POLYPUBLIE

Understanding and Tooling Framework API Evolution

Wei Wu

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

Date	2014
Type	Mémoire ou thèse / Dissertation or Thesis Thèse de doctorat/Ph.D. thesis
URL de PolyPublie PolyPublie URL	https://publications.polymtl.ca/1642/
Conditions d'utilisation Terms of use	© Tous droits réservés / All rights reserved
Référence Citation	Wu, W. (2014). Understanding and Tooling Framework API Evolution [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1642/

Détails de la thèse
Thesis details

Directeurs de recherche Advisors	Yann-Gaël Guéhéneuc, & Giuliano Antoniol
Programme Program	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal
https://publications.polymtl.ca

UNIVERSITÉ DE MONTRÉAL

UNDERSTANDING AND TOOLING FRAMEWORK API EVOLUTION

WEI WU
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
OCTOBRE 2014

© Wei Wu, 2014.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

UNDERSTANDING AND TOOLING FRAMEWORK API EVOLUTION

présentée par : WU Wei
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. DAGENAIS Michel, Ph.D., président
M. GUÉHÉNEUC Yann-Gaël, Doctorat, membre et directeur de recherche
M. ANTONIOL Giuliano, Ph.D., membre et codirecteur de recherche
M. BELTRAME Giovanni, Ph.D., membre
M. LÄMMEL Ralf, Ph.D., membre externe

To my family...

ACKNOWLEDGEMENTS

I have been fortunate to have the opportunity to work with a team of experts from various domains in software engineering. Collaborating, sharing expertise and ideas with them inspired this work.

I am specially grateful to my supervisors, Yann and Giulio. They were always available for constructive feedbacks, no matter where they were, how many demanding tasks they had. We have been using all the possible communication tools to work together around the clock. Their programming, statistical, mathematical, and other skills and knowledge aided me to build this thesis. I really appreciate the guidance, patience, and challenges that they gave me. They led me to reach a level that I never imagined. Thank you Yann and Giulio!

I would like also to thank Foutse and Bram. Their encouragements and insightful comments and suggestions made an important part of this thesis. I will always remember how hard they tried to help me comb the ideas in my mind tangled like earphone wires in the pockets.

I would like to thank Adrien for the excellent work during his internship to organize our experiment. The website that he built saved us a lot of effort to collect and process experiment data. I would like to thank all the people participating the experiment.

I would like to thank Stéphane and Alain for the opportunity to do my internship at Benchmark and to analyse IRIS with ACUA.

I would like to thank Daniel German for providing data and Philippe Galinier for the helpful advices. The discussions with them helped me a lot.

I would like to thank all the members of Ptidej, SoccerLab, SWAT and MCIS teams for collaborations, discussions, suggestions, and all the activities that we did together, especially badminton. I really enjoyed the time with you guys!

I would like to thank my father, my mother, my brother, Wei, and Simon. Their support, patience and understanding were with me through all my Ph.D. program.

My doctoral studies were supported by an FQRNT doctoral research scholarship and the internship at Benchmark was supported by Mitacs.

RÉSUMÉ

Les cadres d'applications sont intensivement utilisés dans le développement de logiciels modernes et sont accessibles au travers de leur Application Programming Interface (API), qui définit un ensemble de fonctionnalités que les programmes clients peuvent utiliser pour accomplir des tâches. Les cadres d'applications ne cessent d'évoluer au cours de leurs vies pour satisfaire la demande de nouvelles fonctions ou pour rapiécer des vulnérabilités de sécurité. L'évolution des cadres d'applications peut engendrer des modifications de l'API auxquelles les programmes clients doivent s'adapter. Les mises à jour vers les nouvelles versions des cadres d'applications prennent du temps et peuvent même interrompre le service. Aider les développeurs à mettre à jour leurs programmes est d'un grand intérêt pour les chercheurs académiques et industriels.

Dans cette thèse, nous réalisons une étude exploratoire de la réalité des évolutions des API et de leurs usages dans le dépôt central de Maven et dans deux grands cadres d'applications avec de larges écosystèmes : Apache et Eclipse. Nous découvrons que les API changent dans environ 10 % des cadres d'applications et touchent 50 % des programmes clients. Il arrive plus souvent que des classes et des méthodes manquent et disparaissent dans les cadres d'applications. Ces classes et méthodes affectent les programmes clients plus souvent que les autres changements des API. Nous montrons aussi qu'environ 80 % des utilisations des API dans les programmes clients peuvent être réduits par refactoring. Forts de ce constat, nous faisons une expérience pour vérifier l'effectivité des règles de changement des API générés par les approches existantes, qui recommandent les remplacements pour les API disparues pendant l'évolution des cadres d'application. Nous confirmons que les règles de changement des API aident les développeurs à trouver des remplacements aux API manquantes plus précisément, en particulier pour des cadres d'applications difficiles à comprendre. Enfin, nous étudions l'efficacité des caractéristiques utilisées pour construire les règles de changement des API et différentes manières de combiner plusieurs caractéristiques. Nous soutenons et montrons que des approches basées sur l'optimisation multi-objective peuvent détecter des règles de changement des API plus précisément et qu'elles peuvent prendre en compte plus facilement de nouvelles caractéristiques que les approches précédentes.

ABSTRACT

Frameworks are widely used in modern software development and are accessed through their Application Programming Interfaces (APIs), which specify sets of functionalities that client programs can use to accomplish their tasks. Frameworks keep evolving during their lifespan to cope with new requirements, to patch security vulnerabilities, *etc.* Framework evolution may lead to API changes to which client programs must adapt. Upgrading to new releases of frameworks is time-consuming and can even interrupt services. Helping developers upgrade frameworks draws great interests from both academic and industrial researchers.

In this dissertation, we first present an exploratory study to investigate the reality of API changes and usages in Maven repository and two framework ecosystems: Apache and Eclipse. We find that API changes in about 10% of frameworks affect about 50% of client programs. Missing classes and missing methods happen more often in frameworks and affect client programs more often than other API changes. About 80% API usages in client programs can be reduced by refactoring. Based on these findings, we conduct an empirical study to verify the usefulness of API change rules automatically built by previous approaches, which recommend the replacements for missing APIs due to framework evolution. We show that API change rules do help developers find the replacements of missing APIs more accurately, especially for frameworks difficult to understand. We describe another empirical study to evaluate the effectiveness of features used to build API change rules and of different ways combining multiple features. We argue and show that multi-objective-optimization-based approaches can detect more correct change rules and are easier to extend with new features than previous approaches.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ v

ABSTRACT vi

TABLE OF CONTENTS vii

LIST OF TABLES x

LIST OF FIGURES xii

LIST OF ACRONYMS AND ABBREVIATIONS xiv

CHAPTER 1 INTRODUCTION 1
1.1 Framework API Changes and Usages 2
1.1.1 Our Contributions 2
1.2 Framework API Change Rules 6
1.2.1 Our Contributions 7
1.3 Organization of the Dissertation 10

CHAPTER 2 LITERATURE REVIEW 12
2.1 Framework API Changes and Usages 12
2.1.1 API Changes 12
2.1.2 API Usages 14
2.2 API Change Rule Building and Program Differentiation 17
2.3 Empirical Study on Program Comprehension 20
2.4 Multi-objective Optimization in Software Engineering 21
2.5 Summary 23

CHAPTER 3 EXPLORATORY STUDY ON API CHANGES AND USAGES . . . 24
3.1 Study Design 27
3.1.1 API Evolution Overview 27
3.1.2 API Changes 30

3.1.3 API Usages . . . 32
3.1.4 API Change Effects . . . 36
3.1.5 Research Questions . . . 37
3.2 Study Execution . . . 38
3.2.1 Tooling . . . 38
3.2.2 Dataset Description . . . 46
3.3 Study Results . . . 49
3.3.1 API Evolution Overview . . . 49
3.3.2 API Changes . . . 49
3.3.3 API Usages . . . 54
3.3.4 API Change Effects . . . 61
3.4 Discussion . . . 67
3.4.1 Comparison to Compilers . . . 67
3.4.2 Threats to Validity . . . 68
3.5 Conclusion . . . 69

CHAPTER 4 EMPIRICAL STUDY ON THE USEFULNESS OF API CHANGE RULES . . . 72
4.1 Study Design . . . 74
4.1.1 Research Questions . . . 75
4.1.2 Objects . . . 75
4.1.3 Tasks . . . 76
4.1.4 Subjects . . . 77
4.1.5 Independent Variable . . . 77
4.1.6 Dependent Variable . . . 78
4.1.7 Mitigating variables . . . 78
4.1.8 Hypotheses . . . 79
4.2 Study Execution . . . 79
4.2.1 Experiment Web Site . . . 79
4.2.2 Experiment Workspace . . . 80
4.2.3 Experiment Process . . . 80
4.2.4 Analysis Method . . . 83
4.3 Study Results . . . 83
4.3.1 Overall Data Analysis . . . 83
4.3.2 Data Analysis per System . . . 85
4.3.3 Summary . . . 92

4.4 Discussion . . . 93
4.4.1 Mitigating Variables . . . 93
4.4.2 NASA Task Load Index . . . 94
4.4.3 Change Rule Types . . . 95
4.4.4 Skeptical Subjects . . . 96
4.4.5 Experiment vs. Real Tasks . . . 98
4.4.6 Subjects vs. Professional Developers . . . 100
4.4.7 Threats to Validity . . . 100
4.5 Conclusion . . . 102

CHAPTER 5 EMPIRICAL STUDY ON FEATURE USAGES IN API CHANGE RULE BUILDING . . . 104
5.1 Study Design . . . 106
5.1.1 Feature Selection . . . 106
5.1.2 Feature Metric Computation . . . 106
5.1.3 Approach Implementations . . . 109
5.2 Study Execution . . . 113
5.2.1 Subject Programs . . . 113
5.2.2 Target Methods . . . 114
5.2.3 Evaluation . . . 115
5.3 Study Results . . . 115
5.3.1 Summary . . . 122
5.4 Discussion . . . 123
5.4.1 Recommendation List Size . . . 123
5.4.2 Change Rules without Correct Replacements . . . 124
5.4.3 Limitation . . . 125
5.4.4 Threats to Validity . . . 126
5.5 Conclusion . . . 128

CHAPTER 6 CONCLUSION . . . 129
6.1 Contributions . . . 129
6.2 Future Work . . . 130
6.2.1 Short-term . . . 131
6.2.2 Long-term . . . 132

REFERENCES . . . 134

LIST OF TABLES

Table 1.1 Framework Evolution 4
Table 1.2 Client Program Evolution 4
Table 1.3 Publication Summary 11
Table 2.1 Approach comparison 18
Table 3.1 Statistical Data of Framework Evolution and Adaptation 28
Table 3.2 Description of framework API change types - reference-type-level . . 32
Table 3.3 Description of framework API change types - method-level 33
Table 3.4 Description of framework API usage types 34
Table 3.5 Categories of classes and interfaces 42
Table 3.6 Numbers of API changes and usages 51
Table 3.7 Numbers of and average numbers of reference types in client program releases 55
Table 3.8 Numbers of reference-type-level API changes with affected methods . 64
Table 4.1 Object Systems 76
Table 4.2 Subjects' Knowledge Levels 79
Table 4.3 Null Hypotheses 80
Table 4.4 Hypothesis Testing Results for Precision 87
Table 4.5 Hypothesis Testing Results for Time 87
Table 4.6 Hypothesis Testing Results for Precision on JHotDraw 88
Table 4.7 Hypothesis Testing Results for Time on JHotDraw 89
Table 4.8 Hypothesis Testing Results for Precision on JFreeChart 89
Table 4.9 Hypothesis Testing Results for Time on JFreeChart 90
Table 4.10 Hypothesis Testing Results for Precision on JEdit 91
Table 4.11 Hypothesis Testing Results for Time on JEdit 92
Table 4.12 Precision: Two-way Permutation Test by Change Rule Type and Knowledge in Software Engineering 94
Table 4.13 Precision: Two-way Permutation Test by Change Rule Type and Knowledge in Java 94
Table 4.14 Precision: Two-way Permutation Test by Change Rule Type and Knowledge in Eclipse 95
Table 4.15 Precision: Two-way Permutation Test by Change Rule Type and Gender 95
Table 4.16 Precision: Two-way Permutation Test by Change Rule Type and Degrees 96
Table 4.17 Hypothesis Testing Results for NASA-TLX 98

Table 4.18 Task Distribution (C-Correct, I-Imperfect, N-No, JHD-JHotDraw, JFC-JFreeChart, JE-JEdit) 103
Table 5.1 Individual, Prioritised and corresponding MOOP approaches 111
Table 5.2 Subject Programs 114
Table 5.3 Numbers of change rules with correct replacements - individual feature metrics 116
Table 5.4 Numbers of change rules with correct replacements - $M5$ vs MADMatch 122
Table 5.5 Average recommendation list sizes 124

LIST OF FIGURES

Figure 3.1 Measurement computation 31
Figure 3.2 API change propagation 34
Figure 3.3 IR and IIR example 35
Figure 3.4 ACUA Modules 40
Figure 3.5 Type-level API change detection algorithm 41
Figure 3.6 Method-level API change detection algorithm 43
Figure 3.7 API usage detection algorithm 44
Figure 3.8 API usage information 45
Figure 3.9 Snippet of solr-core v4.0.0 POM file 45
Figure 3.10 Snippet of lucene-core v3.6.2-v4.0.0 API change report 47
Figure 3.11 Snippet of solr-core v3.6.2 API usage report on lucene-core v3.6.2-v4.0.0 47
Figure 3.12 Percentages of framework and client program upgraded 50
Figure 3.13 Reference-type-level API changes 52
Figure 3.14 Method-level API changes 52
Figure 3.15 Infiltration Ratios 56
Figure 3.16 API Change-Propagation Ratios 57
Figure 3.17 Ideal Infiltration Ratios 59
Figure 3.18 Project API infiltration 60
Figure 3.19 API usage recommendation 61
Figure 3.20 Upgraded with API changes framework and client program percentage 63
Figure 3.21 Percentages of framework and client program upgraded with changed and used APIs 63
Figure 3.22 Reference-type-level API changes used in client programs 64
Figure 3.23 Reference-type-level API changes with affected methods used in client programs 65
Figure 3.24 Method-level API changes used in client programs 66
Figure 3.25 Percentages of framework and client program upgraded with changed and used APIs in all 67
Figure 4.1 Web Site 81
Figure 4.2 Source Code Workspace 82
Figure 4.3 Boxplots for Precision 84
Figure 4.4 Boxplots for Time 86
Figure 4.5 NASA-TLX 96

Figure 4.6 Boxplots for NASA-TLX 97
Figure 5.1 Inheritance Tree Example 109
Figure 5.2 Effectiveness of individual features 117
Figure 5.3 Number of change rules 119
Figure 5.4 Numbers of change rules with correct replacements - non-MOOP vs MOOP 120
Figure 5.5 Average correct replacement positions - non-MOOP vs MOOP 121
Figure 5.6 Average recommendation list sizes without correct replacements . . . 126

LIST OF ACRONYMS AND ABBREVIATIONS

AAM	Add Abstract API Method to class
ACA	API Change Analyser
ACUA	API Change and Usage Auditor
AMTI	Add API Method to Interface
ANRC	Average Numbers of Releases of Client programs
ANRF	Average Numbers of Releases of Frameworks
API	Application Programming Interface
AUA	API Usage Analyser
CFP	Change Formal Parameter
CMTA	Change non-abstract Method To Abstract
CMTF	Change non-final Method To Final
CMTNS	Change static Method To Non-Static
CMTS	Change non-static Method To Static
CR	Change Ratio
CRT	Change Result Type
CSCS	Contract SuperClass Set
CSIS	Contract SuperInterface Set
CTA	Change non-abstract To Abstract
CTB	Change Type Bound
CTF	Change non-final To Final
CV	Confidence Value
DA	Decrease Access
DMA	Decrease Method Access
DR	Deprecation Ratio
EBN	Existing in Both from New
EBO	Existing in Both from Old
EON	Existing Only in New
EOO	Existing Only in Old
ESIS	Expand SuperInterface Set
ETP	Eclipse Third-party Plug-in
FUA	Framework Usage Analyser
IIR	Ideal Infiltration Ratio
IOC	Inversion Of Control

