniques. *Transactions on Software Engineering*, 31:466–480, 2005. 3, 9, 11, 39, 42

[WRH⁺00] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000. 23, 49

[ZJ12] Mohammed J. Zaki and Wagner Meira Jr. Fundamentals of data mining algorithms, 2012. 57, 66

[ZKZW06] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E. James Whitehead. Mining Version Archives for Co-changed Lines. In *International Workshop on Mining Software Repositories*, pages 72–75, 2006. 22

[ZTX⁺10] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining api mapping for language migration. In *International Conference on Software Engineering*, pages 195–204, 2010. 12