IR	Infiltration Ratio
LCS	Longest Common Subsequence
LD	Levenshtein Distance
MD	Method-level Distance
MOOP	Multi-Objective OPtimization
MSC	Missing Class
MSI	Missing Interface
MSM	Missing Method
MVC	Moved Class
MVI	Moved Interface
PUC	Percentage of the Upgraded Client programs
PUCC	Percentage of the Upgraded with Changed APIs Client programs
PUCCR	Percentage of the Upgraded with Changed APIs Client programs in Releases
PUCF	Percentage of the Upgraded with Changed APIs Frameworks
PUCFR	Percentage of the Upgraded with Changed APIs Frameworks in Releases
PUCR	Percentage of the Upgraded Client programs in Releases
PUCUC	Percentage of the Upgraded with Changed and Used APIs Client programs
PUCUCR	Percentage of the Upgraded with Changed and Used APIs Client programs in Releases
PUCUF	Percentage of the Upgraded with Changed and Used APIs Frameworks
PUCUFR	Percentage of the Upgraded with Changed and Used APIs Frameworks in Releases
PUF	Percentage of the Upgraded Frameworks
PUFR	Percentage of the Upgraded Frameworks in Releases
TC	Total Client programs
TF	Total Frameworks
UC	Upgraded Client programs
UCC	Upgraded with Changed APIs Client programs
UCDR	Used Changed and Deprecated Ratio
UCF	Upgraded with Changed APIs Frameworks
UCR	Used Changed Ratio
UCUC	Upgraded with Changed and Used APIs Client program
UCUF	Upgraded with Changed and Used APIs Frameworks

UDR	Used Deprecated Ratio
UF	Upgraded Frameworks
UR	Used Ratio
WMC	With Method-level Change

CHAPTER 1 INTRODUCTION

Frameworks [1] are widely used in modern software development (Bavota et al., 2013) and are accessed through Application Programming Interfaces (APIs), which specify sets of functionalities [2] that client programs can use to accomplish specific tasks. Frameworks keep evolving during their lifespan for various reasons (Businge et al., 2012; Hou and Yao, 2011), such as to cope with new requirements and to patch security vulnerabilities. Framework evolution may lead to API changes (Hou and Yao, 2011) to which client programs must adapt. Upgrading to new releases of frameworks is time-consuming and can even interrupt services. In 2012, Raemaekers et al. report the upgrade of the authentication framework of a software system that ended up consuming a whole week of work, even though developers were using automated tests to verify the upgraded system. In 2014, the online tax-filing service of Canada Revenue Agency was down for five days to patch the Heartbleed bug (CRA, 2014). Five days of interrupted services may cause serious losses to service providers, but five days with a service containing a known security vulnerability may expose service providers to even bigger losses. Therefore, keeping frameworks updated is important to software practitioners.

To adapt to new releases of frameworks, developers must know the API changes between the old and the new releases, read the source code and documents of the two releases of frameworks to understand how to use the new APIs, modify the source code of client programs, and test the upgraded client programs. Usually, they must iterate these steps more than once to complete the upgrading process. Considering the increasing dependency on frameworks in modern software systems (Bavota et al., 2013), upgrading several frameworks at the same time can make the process even challenging.

Consequently, helping developers upgrade frameworks to new releases draws great interests from both academic and industrial researchers. We conduct three studies on framework API evolution and propose an approach to improve existing API change rule building process. The thesis of this dissertation is:

> Following analyses of the reality of API changes and usages, of the usefulness of API change rules, and of the effectiveness of the features used to build these rules, we can build more effective and extendible API change-rule recommendation tools.

In this chapter, we first present our three studies on API changes and usages and API change rule building, then describe the organization of the dissertation.

1. Without loss of generality, we use the term "framework" to refer to both frameworks and libraries.
2. Behaviour changes of the implementations of stable APIs are beyond the scope of our study.

1.1 Framework API Changes and Usages

API change is one of the consequences of framework evolution. Developers can change APIs in different ways and for different reasons. Better understanding of how and why APIs change between framework releases from different angles helps developers evolve APIs with less impact to client programs or take precautions against API changes when they use frameworks. Des Rivières (2008) discussed in details about API contract compatibility and classified API changes according to Java programming language elements, such as package, class, method, *etc.*, but he did not investigate how API changes occur in frameworks and client programs.

How developers use framework APIs is also highly relevant to adapting framework evolution. Certain ways of using APIs may make client programs more difficult to adapt than others. Lämmel et al. (2011) analyzed API composition-style and inheritance-style API usages in more than 6,000 open-source software systems. Businge et al. (2013a) studied 512 Eclipse Third-party Plug-in (ETP) regarding internal and official Eclipse APIs usages. They did not study how the API usages are affected by API changes due to framework evolution.

Yet, there are only two works studying API changes and usages together. Robbes et al. (2012) conducted a study on how developers react to API deprecation in the Smalltalk Squeak/Pharo ecosystem. Their work is limited to a specific API change and a less popular language. Dietrich et al. (2014) investigated the differences between Java compile-time and link-time compatibility and their influences on client programs. They found that such incompatibilities widely exist, but affect client programs rarely.

Therefore, framework API changes have not been studied on a large scale. Previous studies on API usages are not from the angle of adapting client programs to API changes. There are only a few works linking API changes with usages. Yet, framework API changes and usages are two highly related phenomena. They should be studied together. It is important for developers to better understand the reality of framework API evolution and to make more reasonable decisions while evolving frameworks or building client programs.

1.1.1 Our Contributions

To investigate framework API changes and usages together at both large-scale and fine grained levels, we developed a tool, API Change and Usage Auditor (ACUA) (Wu et al., 2014a), to analyse the frameworks and their client programs in Maven repository and apply it on two framework ecosystems: Apache and Eclipse. Investigating the API changes and usages in these frameworks and their client programs helps developers understand better the reality of API evolution in common-used frameworks. In our study, we follow the definition of

des Rivières (2008) and consider public and protected Java programming language elements, such as classes and methods, as APIs. We first study framework evolution and adaptation in general, then look into different types of framework API changes and usages, to answer the following three questions.

How frameworks evolve and client programs adapt?

To study framework evolution and client program adaptation, we focus on framework APIs, through which frameworks provide their functionalities. Framework APIs have different levels, such as package, class and interface, or method. Framework fundamental functions are exposed through methods while packages and classes are the aggregations of methods to facilitate code organizing and understanding (Baxter et al., 2006). Therefore, we report the data related to API at method level, unless stated otherwise.

We adapt the methodology used by Robbes et al. (2012) to investigate framework evolution and client program adaptation at API level in 160,896 releases of 14,987 frameworks and their client programs downloaded from Maven repository by 2014 September. Robbes et al. conducted a study on how developers react to API deprecation in the Smalltalk Squeak and Pharo ecosystem. They investigated the ripple effects caused by the deprecated methods and classes from five perspectives: frequency, magnitude, duration, adaptation, and consistency. We do not analyse duration and consistency in our study, because the time stamps of framework releases are generally not available in programs downloaded from Maven repository and investigating of the adaptations of each API changes manually is not feasible at the scale of this study. Therefore, we report our results on frequency, adaptation, and magnitude.

We find that frameworks and client programs evolve with similar frequency. On average, each framework and client program in Maven repository have 11 and 12 releases, respectively. As shown in Table 1.1 and Table 1.2, most (78%) client programs adapted to new releases of frameworks at lease once, but only to less than half (42%) of frameworks. We label the 78% client programs *upgraded client programs* and the 42% frameworks *upgraded frameworks*.

Most upgraded frameworks change their APIs, but only in a small percentage of their releases. API changes exist in more than half (59%) of these frameworks and in about one quarter (24%) of their releases. Most (90%) of upgraded client programs use frameworks with API changes in more than half (60%) of client program releases.

API changes in about one third (37%) of upgraded frameworks and in about one quarter (24%) of the releases directly affect client programs. Comparing to frameworks, higher percentages of client programs and their releases are affected by API changes in frameworks.

Table 1.1 Framework Evolution

Group	#Frameworks	% In The Group Above	# Framework Releases	% In The Group Above
Total	14,987	NA	160,896	NA
With Upgraded Client Programs	6,291	42%	78,377	49%
With API Changes	3,690	59%	18,587	24%
With API Changes Affecting Client Programs	1,350	37%	4,527	24%
Affecting In Total	9%		3%	

Table 1.2 Client Program Evolution

Group	# Client Programs	% In The Group Above	# Client Program Releases	% In The Group Above
Total	12,201	NA	140,178	NA
Upgraded	9,342	78%	109,772	78%
Using Frameworks with API Changes	8,430	90%	66,066	60%
Affected By Framework API Changes	5,845	69%	29,819	45%
Affected In Total	49%		21%	

Most (69%) of client programs depending on frameworks with changed APIs directly use the changed APIs in about half (45%) of their releases.

While considering all the frameworks and client programs, instead of only those upgraded, we observe that about half of client programs are directly affected by changed APIs of small percentages of frameworks. Only 9% of all the frameworks with changed APIs in 3% of their releases are used by client programs. However, these changed APIs are used by 49% of all the client programs in about one fifth (21%) of client program releases. The magnitude of the influence of API changes is large. More than 29,000 releases of 5,845 client programs are directly affected by API changes.

How framework APIs change?

To study framework API changes, we first investigate how many APIs are changed and how many API changes are documented during framework evolution. These two facts reflect the severity of API changes and the level of help from framework developers.

We use the `@deprecated` annotation in Java to decide if an API change is documented. `@deprecated` is a way in which framework developers communicate with client program developers about API changes directly in the source code. We want to see if framework developers use this annotation often.

We find that most methods with signature changes are API methods and developers only document a small part of these API changes. The changed APIs amounts to 10% of the APIs, but 80% of the methods with signature changes are API methods. Only 2% of changed APIs are marked as deprecated. Moreover, 52% of methods marked as deprecated are not API.

Different types of API changes affect the effort to adapt to the changes. The knowledge of the distribution of API change types helps developer estimate upgrading workload more accurately. Des Rivières (2008) classified API changes into 149 types, but there is no previous study investigating the distribution of these types of API changes in frameworks.

We select 30 of des Rivières's API change types on classes, interfaces, and methods causing incompatibilities, then detect them in 22 frameworks with most-changed-and-used APIs from Apache and Eclipse framework ecosystems and their client programs, based on the study results above. We find that missing classes and methods are the most common changes. These two types of API changes require developers to spend time and effort to search for the replacements of the missing APIs in the new releases.

How client programs use framework APIs?

To study API usages, we first observe how many APIs are used by client programs. We find that client programs use only small part (16%) of the APIs provided by frameworks and 3% of the used APIs are affected by API changes. Framework developers should examine the APIs that they expose and encapsulate the APIs which are not designed to be used by client programs. Concise APIs make frameworks easier to upgrade for client program developers.

Then, we investigate three facts of API usages: which, where, and how APIs are used in client programs. With the list of used APIs and categorized API changes, developers can know what API changes in frameworks that they must adapt their client programs. Researchers can know which types of API changes affect client programs more frequently and provide remedies for such changes.

Knowing where the APIs are used in client programs, such as in which classes or methods, help developers assess the spread of APIs in their client programs. For the same API changes, client programs with more API usages are more difficult to adapt than those with less API usages, because the former have more code to change and test. Developers can make their client programs more resilient to API changes by controlling API usages and researchers can develop approaches or tools to facilitate such tasks.

We study how APIs are used with respect to change-propagation. We report five cases of API change-propagation: extending framework classes, implementing framework interfaces,

using framework reference types or their subtypes as generic types, as method return values, and as formal parameter types. The client program entities in such cases carry the changes in the used framework APIs (if any) to where these client program entities are used. A changed API used widely in client programs may cause the Shotgun Surgery code smell (Fowler et al., 1999), *i.e.*, a lot of little changes in multiple classes, which would result in high upgrade costs. In contrary, as suggested by Bloch (2008), if a client program class only uses a framework class as a private field, the API changes in the framework class only affect this client program class and will not be propagated to other places of the client program. Avoiding change-propagating framework API usages helps developers to alleviate the impact of API changes.

We find that, on average, the APIs of one framework are used in 36% of client classes and interfaces, and more than 80% of such usages could be reduced through applying certain design patterns, such as Adapter (Gamma et al., 1995). About 18% of APIs are used in change-propagating ways.

1.2 Framework API Change Rules

As shown by the results of our study on framework API changes and usages, many client programs are affected by API changes. Classes and methods disappear often between releases. Upgrading to new releases of frameworks thus requires significant effort from developers due to the changed APIs (Raemaekers et al., 2012). Developers must dig into the documents and the source code of the new and previous releases of the frameworks to understand their differences and to make their programs compatible with the new releases. The longer they delay, the more time the upgrading takes, because there will be more changes to absorb.

An effective way to help find the replacements of missing APIs is documentation. However, many frameworks are not sufficiently documented, especially when it comes to the changes between releases and the rules to adapt client programs from an older release to a new one. The Java programming language provides the `@deprecated` annotation to help framework developers mark API changes, but developers rarely use these annotations to document how to update to new releases. As examples, Google provides Android API difference reports [3] regularly but these reports only list the API removed or added. They do not include information on how to replace removed APIs. Eclipse informs developers about internal APIs and official APIs with Provisional API Guidelines [4]. By definition, internal APIs can be changed without notice and documentation. However, a survey conducted by Businge et al. (2013b)

3. http://developer.android.com/sdk/api_diff/8/changes.html
4. http://wiki.eclipse.org/Provisional_API_Guidelines#Before_the_API_freeze

showed that 70% of the developers of Eclipse plug-ins used Eclipse internal APIs and only 3.3% of them always followed Eclipse Provisional API Guidelines. Other ad hoc forms of upgrading documents include dedicated tutorials, such as the one helping Python developers to replace calls to the `os.system()` with functions provided by the `subprocess` module [5]. In general, very few companies explicitly document changes in APIs or provide such information to the public.

Therefore, many approaches (Godfrey and Zou, 2005; Kim et al., 2007; Schäfer et al., 2008; Wu et al., 2010; Kpodjedo et al., 2013) have been developed to identify API evolution by providing API *change rules*, which describe a matching between *target methods*, *i.e.*, methods existing in the old release, but not in the new one and their *replacement methods* in the new release. For example, if method t is defined in version 1.0 of a framework, but not in version 2.0, then t is a target method. If t is replaced by method m in version 2.0, m is the replacement method and $< t,\ m >$ forms a change rule.

However, the change rules generated by previous approaches are imperfect, *i.e.*, the accuracy of the change rules are not 100% and vary between frameworks. There is no previous study showing the usefulness of the imperfect change rules, *i.e.*, if they help developers find the replacement of missing APIs.

1.2.1 Our Contributions

Consequently, we conduct two studies to address the limitations of exiting work on framework API change rule building. First, we perform an experiment to investigate if the imperfect change rules generated by previous approaches really help developers find the replacements of missing API methods more accurately or faster. Second, we analyse the effectiveness of the features used by previous approaches and propose multi-objective-optimization-based approaches to improve the precision and recall of change rules building.

Do API change rules really help developers?

API change rules generated by previous approaches are imperfect or not all of them are correct. Therefore, developers do not know if the change rules are correct until having used them to modify their client programs. There is no empirical study of the usefulness of the imperfect change rules (generated by tools or from other sources) to show that the imperfect change rules help developers to identify the replacements more accurately and faster than without change rules or, rather, that they confuse developers because they are not all correct.

5. http://docs.python.org/2/library/os.html#os.system

Although we could expect that using already-known all-correct change rules would help developers, it could actually slow them down because developers would not be certain that the change rules are correct. Indeed, according to Fagard et al. (1996), providing no information is actually better than providing the wrong information: it is less confusing and distracting. Thus, knowing the usefulness of imperfect change rules could encourage and direct research on framework API evolution.

Therefore, we design and conduct an experiment to evaluate the usefulness of framework API change rules. In our experiment, the subjects find the replacements of target methods from three Java frameworks with the help of all-correct, imperfect, and no change rules. Then, we measure the performance of the subjects by the precision of the replacement methods that they find and the time that they spend.

The statistical analysis of the results shows that the precision of the replacements of target methods found by the subjects with all-correct, imperfect, and no change rules are significantly different with average values of 82%, 71%, and 57%, respectively. The effect size, Cliff's Delta (Grissom and Kim, 2005), of the difference in precision between the subjects with no and imperfect change rules is large and that between the subjects with imperfect and all-correct change rules is moderate. Different from the precision values, the times that the subjects with the three treatments spent to find the replacement methods are not statistically different with average values of 24, 23, and 25 minutes, respectively.

These results are evidence that change rules generated by framework API evolution approaches are useful, even when some of the change rules are incorrect. Yet, as expected, the higher precision the change rules have, the more help they provide. Thus, imperfect change rules can be used instead of unavailable documentation or as complement to partial documentation. Developers of frameworks could also use them as starting point to build upgrading documentation. Yet, researchers should improve the accuracy of API change rule building approaches to provide better help to developers.

How to improve API change rule building?

The results of our exploratory study and experiment show that missing classes and methods are the most frequent API changes affecting client programs and that imperfect change rules generated by previous approaches do help developers identify the replacements of missing APIs. Yet, the level of usefulness of the change rules corresponds to their accuracy. The more accurate the change rules are, the more they help developers. Thus, we conducted another empirical study to explore how effective the features used by previous approaches are, and how we improve the accuracy of the generated change rules.

Previous approaches, such as (Dagenais and Robillard, 2011; Kim et al., 2007; Schäfer et al., 2008; Wu et al., 2010), automatically identify API change rules between an old and a new release of a framework using the similarity values between some features of a target method t and a replacement candidate m. For a given t, these approaches use similarity values to sort the methods in the new releases of frameworks and suggest change rules to developers by recommending those at the top as potential replacements. These approaches use multiple features to detect change rules by prioritizing features. They implement prioritization in two ways. One explicitly gives high priority to certain features. For example, AURA (Wu et al., 2010) combines call-dependency and signature similarity by giving higher priority to the former over the latter. Another assigns weights to features, such as the approaches of Kim et al. (2005).

For a given framework, if an API method and its replacement are not similar with respect to one feature, approaches using this feature cannot detect a correct change rule. For example, some replacement methods may be appropriate when considering only call dependency but not when considering signature similarity. Also, a specific framework may require favouring one feature over the others, *i.e.*, an approach using that feature would detect more correct change rules than those using the others.

To detect more correct change rules, considering more features is promising but not straightforward, because multiple features may give contradictory information, confusing prioritizing approaches. An incorrect change rule suggested by a high-priority (heavy-weight) feature cannot be overridden by lower-priority (light-weight) features. Besides, these approaches are difficult to extend to new features, because their developers must choose the priorities, parameters, and thresholds of the new features with respect to the other features (Kim et al., 2005; Kpodjedo et al., 2013).

Although there are many approaches to detect API changes using different features, the effectiveness of individual feature and their combinations have not been fully investigated. Such studies would help researchers devise more accurate and extendible approaches to build API change rules.

In particular, we use multi-objective-optimization techniques to handle possibly contradictory information given by multiple features while identifying framework API evolution. Multi-objective optimization (Sawaragi et al., 1985) is the process of finding solutions to problems with potentially conflicting objectives. No previous work uses multi-objective-optimization techniques to combine features.

We conduct a study to compare approaches using different features in different ways, including multi-objective-optimization-based and prioritizing techniques, to build change rules. The

goal of our study is two-fold: (1) identify features that are really beneficial to build API change rules and (2) find the combining process that obtains the best accuracy from the features.

We find that not all features are useful individually. Signature similarity is more effective than the other features. Not all of the multi-feature approaches outperform single feature approaches and the results of the multi-objective-optimization-based approaches are more stable and accurate than that of the corresponding prioritising approaches. We conclude that multi-objective optimization is an effective way to combine multiple features to build change rules, especially when there is no prior knowledge to favour certain features.

1.3 Organization of the Dissertation

The organization of this dissertation is presented below and the links between the publications, the corresponding sections and my contributions are summarised in Table 1.3.

Chapter 2 introduces related work on API changes and usages, API change rule building, program differentiation, empirical studies in software engineering, and the applications of multi-objective optimisation in software engineering.

Chapter 3 presents an exploratory study on API changes and usages in Maven repository and two framework ecosystems: Apache and Eclipse. The work presented in this chapter is based on two papers:

SCAM-2014 "ACUA: API change and usage auditor" published as tool paper in the proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM2014).

EMSE-UR1 "An Exploratory Study of API Changes and Usages in Apache and Eclipse Ecosystems" under review in the Journal of Empirical Software Engineering (EMSE).

Chapter 4 presents an empirical study to investigate the usefulness of API change rules to help developers identify replacements of missing APIs. The work presented in this chapter is based on our paper:

EMSE-2014 "The impact of imperfect change rules on framework API evolution identification: an empirical study" published in the Journal of Empirical Software Engineering (EMSE), July, 2014.

Table 1.3 Publication Summary

Publication	Corresponding Section	Contribution
SCAM-2014	Section 3.2.1	Designing, implementing the tool, participating in writing the paper
EMSE-UR1	Parts of Section 3.1, 3.2, and 3.3	Implementing the approach and conducting the experiment, participating in experiment design, analysing data and writing the paper
EMSE-2014	Section 4	Analysing experimental data and writing the paper, participating in experiment design
EMSE-UR2	Section 5	Implementing the approach and conducting the experiment, participating in experiment design, analysing data and writing the paper
ICSE-2010	Parts of Section 5.1 and 5.2	Iimplementing the approach and comparing the results with other approaches, participating in approach design and writing the paper

Chapter 5 presents an empirical study on the feature usage in API change rule building. The work presented in this chapter is based on our papers:

EMSE-UR2 "Feature Usages in Framework API Evolution Identification" under review in the Journal of Empirical Software Engineering (EMSE).

ICSE-2010 "AURA: a hybrid approach to identify framework evolution" published in the proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, May, 2010.

Chapter 6 concludes the dissertation by summarizing our findings and discussing future work.

CHAPTER 2 LITERATURE REVIEW

Following our thesis, we present the related work on: (1) API changes and usages, (2) API change rule building and program differentiation, (3) empirical studies in software engineering, and (4) the applications of multi-objective optimization techniques in software engineering. We describe the related work of the latter two domains, because we conduct an empirical study on the usefulness of API change rules and apply multi-objective-optimization techniques to improve API change rule recommendation.

2.1 Framework API Changes and Usages

2.1.1 API Changes

Robbes et al. (2012)

Robbes et al. (2012) conducted a study on how developers react to API deprecation in the Smalltalk Squeak/Pharo ecosystem. We adapt their methodology to investigate framework evolution and client program adaptation at program level and method. In our study, we investigate the API changes in Java programs hosted in Maven central repository, including those not marked as deprecated.

The Squeak/Pharo ecosystem has more than 3,000 contributors and more than 2,600 programs. They analysed 577 and 186 deprecated methods and classes, respectively. They investigated the ripple effects caused by the deprecated methods and classes from five perspectives: frequency, magnitude, duration, adaptation, and consistency.

For frequency, they discovered that 14% of the deprecated methods and 7% of the deprecated classes caused ripple effects at least in one project. The percentages are low for two reasons. First, most of the deprecated methods and classes were only used internally, *i.e.*, in the projects where they are defined. Second, the client programs were not aware of the deprecation. In our study, besides deprecated methods, we also investigate the frequency of API changes and usages from different angles.

For magnitude, they found that the median of affected projects is 25. Because affected project number depends on the size of dataset, the percentage of affected projects reflects better the magnitude of API changes than the absolute number. Therefore, we use percentage to measure as magnitude instead of concrete number. We investigate the magnitude of API evolution at program and method level, and the magnitude of different API change types.

For duration, they studied it from two angles: reaction time and adaptation time. The former is the time between when the deprecation is introduced and when the client programs started being modified to adapt. The latter is the duration that the client programs took to finish all the adaptations. The median of reaction time was two weeks, but the median of adaptation time was 0 day, which means half of the adaptations were completed in the same day when it started. However, the third quartile of the distribution of the adaptation was 26 days. We do not analyse duration, because the time stamps of framework releases are generally not available in programs downloaded from Maven repository.

For adaptation, they found that the median of the projects reacted to the API deprecation is 5 and the median of involved developers is 7. Only about 20% of affected projects reacted to the deprecated APIs and more than one developer involved in the upgrading process. Similar to magnitude, we study the adaptation at program level instead of at method level, because the latter requires more qualitative analysis and are possible future work.

For consistency, the median of the percentage of the projects adapting the deprecated APIs in the same way was 60%. Some client program developers did not follow the recommendation given by API owners to upgrade their code. We do not analyse consistency, because investigating of the adaptations of each API change is not feasible at the scale of this study and should be conducted manually.

Others

Rivières (2008) summarized 149 API changes, based on entities changed in APIs, such as packages, classes, modifiers. He also pointed out which API changes may cause binary incompatibility (*i.e.*, the class files of the new releases of frameworks cannot be linked to client programs without recompilation) or source incompatibility (*i.e.*, there are errors when the client program source code is compiled with the new releases of frameworks). However, des Rivières did not investigate how the API change types occur in frameworks and client programs. Consequently, we want to conduct such a study to analyse API changes in real systems based on his classification. We start from Apache and Eclipse framework ecosystems because of their popularity.

Hou and Yao (2011) classified API changes in AWT and Swing according to domains and design intention. They summarized eight intentions to change APIs in these two JDK packages, such as conformance to naming conventions, simplification, introduce of new concepts, *etc.* They found that the proportion of changed APIs is small. The changes are mainly for correction or clarification, but feature redesign is an important cause of API changes.

Cossette and Walker (2012) investigated the binary incompatibility between 16 release pairs of three Java systems (Struts, Log4j, JDOM) manually. According to how easy to adapt API changes, they classified API changes into three categories: fully automatable, partially automatable, and hard to automate changes. They reported the purposes of API changes from a different angle of Hou and Yao (2011), such as exposing internal implementation, generalizing functionalities, replaced with external APIs, *etc.* They also reported that single recommendation techniques only applies to parts of the changed APIs.

Dietrich et al. (2014) investigated the differences between Java compile-time and link-time compatibility and their influences on the client programs. They analysed 564 releases of 109 programs from the QUALITAS corpus (Tempero et al., 2010), excluding those from Eclipse and Azureus. They found that 75% of the upgrades have link-time incompatibilities and these incompatibilities are also compile-time incompatible, except two cases. They also discovered that such incompatibilities only affect 8 client programs.

2.1.2 API Usages

Lämmel et al. (2011)

Lämmel et al. (2011) conducted a large-scale AST-based analysis of framework-style and library-style API usages in the Java projects hosted in SourceForge [1] Repository. This is the first study which analysed API usages at large-scale from framework-style and library-style points of views. Our exploratory study of API changes and usages is inspired by their work.

Framework-style and library-style API usages have different impacts on the client programs when these APIs change. In framework-style usage, client programs must extend classes or implement interfaces from frameworks which requires developers have better knowledge of framework internal implementation and propagates API changes (Bloch, 2008). Library-style usages do not require inheritance and relatively easy to be encapsulated locally. It is important for developers to know the proportion between the two types of usage to estimate the cost to upgrade to new releases of frameworks.

They analysed Java projects managed by Ant [2] form the SourceForge hosted SVN repository. To guarantee the accuracy of the analysis, their main fact extract technique was based on building the AST from the source code of the Java projects. AST-based analyses require that the projects must be buildable. Lämmel et al. downloaded more than 6,000 Java projects and made 1,476 buildable manually. There are 69 used frameworks in their dataset.

1. http://sourceforge.net/
2. http://ant.apache.org/

Then, they selected 60 projects marked as "mature" or "stable" by SourceForge and with more than 100 commits from the 1,476 *built projects*. They named the 60 projects as *reference projects*. They report API usage analysis results according to the built projects and the reference projects. Besides AST-based factor extract, they also used token-based techniques to gather information, such as `import` statements in the Java source code. They analysed API usages from three perspectives:

First, they investigated API footprint at framework and API method level. At framework level, they found that the built projects use 4.7 and the reference projects use 6.9 frameworks per project, on average. At method level, the built projects use 370 and the reference projects use 866 API methods per project, on average. The API footprint growths with project sizes.

Second, they investigated API coverage, *i.e.*, the percentage of API methods used in client programs in the total API methods defined in frameworks. They reported the API coverage of two XML parsing frameworks, JDOM [3] and SAX [4]. The former is used typically in library-style and the latter is in framework-style. They found that only 24% of API methods in JDOM and 49% of API methods in SAX are used by their client programs, respectively.

Third, they investigated how many frameworks are actually used in framework-style. Frameworks usually provide both interfaces for client programs to implement and implementations for concrete functions. Client program developers can choose either of them according to their goals. Lämmel et al. find that only about half (35) of frameworks are used in framework-style in their dataset.

Our exploratory study of API changes and usages extends their work. First, we parse Java class files instead of source code to avoid the making-buildable effort, so our analysis can be applied to larger data set. Second, we analysed Maven [5] projects instead of Ant projects. Maven projects have specific places to download framework and client program Jar files. Thus, we can fully automate fact extracting process. Third, Maven projects have version information of the depended frameworks in the configuration files, so we can detect the API changes between the used and later releases of frameworks to study API changes and usages together. Fourth, we investigate API footprint in more details. We study API usages from infiltration and propagation perspectives. We also distinguish two subtypes of framework-style usages, *i.e.*, inheritance for Inversion Of Control (IOC) and optional inheritance.

3. http://www.jdom.org/
4. http://sax.sourceforge.net/
5. http://maven.apache.org/

Others

Roover et al. (2013) conducted a detailed multi-dimensional API usage analysis on QUALITAS corpus (Tempero et al., 2010). They solved the dependencies of the projects in QUALITAS corpus and added metadata generated by their analysis. The metadata include API names (equivalent to framework names), domains, and facets (groups of sub-functions of frameworks). They named the extended QUALITAS corpus for API usage analysis QUAATLAS (QUALITAS API Atlas). They explored API usages in QUAATLAS from different perspectives and presented them in a set of insights. Each insight is composed of a list of attributes, such as intent, stakeholder, view, *etc.* Insights can be project-centric (from the point of view of client programs) or API-centric (from the point of view of frameworks). They also proposed an interactive approach and developed a Web-enabled tool Exapus to help developers explore API usages regarding these insights.

The in-depth analysis and the interactive GUI tool in Roover et al. (2013) help developers understand API usages from various angles, while our study links API usages with API changes and focuses on how to identify and contain optional API usages in order to reduce framework upgrading effort.

Other researchers studied API usages from different points of view. SpotWeb (Thummalapenta and Xie, 2008) distinguishes frequently-used and rarely-used APIs by mining open-source software repositories. Kawrykow and Robillard (2009) proposed an approach to detect the cases where client program developers imitate the functions provided by libraries instead of using corresponding APIs. LibSync (Nguyen et al., 2010) helps developers learn complex API usage change patterns from the clients that have been already upgraded. Portfolio (McMillan et al., 2011) searches and visualizes relevant functions and their usages from a database. Lämmel et al. (2011) analysed the framework API usages regarding potential and actual reuse in Microsoft .Net programs with static and dynamic analyses. Businge et al. (2013a) studied 512 ETPs regarding internal and official Eclipse APIs usages. They found that 44% of these ETPs use internal Eclipse APIs and ETP developers continue to use Eclipse internal APIs. APIMiner (Montandon et al., 2013) extracts API usage examples using program slicing technique. Web API is a new type of APIs emerged with Internet. Espinha et al. (2014) studied the evolution of four popular Web APIs and their impacts on clients.

2.2 API Change Rule Building and Program Differentiation

Previous approaches help developers evolve their programs when the used frameworks evolve. Some of them capture API-level changes and require the framework developers to manually enter the change rules or to use a particular IDE to automatically record changes (Chow and Notkin, 1996; Dig et al., 2007; Henkel and Diwan, 2005; Kemper and Overbeck, 2005). The limitation of such approaches is that framework developers may not be available.

Other approaches take different software artefact as inputs and use API features to build change rules between framework releases. The inputs and features used by these approaches are summarized in Table 2.1. Each feature may be represented by different metrics in a specific approach. For example, *Code Lexical* can be computed using the Longest Common Subsequence (LCS) (Gusfield, 1997) or Levenshtein Distance (LD) (Levenshtein, 1966) between two method signatures.

These approaches take two types of inputs. SemDiff (Dagenais and Robillard, 2011), the approach of Kim et al. (2005), and HiMa (Meng et al., 2012) use software repository commits. They are not applicable to the programs whose version control system repositories are not available. Other approaches (Godfrey and Zou, 2005; Kim et al., 2007; Kpodjedo et al., 2013; Schäfer et al., 2008; Wu et al., 2010; Xing and Stroulia, 2007a) take the source code of two releases of a framework as input. Among them, the approach of Schäfer et al. (2008) also uses the client code as a part of its input. In our study, we consider only source code because it is the most common input.

To build change rules, previous approaches used the similarities of some features, such as call dependency or signature similarity, to sort the methods in the new releases of frameworks and recommend those at the top as potential replacements for target methods. Besides these approaches in Table 2.1, Cossette and Walker (2012) also reported that developers often use comments in source code to find the replacements of missing methods. Therefore, we also include source code comments in this study even only UMLDiff uses it.

Among previous approaches, SemDiff (Dagenais and Robillard, 2011) and the approach of Kim et al. (2007) used single feature: call-dependency and method signature similarities, respectively. Approaches using a single feature cannot identify replacement methods that are not similar to the target methods, according to the feature metric. Also, they may recommend different replacement methods at each execution. If there are many methods with the same highest value of similarity in the new release, the replacement might not be recommended in an execution.

The other approaches combine multiple features to improve the precision and recall (Cohen,

Table 2.1 Approach comparison

Approach	Input	Features						
		Call-dependency	Code Lexical	Inheritance	Software Metric	Structural	SVN Commit Comments	Source Code Comments
SemDiff (Dagenais and Robillard, 2011)	SVN Commits	X						
Beagle (Godfrey and Zou, 2005)	Source Code	X	X		X			
Kim et al. (2005)	CVS Commits	X	X		X			
Kim et al. (2007)	Source Code		X					
MADMatch (Kpodjedo et al., 2013)	Source Code	X	X	X		X		
HiMa (Meng et al., 2012)	SVN Commits	X	X				X	
Schäfer et al. (2008)	Source Code*	X	X	X				
AURA (Wu et al., 2010)	Source Code	X	X					
UMLDiff (Xing and Stroulia, 2007a)	Source Code		X	X		X		X

* Including client program source code

1988), *i.e.*, to detect change rules more accurately and to detect more correct change rules. The common technique of the approaches to use multiple features is to prioritise the features. Some of them give the features different weights (Kim et al., 2005; Kpodjedo et al., 2013). The most appropriate weight values are program specific and it is difficult to know them in advance. In the approach of Kim et al. (2005), they chose the weighing schema according to the oracle set built by 10 human judges. The weighting schemata are completely different for the two programs that they analysed. Other approaches prioritised the features in their algorithms. They first use some features to detect change rules, then use the other features to refine them or detect more. For example, AURA (Wu et al., 2010) and the approach of Schäfer et al. (2008) give call-dependency similarity a higher priority than signature similarity. They first detect change rules with call-dependency similarity, then use signature similarity to improve the precision of the change rules built by call-dependency similarity.

Besides which features to use, the accuracy of previous approaches depends on how to prioritise the features and on specific programs as well. The contradictory information that features may mislead prioritising approaches. For example, `FileCellRenderer.getTree CellRendererComponent(...)` in jEdit v4.1 does not exist in v4.2. Call-dependency similarity recommends `VFSDirectoryEntryTable.propertiesChanged()` as replacements, because this method is called in the same context as the missing method. However, according to signature similarity, the replacement method should be `FileCellRenderer.getTableCell RendererComponent(...)`, which is the correct replacement. If we give call-dependency higher priority, the incorrect change rule suggested by call-dependency similarity cannot be overridden by the signature similarity and compromises the accuracy.

Therefore, to consider more features, developers must decide where to put the new features, which priority to give, what parameters to set, *etc.* Making such decisions is not straightforward and program-dependent, because of the potential conflicts brought by the new features. Prioritizing approaches are difficult to generalize and extend because of such difficulty.

Furthermore, prioritising approaches are complicated to setup due to their algorithm using thresholds and parameters. For example, MADMatch (Kpodjedo et al., 2013) is a state-of-the-art approach in program differentiation. MADMatch uses a novel cost model to match software elements between releases. However, it has eight basic cost parameters and five aggregate cost parameters. Its performance depends on their values. Consequently, we want to compare with MADMatch to see if there are less complicated approaches with similar or better accuracy.

2.3 Empirical Study on Program Comprehension

Another field related to this work is the empirical studies on program comprehension. Lawrie et al. (2007) conducted an empirical study to investigate the influence of three types of identifiers on program comprehension. They compared the number of correct answers of developers after reading programs using single letter, abbreviation, and full-word identifiers. They found that developers understand better the programs with full-word identifiers. However, there is no statistically significant difference between abbreviation and full-word identifiers in many cases and using descriptive abbreviation identifiers could reduce developers' memory burden.

Yusuf et al. (2007) conducted an experiment to study how developers use the different information in UML class diagrams with eye-tracking system. They found that experts usually navigate from the center to the edges of the diagrams while the patterns of beginners are top-to-bottom or left-to-right. Also, experts use the different information, such as stereotype, color, and layout, in the diagrams to help navigation.

Porras and Guéhéneuc (2010) evaluated the impact of the presentations on design pattern comprehension with an eye-tracking system. They compared pattern-enhanced class diagrams, stereotype-enhanced UML diagrams, and pattern-role notation with standard UML presentation. They used three metrics based on the fixation time collected by the eye-tracking system to measure developers' effort. They found that stereotype-enhanced UML diagrams are better in identifying the design patterns that a class participates and pattern-enhanced class diagrams are more efficient in identifying the classes involving a design pattern.

Sharif and Maletic (2010) and Sharafi et al. (2012) use eye-tracking systems to compare camel-case and underscore identifier styles. The former focuses on accuracy and speed when developers recognize identifiers while the latter studies identifier comprehension from the point of view of genders. Sharif and Maletic (2010) found that there is no difference between underscore and camel-case identifiers in terms of accuracy, but developers recognize underscore identifiers faster. Sharafi et al. (2012) found that there is no statistically significant difference in the final results between male and female developers, but they use different comprehension strategies. Females spent more time to examine wrong answers than males.

Abbes et al. (2011) performed an experiment to evaluate the impact of two anti-patterns (Blob and Spaghetti Code) on program comprehension. They measured the performance of subjects using NASA task load index (Hart and Staveland, 1988), the time that they spent, and the percentages of their correct answers. They found that combining of two anti-patterns compromises program comprehension of developers while only one anti-pattern does not have significant effect on developers.

Soh et al. (2012) performed an experiment to assess the role of professional status and expertise on UML class diagram comprehension. They also took the preciseness of the description of the tasks into account. They found that students spent less time than professionals and professionals gave more precise answers than students. Expertise is more important than professional status regarding the accuracy and speed of the subjects. If the descriptions of the tasks are precise, novice students can have comparable performance as the other subjects.

Ali et al. (2012) used eye-tracking system to rank developers' preferred source code entities, such as class names, method names, during requirement-code tracing tasks. They found that developers try to understand source code mainly using method names and comments and developers pay more attention to the source code entities which reflects domain concepts. Based on their discoveries, they proposed two new schemes, SE/IDF and DOI/IDF, to improve the weighting systems in requirement tracing approaches.

Empirical studies help us evaluate different methods or techniques (Wohlin et al., 1999). There is no previous work evaluating the usefulness of API change rules generated by previous approaches. We want to conduct an experiment to fill the gap.

2.4 Multi-objective Optimization in Software Engineering

Many problems in software engineering have to deal with potential conflicts. For example, the Next Release Problem (NRP) has two potential conflicting objectives: the value that requirements bring and the cost to implement these requirements. Implementing more requirements in the new release of a product may cause the increase of total cost. Controlling cost by excluding requirements may decrease the satisfaction level of customers. Because multi-objective optimization techniques can solve problems with conflicting objectives, Zhang et al. (2007) presented an approach which uses Non-dominated Sorting Genetic Algorithm II (NSGA-II) (Deb et al., 2002), a multi-objective optimization algorithm to propose solutions for NRP. They compare NSGA-II with Weighted Single-Objective GA and Pareto GA. They found that NSGA-II outperformed Pareto GA and Weighted Single-Objective GA is helpful to find the extreme solutions.

Saliu and Ruhe (2007) also attacked NRP with multi-objective optimization. They took the dependencies between the components implementing requirements into account. The dependencies can reduce the cost of implementing features with common components. By adding such features into the same release, their approach could propose solutions with better customer satisfaction level than those which do not consider component dependencies.

Finkelstein et al. (2009) used multi-objective optimization to solve NRP from the concern of fairness. The issue of fairness in NRP is caused by that customers may give a same requirement different values. Including a requirement is fair to one customer and might be unjust to another. They created three formulations on fairness of requirement allocation: the number of included requirements, the value of included requirements, and the percentages of value and cost of included requirements. They evaluated their formulations with two real world datasets and a synthetic dataset and two multi-objective optimization algorithms. The results showed that their approach outperforms random search.

Gueorguiev et al. (2009) applied multi-objective optimization to project robustness problem, *i.e.*, how to make project plans resilient to unpredictable delays. They formulated this problem with three objectives: Completion Time, Completion Time with New Tasks, and Completion Time with Delayed Tasks. They applied a multi-objective optimization algorithm SPEA II (Zitzler and Thiele, 1999) and evaluated their solutions on four industrial projects from three companies. They confirmed that their multi-objective optimization-based approach outperforms random search.

Two previous works provided detailed information about multi-objective optimization in software engineering. Panerati and Beltrame (2014) compared 15 multi-objective optimization algorithms on multiprocessor system-on-chip design-space exploration task. Harman et al. (2012) surveyed the literature on Search-Base Software Engineering, including multi-objective optimization applications. Hitherto, no previous work applied multi-objective optimization to solve the potential conflicts brought by different features used in API change rule building. Therefore, we want to investigate the applicability of multi-objective optimization techniques to this problem.

2.5 Summary

Few previous works studied API changes and usages together. No previous approaches studied API changes on a large scale. Studies on both API changes and usages can be conducted at more fine-grained level. There is no empirical study to show the usefulness of the API change rules generated by previous approaches, *i.e.*, if the change rules can help developers to identify the replacement methods more accurately or faster than without them. Despite many approaches to build API change rules, the effectiveness of the features used by these approaches has not been investigated. Also, there is still room to improve the accuracy of previous approaches building API change rules. Therefore, we conduct three studies presented in this dissertation to prove our thesis:

> Following analyses of the reality of API changes and usages, of the usefulness of API change rules, and of the effectiveness of the features used to build these rules, we can build more effective and extendible API change-rule recommendation tools.

CHAPTER 3 EXPLORATORY STUDY ON API CHANGES AND USAGES

The first study in this dissertation is a large-scale and fine-grained analysis of the reality of framework API changes and usages. Object-oriented frameworks are widely used in software systems today (Raemaekers et al., 2012), because they reduce development time and increase the user-perceived quality of programs through the reuse of existing code implementations that are considered more reliable and stable (Gamma et al., 1995). Yet, as other software artefacts, frameworks evolve for various reasons (Hou and Yao, 2011), such as to cope with new requirements or patch security vulnerabilities.

Because framework upgrades are necessary but costly (Raemaekers et al., 2012; CRA, 2014), developers must understand better the evolution of frameworks and consider it during development process. They should assess and forecast the cost of each framework upgrade based on the time and the scope of the upgrades. Frameworks are used through their Application Programming Interfaces (APIs), which specify a set of functionalities that client programs can use. Differences in API changes and usages affect upgrade costs. For example, developers can adapt to a class that moved from one package to another more easily than to the removal of a class. To adapt to the latter change, developers must find a replacement of the removed class or re-implement it, while for the former, they can simply update the package of the class. Developers need tools to collect facts about framework API changes and usages in their client programs as the basis of preparing for API changes and upgrading cost estimation.

Also, researchers must understand better the API changes and usages at a fine-grained level, so that they can discover the more important aspects of API evolution and develop more efficient solutions accordingly. For example, previous approaches (Dagenais and Robillard, 2008; Kim et al., 2007; Meng et al., 2012; Schäfer et al., 2008; Wu et al., 2010) build change rules to help developers find replacements of missing APIs methods. Only if most of the missing API methods are caused by missing classes or methods, are the change rules built by such approaches useful.

However, previous approaches did not study API changes on a large scale. Des Rivières (2008) discussed API contract compatibility in details and classified API changes according to Java programming language elements, such as package, class, method, *etc.*, but he did not investigate how the API changes occur in frameworks and client programs. Hou and Yao (2011) classified API changes in AWT and Swing according to domains and design intention. Cossette and Walker (2012) investigated the binary incompatibility between several releases of three Java systems manually, using a different classification from des Rivières. These

approaches did not investigate how these API changes affect client programs.

For API usages, most of previous approaches did not link API usages with potential changes. SpotWeb (Thummalapenta and Xie, 2008) distinguishes frequently-used and rarely-used APIs by mining open-source software repositories. Kawrykow and Robillard (2009) proposed an approach to detect the cases where client program developers imitate the functions provided by libraries instead of using corresponding APIs. Lämmel et al. (2011) analysed API composition-style and inheritance-style API usages in more than 6,000 open-source software systems. Businge et al. (2013a) studied 512 Eclipse Third-Party Plug-ins (ETPs) regarding internal and official APIs usages. Roover et al. (2013) proposed Exapus, an interactive approach and developed a Web-enabled tool, to explore API usages from different views.

Moreover, few previous work studied API changes and usages together. Robbes et al. (2012) conducted a study on how developers react to API deprecation in the Smalltalk Squeak/Pharo ecosystem. Their work is limited to a specific API change and a less popular language. Dietrich et al. (2014) investigated the differences between Java compile-time and link-time compatibility and their influences on the client programs. They found that such incompatibilities widely exist, but affect client programs rarely.

Therefore, we investigate the reality of API changes and usages together at large-scale and fine-grained level to answer the following research questions:

— RQ1: How do framework APIs evolve?
— RQ2: How do client programs use framework APIs?
— RQ3: How do framework API changes affect client programs?

To conduct our study, we developed a tool, ACUA (Wu et al., 2014a), to extract facts about API changes and usages from frameworks and client programs. In our study, we follow des Rivières's definition and consider public and protected Java programming language elements, such as classes and methods, as APIs. With ACUA, we first adapt the methodology used in Robbes et al. (2012) to investigate API evolution in general, then detect the types of API changes and usages on classes, interfaces, and methods, because they are the most important program entities (Ali et al., 2012).

We use ACUA to mine API changes and usages in 160,896 releases of 14,987 frameworks and their client programs downloaded from Maven repository by 2014 September. Maven is a project management tool from the Apache Software Foundation [1]. Maven repository hosts projects with the information about the dependencies between client programs and frameworks in their configuration files.

1. http://maven.apache.org/

To investigating the API changes and usages at fine-grained level, based on the results of large-scale API change and usage analysis on Maven repository, we focus on the top 11 framework releases with most changed APIs used by client programs from Apache and Eclipse ecosystems. The frameworks provided by these two ecosystems are widely used in open-source software development (IBM, 2006; van Zyl and Apache, 2005) and have evolved during the last two decades with structured dependency and version information between client programs and frameworks.

Also, the two ecosystems use different strategies to manage API evolution. Eclipse informs developers about internal APIs and official APIs with Provisional API Guidelines [2]. By definition, internal APIs can be changed without notice and documentation. Apache does not have explicit regulations on API changes. We want to investigate if there is any difference between the frameworks from the two ecosystems with respect to API changes and usages.

ACUA categorises API changes according to des Rivières' classifications (Rivières, 2008) and reports three facts of API usages: which, where, and how APIs are used in client programs. With list of used APIs and categorized change APIs, developers can see to what changes they must adapt their client programs. Researchers can prioritise different types of API changes according to their frequency in frameworks.

Knowing where the APIs are used in client programs, such as in which classes or methods, developers can know the spread of API usage in client programs. For the same API changes, client programs with more API usages are more difficult to adapt to API changes than those with less API usages, because the former may have more code to change. Developers can make their client programs more resilient to API changes by controlling API usages and researchers can develop approaches or tools to facilitate such tasks.

ACUA reports how APIs are used with respect to change-propagation and encapsulation with composition (Bloch, 2008). ACUA distinguishes five cases of API change-propagation: extending framework classes, implementing framework interfaces, using framework reference types or their subtypes as generic types, as method return values, and as formal parameter types. The client program entities in such cases propagate the changes in the used framework APIs (if any) to where these client program entities are used. In contrary, if a client program class only uses a framework class as a private field, the API changes in the framework class only affect this class and will not be propagated to other places of the client program. Avoiding using framework APIs in change-propagating ways can also help developers to alleviate the impact of API changes. Supporting approaches and tools are desirable for developers as well.

2. http://wiki.eclipse.org/Provisional_API_Guidelines

We analyse the 22 frameworks from Apache and Eclipse with ACUA. We find that missing classes and methods happen more often in frameworks and affect client programs more often, also missing interfaces affect client programs more often than they happen in frameworks. These phenomena confirm the usefulness of existing API change rule building approaches. On average, the APIs of one framework are used in 36% of client classes and interfaces, and more than 80% of such usages could be reduced through applying certain design patterns, such as Adapter (Gamma et al., 1995). About 20% and 15% of APIs in Apache and Eclipse frameworks, respectively, are used in change-propagating ways. Avoiding these usages, such as preferring composition over inheritance (Bloch, 2008), can alleviate the impact of framework API changes.

3.1 Study Design

We adapt the methodology used in Robbes et al. (2012). Robbes et al. conducted a study on how developers react to API deprecation in the Smalltalk Squeak and Pharo ecosystem. They investigated the ripple effects caused by the deprecated methods and classes from five perspectives: frequency, magnitude, duration, adaptation, and consistency. We do not analyse duration and consistency in our study, because the time stamps of framework releases are generally not available in programs downloaded from Maven repository and investigation of the adaptations of each API change manually is not feasible at the scale of this study. Therefore, we report our results on frequency, adaptation, and magnitude. In this section, we present background information about API evolution in general, APIs changes and usages types, including the definitions on which we base our empirical study.

3.1.1 API Evolution Overview

To measure framework evolution from the perspectives of frequency, adaptation, and magnitude at program level, we first collect the statistical data listed in Table 3.1. Then, we compute the corresponding metrics below. In Table 3.1, each column represents a subset of programs in our data set:

Total Frameworks (TF): frameworks, *i.e.*, all the programs in our data set.

Total Client programs (TC): client programs, *i.e.*, all the programs depending on other programs.

Upgraded Frameworks (UF): the frameworks which have at least two releases used by a client program.

Upgraded Client programs (UC): the client programs which use at least two releases of a given framework.

Table 3.1 Statistical Data of Framework Evolution and Adaptation

		Total	Upgraded	Upgraded with Changed APIs	Upgraded with Changed and Used APIs
Framework	Number	F#	UF#	UCF#	UCUF#
	Release Number	FR#	UFR#	UCFR#	UCUFR#
Client Program	Number	C#	UC#	UCC#	UCUC#
	Release Number	CR#	UCR#	UCCR#	UCUCR#

Upgraded with Changed APIs Frameworks (UCF): the upgraded frameworks which have API changes between their releases.

Upgraded with Changed APIs Client programs (UCC): the upgraded client programs which use the framework releases with API changes.

Upgraded with Changed and Used APIs Frameworks (UCUF): the upgraded frameworks which have changed APIs used by the upgraded client programs.

Upgraded with Changed and Used APIs Client program (UCUC): the upgraded client programs which use the changed APIs of frameworks.

Frequency

We use the Average Numbers of Releases, $ANRF$ and $ANRC$, to reflect the frequency of framework and client program evolution, respectively.

$$ANRF = \frac{FR\#}{TF\#} \qquad ANRC = \frac{CR\#}{TC\#} \tag{3.1}$$

Adaptation

We use four metrics to measure the adaptation of framework evolution. For frameworks, we compute PUF, the Percentage of the Upgraded Frameworks, and the same percentage counted in the release number, $PUFR$, where the postfix R represents releases.

$$PUF = \frac{UF\#}{TF\#} \qquad PUFR = \frac{UFR\#}{FR\#} \tag{3.2}$$

Similarly, we compute PUC, the Percentage of Upgraded Client programs, and the same percentage counted in the release number, $PUCR$.

$$PUC = \frac{UC\#}{TC\#} \qquad PUCR = \frac{UCR\#}{CR\#} \tag{3.3}$$

Magnitude

We report the magnitude at two levels. First, we check how many upgraded frameworks and client programs with changed APIs, then we look closer to see how many of them with changed and used APIs. To measure the magnitude of API changes in frameworks and client programs, we used the Percentages of the Upgraded with Changed APIs Frameworks (Client programs) in the upgraded frameworks (client programs):

$$PUCF = \frac{UCF\#}{UF\#} \qquad PUCFR = \frac{UCFR\#}{UFR\#} \tag{3.4}$$

$$PUCC = \frac{UCC\#}{UC\#} \qquad PUCCR = \frac{UCCR\#}{UCR\#} \tag{3.5}$$

In a similar way, we used the Percentages of Upgraded with Changed and Used APIs Frameworks (Client programs) in the upgraded with changed and used APIs frameworks (Client programs) to measure the magnitude of API changes in frameworks affecting client programs:

$$PUCUF = \frac{UCUF\#}{UCF\#} \qquad PUCUFR = \frac{UCUFR\#}{UCFR\#} \tag{3.6}$$

$$PUCUC = \frac{UCUC\#}{UCC\#} \qquad PUCUCR = \frac{UCUCR\#}{UCCR\#} \tag{3.7}$$

Summary

Using the 14 metrics defined above, we investigate the frequency, adaptation and magnitude of API evolution in the frameworks and client programs. These data present the overview of the framework API evolution and client program adaptation at program level.

The frequency is represented by ANRF and ANRC, the average release numbers of framework and client programs.

The adaptation of client programs is measured by PUC, the percentage of client programs using at least two releases of a framework in all the client programs. For frameworks, the adaptation is PUF, the percentage of the frameworks used by the upgraded client programs in total programs. PUCR and PUFR are the same percentages as PUC and PUF, but computed in release numbers instead of program numbers.

The magnitude is described at two levels. First, we investigate PUCC, the percentage of client programs using the frameworks with API changes in the next releases, and PUCF, the percentage of the frameworks used by such client programs. The subset of client programs depend on the frameworks with API changes, but the changed APIs in the frameworks may

not be directly used by the client programs. Second, we look closer and compute PUCUC, the percentage of client programs using changed APIs in frameworks directly, and PUCUF, the percentage of frameworks with changed and used APIs. The corresponding metrics with the postfix R are computed in release numbers.

3.1.2 API Changes

In this section, we present the metrics to measure API change frequency from two angles.

API Changes At Method Level

First, we study the frequency of API changes at method level. We measure how many APIs are changed and how many API changes are documented during framework evolution. These two facts can reflect the severity of API changes and the level of help from framework developers. We use the `@deprecated` annotation in Java to decide if an API change is documented. We consider the APIs marked with this annotation as documented API changes. We define two metrics to measure these two facts: Change Ratio (CR), the percentage of changed APIs in total APIs, and Deprecation Ratio (DR), the percentage of deprecated APIs in total APIs.

$$CR = \frac{Changed\ API\#}{Total\ API\#} \qquad DR = \frac{Deprecated\ API\#}{Total\ API\#} \tag{3.8}$$

API Changes Classification

Furthermore, we want to examine API change frequency from another angle. Rivières (2008) summarized 149 API changes, based on entities changed in APIs, such as packages, classes, modifiers. He also pointed out which API changes may cause binary incompatibility (*i.e.*, the class files of the new releases of frameworks cannot be linked to client programs without recompilation) or source incompatibility (*i.e.*, there are errors when the client program source code is compiled with the new releases of frameworks).

In this paper, we do not distinguish between these two types of incompatibilities because, as stated by Buchholz (2008), *"Every change is an incompatible change"* and risk/benefit analyses are required for all of them. Hence, client developers should be informed of all API changes and we start from those causing both incompatibilities.

In the classification of API changes proposed by des Rivières, we selected those on classes, interfaces and methods, also causing incompatibilities, because they are the fundamental entities of object-oriented programming languages and the incompatibilities break client pro-

grams. We reorganized des Rivières' API changes into 23 categories according to their effects on the client programs. For example, we split *Delete API type from API package* to *Missing API Type* and *Moved API Type* because any modern IDE can help developers locate the replacements of the latter easily, but not those of the former.

Among our API change types, 15 are at reference type (classes and interfaces) level (shown in Table 3.2) and eight are at method-level (shown in Table 3.3). These types of API changes do not have the same effect on client programs. For example, adapting to the addition of a new `int` parameter to an API method is different than adapting to the removal of an API method, because for the latter, developers must find a replacement of the removed method or re-implement it. Knowing the distribution of the types of API changes is important for preparation of API changes in development process and accurate estimations of programs upgrade workloads.

Therefore, we use P_T to measure the frequency of an API change type T between the two releases of a framework. We first classify the API changes according to the 23 types described above. Then, we count $\#T$, the number of changed APIs under each type T and $\#ALL$, the total number of changed APIs. Next, we compute P_T, the percentage of the changed APIs belonging to API change type T following Equation (3.9).

$$P_T = \frac{\#T}{\#ALL} \tag{3.9}$$

Figure 3.1 shows an example of P_T calculation. In this example, there is one MSC (Missing Class) and two WMC (With Method-level Changes). Therefore, the frequency of API change types MSC and WMC are $P_{MSC} = 33\%$ and $P_{WMC} = 66\%$, respectively.

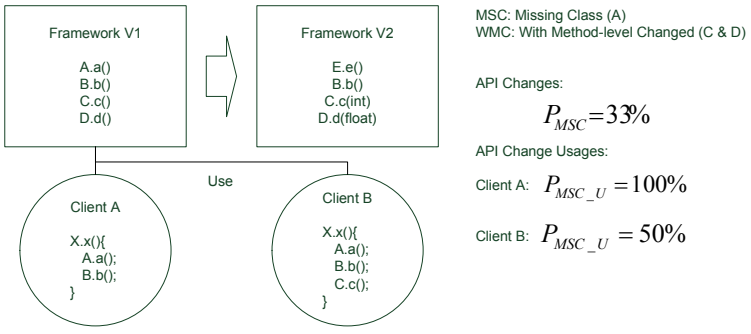


Figure 3.1 Measurement computation

Table 3.2 Description of framework API change types - reference-type-level

ACUA	Des Rivières
MSC (Missing Class)	Delete API type from API package
MSI (Missing Interface)	
MVC (Moved Class)	
MVI (Moved Interface)	
CSIS	Contract SuperInterface Set (direct or inherited)
ESIS	Expand SuperInterface Set (direct or inherited)
CTB (Change Type Bound)	Add, delete, or change type bounds of type parameter
	Add type parameter
	Delete type parameter
	Re-order type parameters
WMC (With Method-level Change)	Add API method
	Delete API method
	Move API method up type hierarchy
	Move API method down type hierarchy
	All method-level changes
CTK	Change Type Kind of APIs: (class, interface, enum, or annotation type)
DA	Decrease Access: change public type in API package to make non-public
AAM	Add Abstract API Method to class
CTF	Change non-final To Final
CTA	Change non-abstract To Abstract
CSCS	Contract SuperClass Set (direct or inherited)
AMTI	Add API Method to Interface

Summary

We investigate API change frequency from two angles. First, we investigate CR, the percentage of the changed APIs in total APIs, and DP, the percentage of the APIs whose changes are documented during framework evolution. These two facts can reflect the severity of API changes and the level of help that the framework developers provide to support client program upgrading. Second, we adapt the API change classification of Rivières (2008) to compute P_T, the distributions of different types of API changes. The distributions help client program developers and researchers better understand how framework developers evolve APIs.

3.1.3 API Usages

Analogously, we introduce the metrics for API usage frequency in this section.

API Usages at Method Level

Similar to API changes, we first investigate API usage frequency at method level. We compute two metrics Used Ratio (UR) and Used Deprecated Ratio (UDR) to reflect how widespread

Table 3.3 Description of framework API change types - method-level

ACUA	Des Rivières
MSM (Missing Method)	Change method name
	Delete API Method
	Move API method up type hierarchy
	Move API method down type hierarchy
CFP (Change Formal Parameter)	Add or delete formal parameter Change type of a formal parameter
CMTNS	Change static Method To Non-Static
CMTS	Change non-static Method To Static
CMTF	Change non-final Method To Final
CMTA	Change non-abstract Method To Abstract
CRT	Change Result Type (including void)
DMA	Decrease Method Access: from public access to protected, default, or private access from protected access to default or private access

API usage is in client programs.

$$UR = \frac{Used\ API\#}{Total\ API\#} \qquad UDR = \frac{Used\ \&\ Deprecated\ API\#}{Total\ API\#} \tag{3.10}$$

API Usages Classification

Also, different types of APIs usages have an impact on the process of adapting client programs to new releases of frameworks and can be investigated form various perspectives (Thummalapenta and Xie, 2008; Kawrykow and Robillard, 2009; Lämmel et al., 2011; Businge et al., 2013a; Roover et al., 2013). We study the API usage types regarding API change-propagation. As shown in Table 3.4, type $U1$ and $U2$, *i.e.*, class extensions and interface implementations are two inheritance-style usages that require the understanding of the internal implementation of frameworks and propagate framework APIs (Bloch, 2008). Indeed, sub classes and classes implementing interfaces also expose the APIs of the super classes or interfaces defined in the frameworks. If those APIs change, the affected sub classes and interface implementations will carry the changes to where they are used in the client programs. However, it is not always possible to eliminate API inheritance-style usages completely. Frameworks are designed for the purpose of inversion of control (IOC) (Gamma et al., 1995), *i.e.*, client programs become a part of frameworks by overriding or implementing methods in the classes or interfaces provided by the frameworks. However, if a sub class extending framework classes or

Table 3.4 Description of framework API usage types

Type	Framework API Usage	Change-propagating
U1	Inheritance for IOC	Yes
U2	Optional inheritance	Yes
U3	As generic types	Yes
U4	As method return types	Yes
U5	As method formal parameter types	Yes
U6	With method implementations	No

implementing framework interfaces does not override any methods invoked inside the framework, it is not for IOC and probably is an *optional-inheritance*, that can be replaced with a *composition* (Bloch, 2008), to encapsulate framework APIs with client program classes, thus to reduce the impact of the APIs changes. An example of composition-style usage is the use of framework reference types as private fields in client classes while only accessing them within method implementations.

Besides inheritance-style usages, there is also the possibility to use framework classes or interface types or their subtypes in change-propagating ways, as $U3 - U5$ in Table 3.4. If client programs use framework APIs as generic types, method return types, or formal parameter types, client programs still propagate the API changes within the framework classes or interfaces used, even they do not inherit these framework classes or interfaces. As illustrated in Figure 3.2, in Client A, the developer extends framework class `A` to create class `X` and use `X` as parameter in other methods. If there is an API change in `A`, every place using `X` must be modified. In Client B, `A` is used in a composition-style. When `A` changes, only the wrapper `X` is affected. Thus, avoiding API change-propagation can help developers adapt to API changes while upgrading frameworks. They only need change the code that directly uses framework APIs.

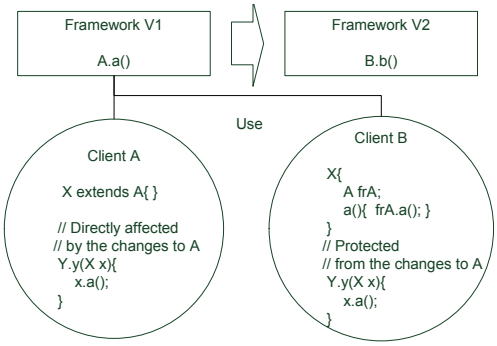


Figure 3.2 API change propagation

To capture how widespread API usage is in client programs, we first define the Infiltration Ratio (IR) metric. IR reflects the percentage of client program entities that use the framework APIs directly. The lower is the value of IR, the easier it will be for developers to adapt API changes. In Equation (3.11), $RefTypes$ stands for reference types (*e.g.,* classes and interfaces).

$$IR = \frac{\#RefTypes\ Using\ APIs}{\#Total\ RefTypes} \tag{3.11}$$

Then, based on our API usage type classification, we also define the Ideal Infiltration Ratio (IIR) metric to compute the lowest IR value that a client program can reach. Here, $\#RefTypes\ For\ IOC$ is computed using the number of reference types in client programs that override framework methods called inside frameworks. An IOC (*i.e.,* inversion of control) reference types must use framework APIs through inheritance. Other types of API usages can be encapsulated with small numbers of local classes. The practical lowest IR value should be slightly larger than IIR, because we must consider encapsulating classes.

$$IIR = \frac{\#RefTypes\ For\ IOC}{\#Total\ RefTypes} \tag{3.12}$$

Figure 3.3 shows an example of API usage by a client program. In this example, `FC` is a framework class and `FI` is a framework interface of IOC, because its method `fr_mi()` is invoked inside the framework. The client program has three classes: `C1` extends `FC`, `C2` implements `FI`, and `C3` does not use framework APIs. Among them, only `C2` is the class connecting with the framework through IOC. Therefore, IR and IIR values for this client program are 67% and 33%, respectively.

As shown in Table 3.4, Type $U1 - U5$ are change-propagating ways to use framework APIs.

```
// Framework
public class FC {
        public void fr_mc(FI frI){ frI.fr_mi(); } }

public interface FI{
        public void fr_mi(); }

// Client
public class C1 extends FC{ ... }

public class C2 implements FI{
        public void fr_mi(){ ... } }

public class C3 { ... }
```

Figure 3.3 IR and IIR example

$U1$ is required if client programs want to integrate with frameworks. $U2 - U5$ propagate framework APIs, when the API changes, because not only they are affected when the used APIs change, but also they propagate the impact to where these client program elements are used. Also, some cases of $U6$ may be caused by $U2 - U5$. Therefore, reducing the cases of $U2 - U5$ is a way to decrease IR. Consequently, we define ACPR (API Change-Propagation Ratio) to measure the severity level of API change-propagation. Lower value of ACPR represents less API change-propagation.

$$ACPR = \frac{\#RefTypes\ With\ U2 - U5}{\#Total\ RefTypes} \tag{3.13}$$

Summary

We study API usage frequency at method level first, then investigate it from the point of views of API change-propagation. We report them in UR, the percentage of used APIs in client programs, and UDR, the percentage of used and deprecated APIs, to describe API usages at method level. We also report IR, the percentage of the classes and interfaces in client programs to measure the wideness of API usage. Based on our API usage classification, we compute IIR, the lowest IR that a client program can reach without changing the interaction with a used framework, and ACPR, the percentage of APIs used in a way that carries their changes to where they are used. UR and UDR give us a reference point of API usage study. IR, IIR, and ACPR provide basis when developers estimate the cost to upgrade the client programs to new releases of frameworks.

3.1.4 API Change Effects

After investigating API changes and usages individually, we study further to see the magnitude of changed APIs affecting client programs.

API Change Effects at Method Level

To describe API change effects at method level, we define two metrics: Used Changed Ratio (UCR), the percentage of changed APIs in used APIs by client programs, and Used, Changed, and Deprecated Ratio (UCDR), the percentage of APIs used in client programs, marked as deprecated in one release, and not existing in the next releases of frameworks.

$$UCR = \frac{Used\ \&\ Changed\ API\#}{Total\ API\#} \tag{3.14}$$

$$UCDR = \frac{Used\ \&\ Changed\ \&\ Deprecated\ API\#}{Total\ API\#} \tag{3.15}$$

API Change Type Effects

To observe the effects of each type of API change T on a client program, we use the metric P_{T_U} to measure the proportion of APIs that experienced a change of type T and are used by client programs, in the total number of changed APIs used by client programs. We compute P_{T_U} following Equation (3.16).

$$P_{T_U} = \frac{\#T_U}{\#ALL_U} \tag{3.16}$$

where $\#T_U$ is the number of changed APIs of Type T used by client programs and $\#ALL_U$ is the number of all changed APIs used by client programs.

Figure 3.1 shows an example of P_{T_U} calculation. In this example, client `A` only uses one changed API affected by NEC and client `B` uses another change API affected by WCM additionally. Thus, P_{MSC_U} is 100% for `A` and P_{MSC_U} is 50% for `B`. Developer and researchers can know which types of API changes affect client programs more often with P_{T_U}.

Summary

Besides API change and usage frequency, we study the magnitude of changed APIs affecting client programs. We measure the magnitude at method level with UCR, the percentage of changed APIs used in client programs, and UCDR, the percentage of changed and deprecated APIs used in client programs. To measure the magnitude of the effects of API change types, we define P_{T_U}, the percentage of the APIs under a change type T used in client programs.

3.1.5 Research Questions

With the set of metrics defined above, we can provide more detailed information to answer three high-level research questions:

— RQ1: How do framework APIs evolve?
— RQ2: How do client programs use framework APIs?
— RQ3: How do framework API changes affect client programs?

Concretely, we can use these metrics to answer the following detailed research questions:

— RQ1.1: How often frameworks and client programs evolve? ($ANRF$ and $ANRC$)
— RQ1.2: How many client programs adapt to framework evolution? (PUF and PUC)
— RQ1.3: How many APIs change during framework evolution? (CR)
— RQ1.4: How many APIs are marked as deprecated during framework evolution? (DR)
— RQ1.5: How often does each type of API changes happen in frameworks? (P_T)

— RQ2.1: How many APIs are used by client programs? (UR)
— RQ2.2: How many used APIs are marked as deprecated? (UDR)
— RQ2.3: How widely do APIs infiltrate in the client programs? (IR)
— RQ2.4: How widely does API change-propagation exist in client programs? ($ACPP$)
— RQ2.5: How many API usages in client programs can be encapsulated? (IIR)

— RQ3.1: How many client programs adapt to frameworks with API changes? ($PUCF$, $PUCC$, $PUCUF$, and $PUCUC$,)
— RQ3.2: How many changed APIs are used by client programs? (UCR)
— RQ3.3: How often is each type of API changes used in client programs? (P_{T_U})

In the next section, we present how to conduct our study.

3.2 Study Execution

To present our study execution, we first introduce the tool, ACUA, to collect API changes and usages to compute the metrics defined above. After, we describe the dataset to analyse.

3.2.1 Tooling

At the beginning, we describe the modules of ACUA in brief, then provide more detailed information about its API change and usage detection algorithms. In the end, we demonstrate the work flow, and the inputs and the outputs of ACUA with a running example. The functional modules of ACUA are shown in Figure 3.4. The elements with gray background are the modules and those with white background are the inputs and outputs.

Inputs

Maven Projects For frameworks and their client programs managed as Maven projects, ACUA takes the Maven POM (Project Object Model) configuration files of two releases of frameworks as inputs, because ACUA analyses API changes between the two releases. Maven

is a project management tool from the Apache Software Foundation [3]. Maven projects store the information about the dependencies between client programs and frameworks in POM files. The advantage of using Maven POM files is that Maven is widely used in software development and provides the complete information required by ACUA, such as dependencies and Jar file locations.

Other projects We convert other projects with dependency information, such as Eclipse plug-ins, to Maven projects, then analysed by ACUA. For example, the dependency information of Eclipse plugins are managed in two ways. From Eclipse 1.x to 3.0, the required plug-ins are specified under the `requires` node in the `plugin.xml` file contained in the plug-in folders. From Eclipse 3.1 to 4.x, the plug-in dependencies are configured in the `Require-Bundle` or `Import-Package` sections of the `MANIFEST.MF` files of the Jar files. We use the two formats of dependency information in Eclipse plug-ins, converts them into POM files and install the generated POM files and corresponding plug-in Jars into the Maven repository, so that Eclipse plug-ins can be processed uniformly as Maven projects. For non-Maven programs whose dependencies do not always have version information, like projects managed by Apache Ant, developers must complete the version information.

Modules

Framework Usage Analyser (FUA) uses the information in POM files to detect the changes in framework versions. FUA also requires a connection to Maven repository to download the Jar files of the corresponding versions of the frameworks and their client programs. FUA interacts with Maven repository through Aether library (Bentmann and Zyl, 2012).

Next, Model Builder parses the Jar files of the frameworks and the client programs to build the models containing reference types, method definitions, call and inheritance relations of framework and client program releases, according to our meta-model. Model builder uses the ASM Java bytecode analysis framework Bruneton et al. (2000) to extract the model data from the Jar files. We analyse Jar files because they contain fully-qualified names of Java entities, such as classes and methods. We save the effort to resolve the entity names, which is required to analyse Java source code.

Taking the models as input, API Change Analyser (ACA) detects the API changes between the two releases of frameworks and classifies them into the types presented in Section 3.1.2. ACA outputs the API changes as API change reports of the two releases of each framework.

3. http://maven.apache.org/

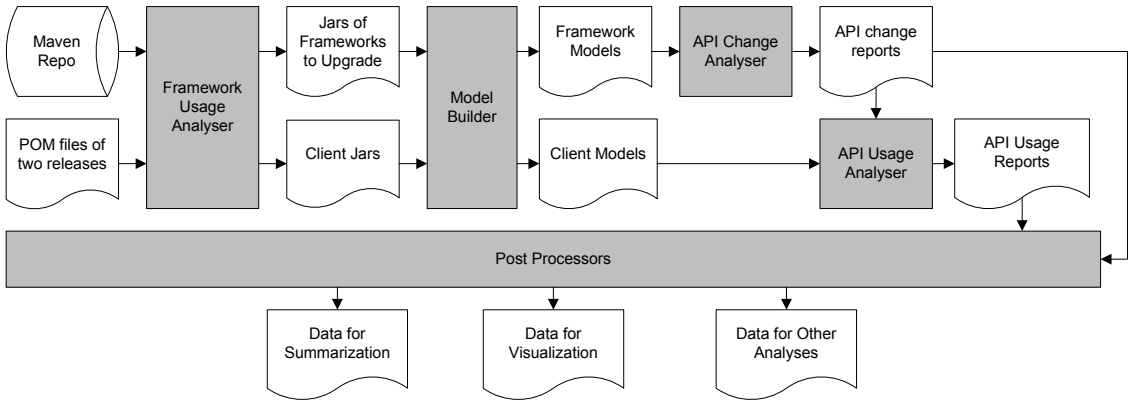


Figure 3.4 ACUA Modules

Based on the API change reports and the client models, API Usage Analyser (AUA) detects which, where, and how the APIs of the frameworks used in the client programs, then verifies if the used APIs are changed in the new releases of frameworks and are affected by which types of API changes. As the output, AUA generates API usage reports for each framework including how the used APIs are affected by API changes.

API change analysis ACUA detects API changes between two releases of a framework at reference type (classes and interfaces) level and method-level in ACA. To analyse reference-type-level API changes, ACA first classifies the reference types of two releases of a framework into four categories as shown in Table 3.5: EBO (Existing in Both releases from the Old), EBN (Existing in Both releases from the New), EOO (Existing Only in the Old release), and EON (Existing Only in the New release). The names of classes and interfaces contain their package names and reference type names, without considering type kind (class or interface), modifiers, and generic type parameters. So, an interface in EBO may become a class in EBN with the same name. Also, a class with the same name in EBO and EBN may have differences in their methods because of behaviour changes.

Then, ACA checks if the classes and interfaces in EOO have counterparts in the EON with the same type names, but different package names. Those having such counterparts are classified as Moved types (classes, interfaces) and the rest are classified as Missing types.

Next, ACA checks the classes and interfaces in EBO and EBN to detect the types of the rest API changes. Figure 3.5 is the flowchart of the reference-type-level API change type detection algorithm.

For classes and interfaces in EBO, ACA detects method-level API changes as follows. First,

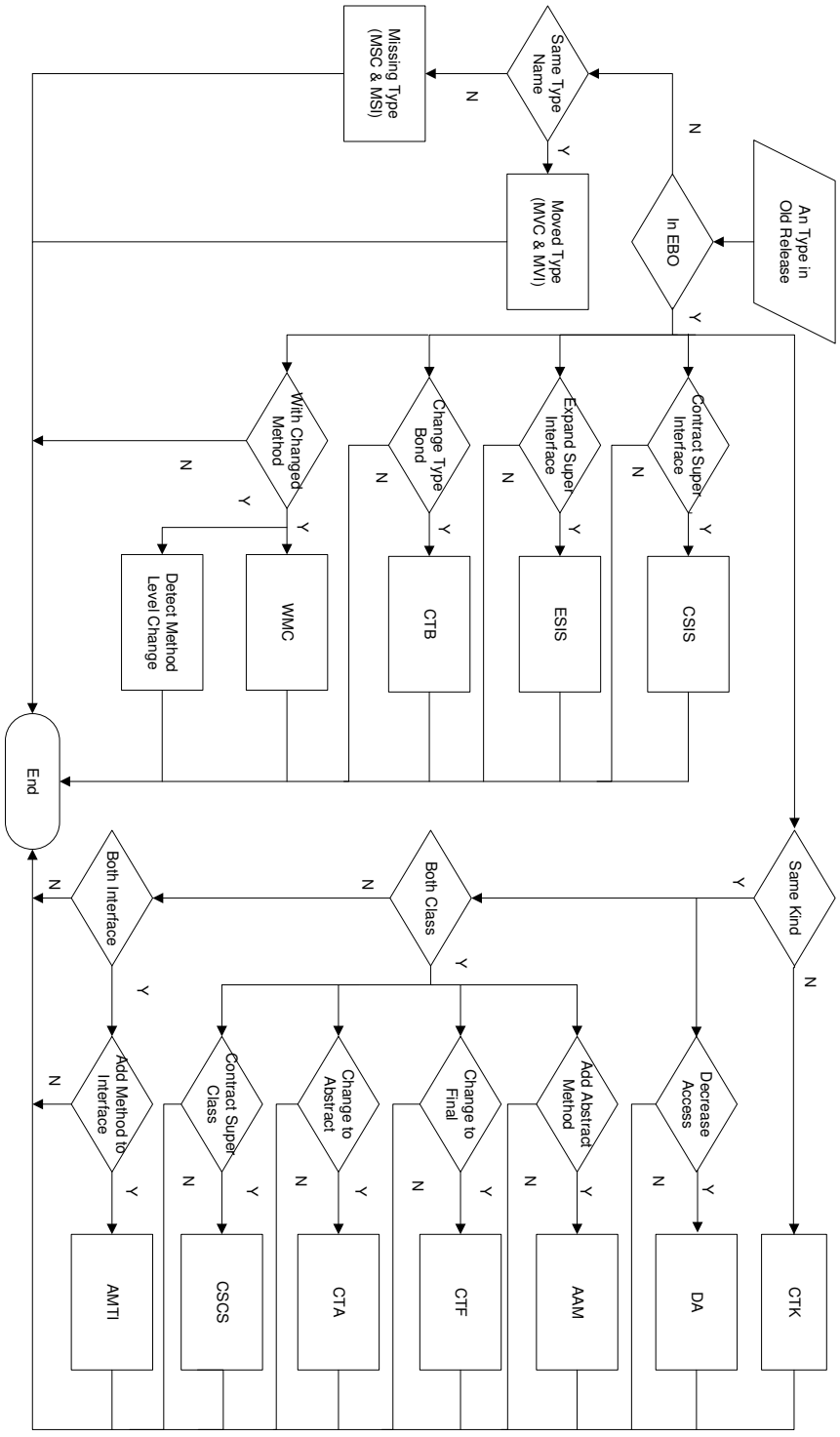


Figure 3.5 Type-level API change detection algorithm

Table 3.5 Categories of classes and interfaces

Releases	Classes and interfaces	
	With same names in both releases	With names only in one release
Old	EBO (Existing in Both from Old)	EOO (Existing Only in Old)
New	EBN (Existing in Both from New)	EON (Existing Only in New)

ACA checks if there is another method with the same name in EBN. If there is not, ACA classifies the method as a Missing Method. If there is one, ACA checks if the other method-level API changes happened to the method. The flowchart of the method-level detection algorithm is shown in Figure 3.6.

A changed API can belong to more than one API change types. For example, added field to interface and added method to interface can happen to the same interface.

API Usage Analysis AUA checks which and how APIs are used by analysing API type inheritances and API method invocations in client programs. It reports the usage types $U1 - U6$ in Table 3.4. The detection algorithm is shown in Figure 3.7. AUA collects which APIs are used, if they are used for inversion of control, where they are used and the types of API changes by which they are influenced. The results of API usage detection are stored in a data structure shown in Figure 3.8.

Running Example We describe the working flow of ACUA using solr-core [4] version 3.6.2 as a running example. Solr is a open source search platform based on Apache Lucene [5]. One framework used by solr-core v3.6.2 is lucene-core v3.6.2 and the Solr development team wants to upgrade to lucene-core v4.0.0 in the next release (assuming it is also v4.0.0). They use ACUA to collect the information about the API changes between lucene-core v3.6.2 and v4.0.0 and how the changes used in Solr v3.6.2, because they need to estimate the upgrading work load and plan upgrading task accordingly.

Solr developers first create a POM file for v4.0.0 in which the version of lucene-core is changed to v4.0.0 according to the upgrading goal. A snippet of the POM file is shown in Figure 3.9. In POM file, The releases of solr-core and lucene-core are represented by `<groupId>`, `<artifactId>`, and `<version>`. The release information of lucene-core is under `<dependency>` node and that of solr-core is at the root level.

Then, Solr developers run ACUA with the two POM files of Solr, v3.6.2 and v4.0.0, respec-

4. http://lucene.apache.org/solr/
5. http://lucene.apache.org/

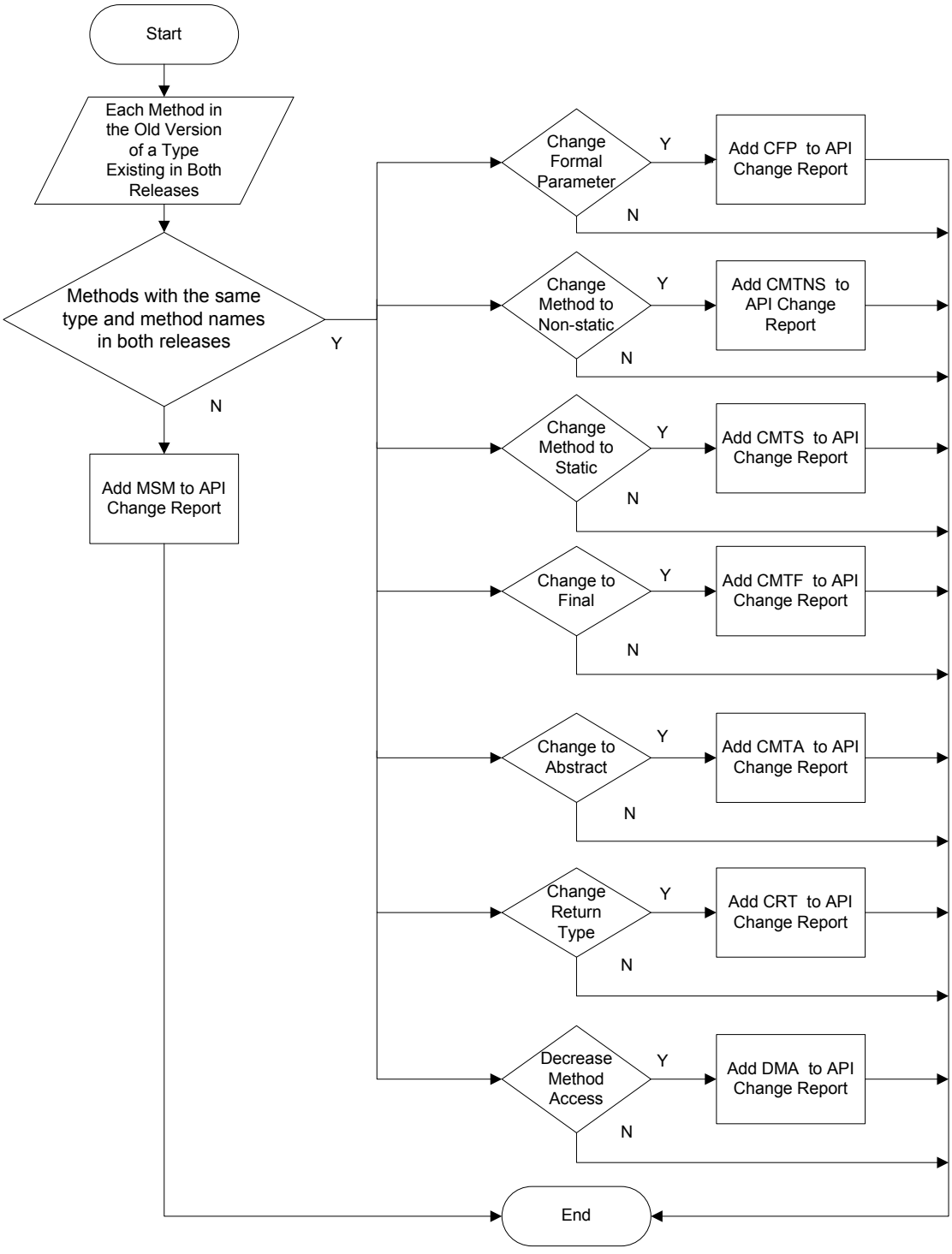


Figure 3.6 Method-level API change detection algorithm

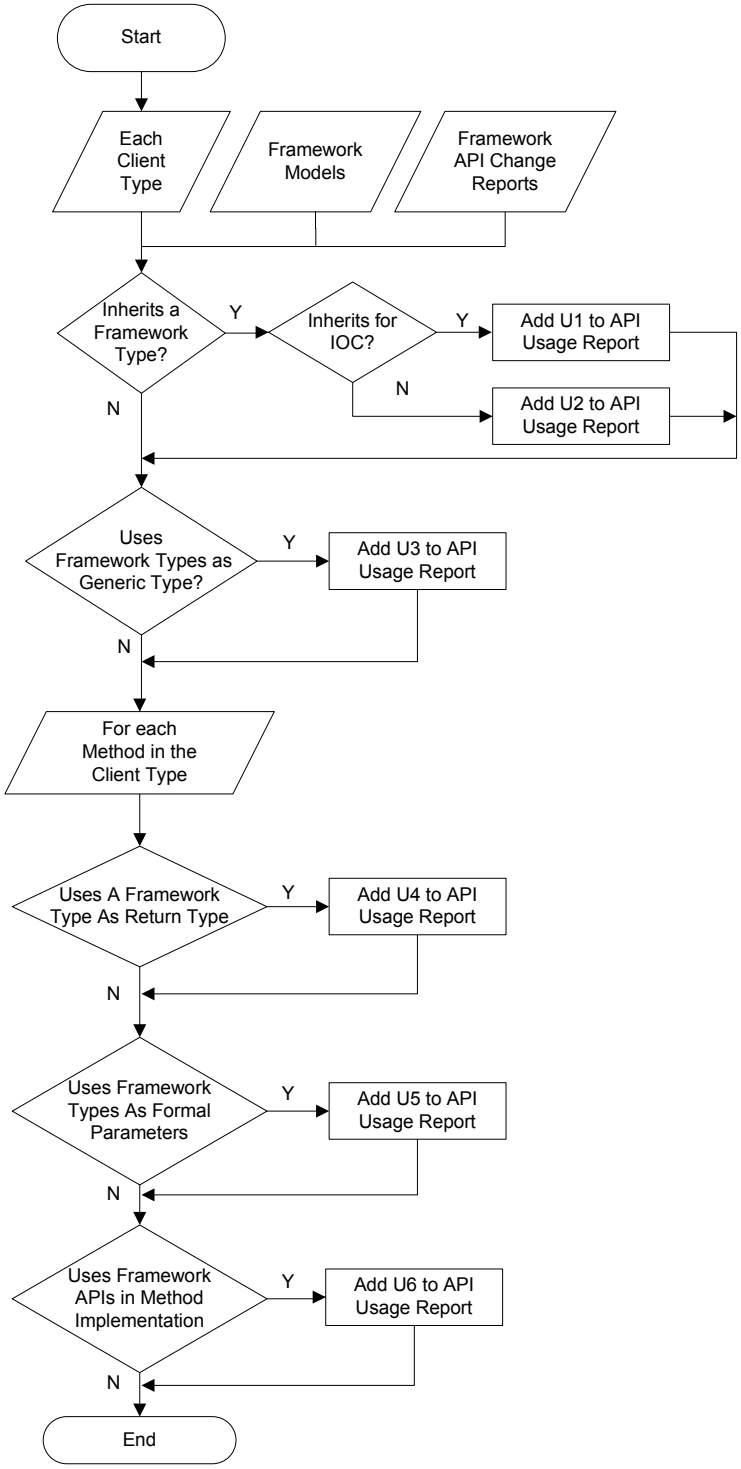


Figure 3.7 API usage detection algorithm

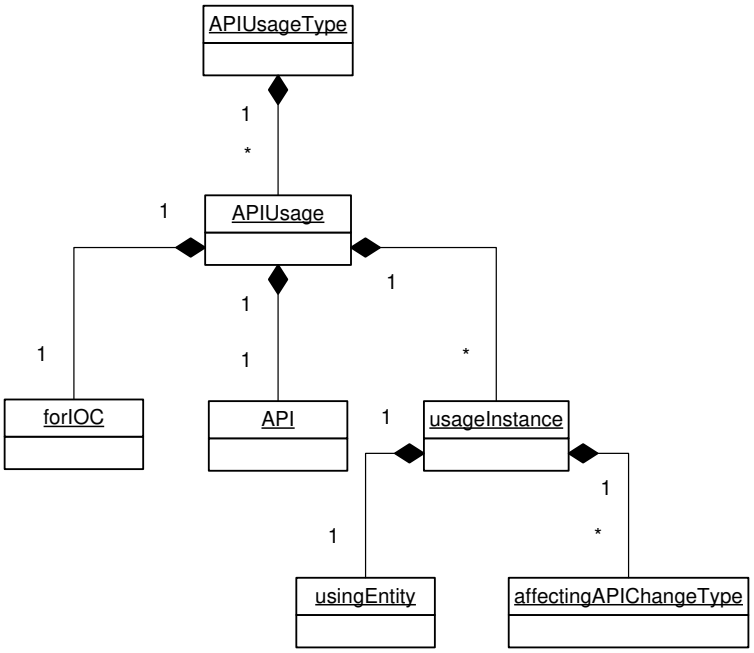


Figure 3.8 API usage information

```
...
<groupId>org.apache.solr</groupId>
<artifactId>solr-core</artifactId>
<version>4.0.0</version>
...
<dependencies>
        <dependency>
                <groupId>org.apache.lucene</groupId>
                <artifactId>lucene-core</artifactId>
                <version>4.0.0</version>
        </dependency>
...
```

Figure 3.9 Snippet of solr-core v4.0.0 POM file

tively. ACUA detects the version changes in lucene-core (v3.6.2 to v4.0.0), then it analyses the Jar files of the two releases of lucene-core and solr-core v3.6.2 to generate API change and API usage reports for them. To facilitate post processing, API change and API usage reports generated by ACUA are in XML format. Figure 3.10 is a part of API change report for lucene-core v3.6.2 to v4.0.0. In the report, the information of two releases of frameworks, the types of API changes, the levels and the changed APIs are stored in specific nodes. Developers can search, analyse them according to the changed APIs.

The API usage report lists the information of the release of the client program and of the two releases of the framework to upgrade, the used APIs, the types of the usages and the change types affecting the APIs in the new release of the framework, as shown in Figure 3.11.

The reports in XML format are easy to automatically process, but difficult to read by human. ACUA also provides post-process modules to convert the reports to plain text format and to summarize the contents in the reports for statistical analysis or visualization. As shown in Section 5.3, developers can process the API change and API usage reports and generate the graph of the distributions of the types of API changes in the frameworks and compare it with that of the distributions of the types of API changes used in the client programs.

3.2.2 Dataset Description

In our study, we use two sources of data: Maven central repository (van Zyl and Apache, 2005) and Eclipse SDKs (Eclipse, 2014). The programs from these two sources have structured dependency and version information between client programs and frameworks. We aim to study both API changes and usages, so we need not only frameworks and their client programs, but also their dependencies and versions. Because of the increasing inter-dependencies between programs (Bavota et al., 2013), a program can be both a client program of other frameworks or used as a framework by other programs. In our study, if a program exists in the dependencies of other programs, we treat it as a framework and analyse its API changes. If the program also depends on other programs, we analyse the used APIs as well.

Our exploratory study of API changes and usages has two objectives: an overview and a more detailed investigation into different types of API changes and usages. We use different datasets to achieve the two goals.

To present an overview of API changes and usages, we analyse the programs for Maven central repository at method level. There are 14,987 programs in our snapshot of Maven central repository taken by September 2014. Among them, 3673 programs are developed by Apache and 115 programs are developed by Eclipse. Methods are the most important

```
<Incompatibilities>
    <programName>
    org.apache.lucene:lucene-core
    </programName>
    <oldVersion>3.6.2</oldVersion>
    <newVersion>4.0.0</newVersion>
    <incompatibilities>
        <type>AddAbstractMethod</type>
        <instances>
            <level>type</level>
            <oldAPI>
            class org.apache.lucene.analysis.Analyzer
            </oldAPI>
        </instances>
                ...
    </incompatibilities>
        ...
</Incompatibilities>
```

Figure 3.10 Snippet of lucene-core v3.6.2-v4.0.0 API change report

```
<exposureReport>
    <clientName>
    org.apache.solr:solr-core
    </clientName>
    <clientVersion>3.6.2</clientVersion>
    <frameworkName>
    org.apache.lucene:lucene-core
    </frameworkName>
    <oldVersion>3.6.2</oldVersion>
    <newVersion>4.0.0</newVersion>
    <apiUsages>
        <type>EXTENSION</type>
        <apiUsages>
            <oldAPI>
            class org.apache.lucene.analysis.Analyzer
            </oldAPI>
            <invocatedInFramework>
            true
            </invocatedInFramework>
            <instances>
                <entities>
                class org.apache.solr.analysis.TokenizerChain
                </entities>
                <incompatibilityType>
                AddAbstractMethod
                </incompatibilityType>
            </instances>
            ...
        </apiUsages>
                ...
    </apiUsages>
        ...
</exposureReport>
```

Figure 3.11 Snippet of solr-core v3.6.2 API usage report on lucene-core v3.6.2-v4.0.0

software elements (Ali et al., 2012) and are studied by most of previous works (Dagenais and Robillard, 2011; Godfrey and Zou, 2005; Kim et al., 2007; Kpodjedo et al., 2013; Schäfer et al., 2008; Wu et al., 2010; Xing and Stroulia, 2007a).

To investigate the different types of API changes and usages, we use the frameworks from two framework ecosystems Apache and Eclipse as the dataset. The frameworks provided by these two ecosystems are widely used in open-source software development (IBM, 2006; van Zyl and Apache, 2005) and have evolved during the last two decades. The two ecosystems are different in managing API evolution. Eclipse provides Provisional API Guidelines [6] to distinguish internal APIs and official APIs. By the definition, internal APIs can be changed without notice and documentation while official APIs are more stable. Apache does not have explicit regulations on API changes. We want to investigate if there is any difference between the frameworks from the two ecosystems with respect to API changes and usages. Also, since internal and third party client programs may use frameworks in different ways, we study the two types of client programs separately.

For Apache frameworks and their client programs, we identified the projects whose IDs start with “org.apache" as Apache frameworks or client programs and others as third party. For Eclipse frameworks and internal client programs, we select 16 releases of Eclipse SDK (1.0-4.3) directly from Eclipse archive website (Eclipse, 2009) instead of Maven central repository, because not all Eclipse frameworks are hosted there. We have 145 Eclipse frameworks (we treat each plug-in as a framework) with 1,017 releases. Eclipse does not have a central repository (as Maven repository) to host third-party plug-ins with structured version information. Businge et al. (2010) studied the evolution of 21 Eclipse third-party plug-ins. Thus, we selected 15 plug-ins and downloaded 41 releases of the third-party plug-ins used in Businge et al. (2012), which have explicit mapping to Eclipse API levels on their host websites. We manually downloaded these releases and build the client-framework version mappings.

Based on the results of the study on the overview of API changes and usages, we sort the framework releases by the numbers of changed APIs that are used by their client programs. Only 11 Eclipse framework releases were used by both internal and third-party client programs. To compare the same number of frameworks of the two ecosystems, we choose also top-11 framework releases from Apache. Finally, our data set contains 198 internal and 130 third-party client program releases for the 11 Apache framework releases and 84 internal and 28 third-party client program releases for the 11 Eclipse framework releases, in our dataset for the study on API change and usage types.

6. http://wiki.eclipse.org/Provisional_API_Guidelines

3.3 Study Results

This section presents and discusses the results of our study, according to the research questions introduced in Section 3.1.5.

3.3.1 API Evolution Overview

Following the methodology used by Robbes et al. (2012), we investigate framework API evolution in terms of frequency, adaptation, and magnitude. In this section, we present the results about the former two and the results of magnitude will be presented in Section 3.3.4.

RQ1.1: How often frameworks and client programs evolve?

For frequency, on average, each framework and client program in Maven repository have 11 (ANRF) and 12 (ANRC) releases, respectively.

RQ1.2: How many client programs adapt to framework evolution?

For adaptation, 78% (PUC) of client programs upgraded to new releases of frameworks and only 42%(PUF) of frameworks have upgraded client programs, as shown in Figure 3.12. The data on release numbers are similar with PUCR of 78% and PUFR of 49%.

Summary

Frameworks and client programs evolve with similar frequency, 11 and 12 releases per framework and client program, respectively. We argue that the increasing inter-dependencies between programs (Bavota et al., 2013) make the differences between framework and client programs less and less obvious.

Most client programs (78%) adapted to new releases of frameworks at least once, but only to less than half (42%) of frameworks. It is interesting to investigate further to find out the characteristics of the frameworks that client programs keep evolving with.

3.3.2 API Changes

API changes are the consequence of framework evolution. Better understanding of how APIs change helps developers estimate and perform upgrading tasks. Also, it helps researchers develop more effective to reduce upgrading costs. General information of API changes at large scale can provide a reference point to evaluate the severity of API changes. Different

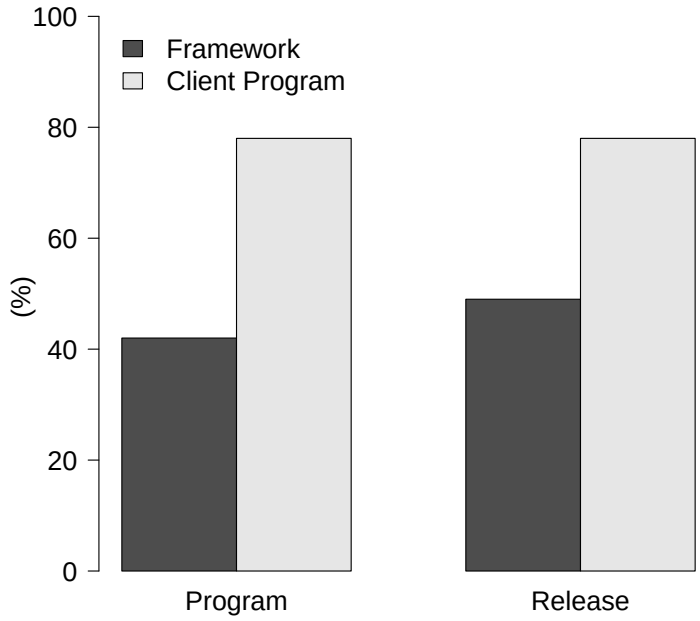


Figure 3.12 Percentages of framework and client program upgraded

types of API changes may have different impacts on client programs. Knowing which types of API change happen the most in frameworks is important for client program developers and researchers in software engineering. The answer of this question can help them better understand how framework developers modify APIs. Client program developers could take more precautions to frequent API change types and researchers can give higher priorities to develop approaches or tools for the API changes which occur more often.

RQ1.3: How many APIs change during framework evolution?

Most upgraded frameworks change their APIs, but only in a small percentage of their releases. API changes exist in more than half (59%) of these frameworks and in about one quarter (24%) of their releases. At method level, 10% APIs changed during framework evolution. Also, we find that, among all the methods with signature changes, 80% are API methods. Therefore, most changes that developers made to method signatures are in API methods, although the changed APIs are a small part of APIs.

RQ1.4: How many APIs are marked as deprecated during framework evolution?

Only 2% of changed APIs are marked as deprecated by framework developers and 52% of methods marked as deprecated are not API methods. Robbes et al. (2012) found the same phenomenon in Smalltalk frameworks. Developers do not document most API changes. In half of the cases, framework developers document the changes in methods for internal use.

RQ1.5: How often does each type of API changes happen in frameworks?

Figure 3.13 and 3.14 show the distributions of reference-type-level and method-level API change types in Apache and Eclipse frameworks. The total numbers of API change types in frameworks are shown in Table 3.6. At reference-type-level, the total number of API changes are 807 for Apache frameworks and 2,731 for Eclipse plug-ins. Three of the top four API change types in both ecosystems are the same: WMC (With Method-level Changes, 1st), MSC (Missing Class, 2nd), and ESIS (Expand Super Interface Set, 4th). The third most frequent type of API change in Apache frameworks is CSIS (Contract Super Interface Set). In Eclipse framework, the third most frequent type of API change is CSCS (Contract Super Class Set). These four types of API changes represent more than 80% of all the changes (87% for Apache and 82% for Eclipse).

The more detailed classifications of WMC are presented at method-level API changes. In total, Apache frameworks have 1,403 and Eclipse plug-ins have 3,393 method API changes. The top four types of API changes are the same, *i.e.,* MSM (Missing Method), CFP (Change Formal Parameter), CRT (Change Return Type), and DMA (Decrease Method Access), in both ecosystems and they cover 98% of the total number of method-level API changes.

Adapting different types of API changes does not require the same effort. MSC is the second and MSM is the most frequent API change at reference type and method-levels, respectively. MSC, and MSM may be caused by renaming and removing. Although renaming is relatively easy to adapt, finding the correct replacements for a large number of renamed methods or classes is still time-consuming. The adaptation of client programs to removed APIs is even more challenging.

Table 3.6 Numbers of API changes and usages

Framework		Apache		Eclipse	
API		Type-level	Method-level	Type-level	Method-level
Change		807	1,403	2,731	3,393
Usage	Internal	493	378	96	43
	Third-party	77	58	35	25

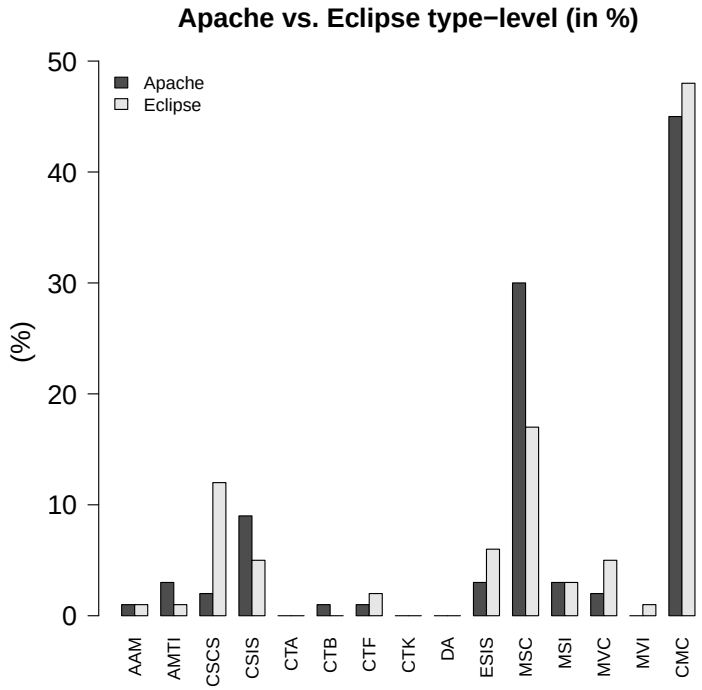


Figure 3.13 Reference-type-level API changes

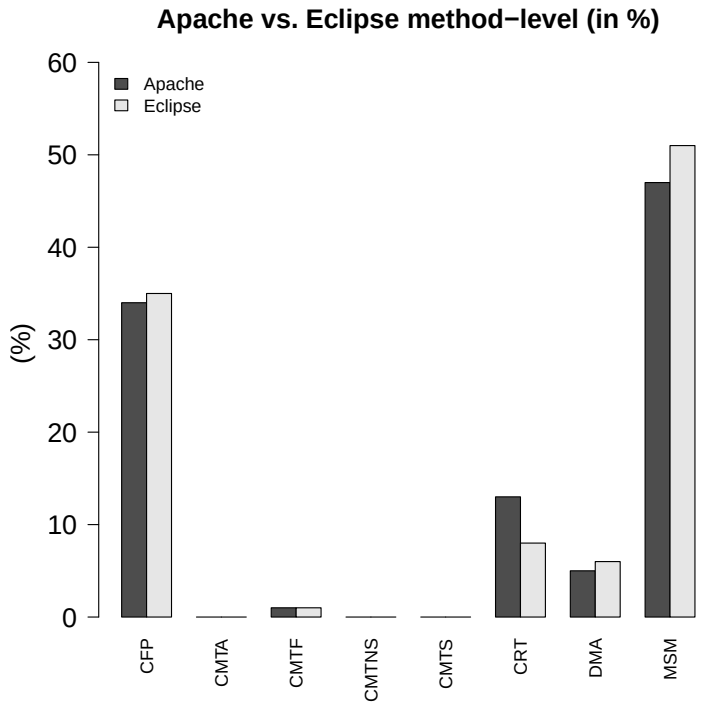


Figure 3.14 Method-level API changes

If APIs still have the same reference type or method name after being changed, finding the replacements of the changed APIs is easier for client program developers because modern IDEs, such as Eclipse, support searching by name effectively. If reference type names are changed, developers will spend time on identifying a suitable starting point for their upgrade. There is no reliable way for developers to locate the replacements of reference types with different names yet. Although IDEs support fuzzy searches and there are approaches to generate API change rules, the results of fuzzy searches depend on the quality of the queries and API change rule tools are not 100% accurate. Developers must find the replacements of the missing classes, interfaces, and methods manually, when the searches do not provide the right replacements or the change rules are not correct. Therefore, framework developers should avoid removing reference types and methods used as APIs or changing their names.

For unavoidable API name changes, effective documentations can ease the upgrading process. Especially, more detailed descriptions about where and when the new APIs are used in the new releases of frameworks are necessary for inheritance-style usage. It is time-consuming to provide detailed documentations for frameworks. However, the imperfect change rules generated by tools, such as (Arnaoudova et al., 2014; Dagenais and Robillard, 2011; Kim et al., 2007; Meng et al., 2012; Wu et al., 2010), do help developers to find the replacement of missing APIs and can be used as supplements to insufficient documentations, as shown by our empirical study (Wu et al., 2014b). If tools can generate documentations while performing code-refactoring, it would also be a great help, because Dig and Johnson (2006) showed more than 80% API changes are caused by refactoring. In practice, the percentages of the API changes caused by refactoring vary between frameworks, but these API changes may be difficult to detect because sequential refactorings can confuse the tools.

Summary

Most (59%) upgraded frameworks change their APIs but, only in a small percentage (24%) of their releases. Most changes (80%) that developers made to method signatures are in API methods, although the changed APIs are a small part (10%) of APIs. Developers only document small part (2%) of API changes. In about half (52%) of the cases, framework developers document the changes in methods for internal use.

Missing classes (MSC) and methods (MSM) are the most frequent API change. MSC and MSM may be caused by renaming and removing. Adapting client programs to frameworks with such API changes is challenging. Framework developers should provide more detailed documentation to guide client program developers upgrading. When such documentations are missing, the imperfect change rules generated by tools, such as (Arnaoudova et al., 2014;

Dagenais and Robillard, 2011; Kim et al., 2007; Meng et al., 2012; Wu et al., 2010), can help developers find the replacement of missing APIs. Tools generating documentations while performing code-refactoring is interesting future work.

3.3.3 API Usages

Framework APIs may change between releases. Knowing how widely APIs infiltrate or are used in client programs will help developers estimate the upgrading cost of API changes. For example, a changed API used in many locations in client programs may cause the Shotgun Surgery (Fowler et al., 1999) code smell, *i.e.,* lots of little changes in multiple classes, which would result in high upgrade costs. General information of API usages at large scale can provide a reference point when developers perform such evaluations.

API infiltration can be caused and amplified by using API in change-propagating ways, such as $U2 - U5$ in Table 3.4. It is interesting to know how much of the API usages belonging to $U2 - U5$, from which developers can start reducing API infiltration. Researchers can develop approaches or tools to facilitate this task.

RQ2.1: How many APIs are used by client programs?

Client programs use only 16% of the APIs provided by the frameworks. The low number of the used APIs may be caused by that client program developers do not need or do not know how to use the other APIs (Kawrykow and Robillard, 2009). In both cases, framework developers should examine the design and documentation of the APIs that they currently provide, then decide to encapsulate the unwanted APIs or provide more efficient documentation to help client program developers use those necessary.

RQ2.2: How many used APIs are marked as deprecated?

We did not find any case where client programs use APIs marked as deprecated by framework developers. This is a positive reaction from client program developers to the warnings given by framework developers about API changes. However, framework developers still keep 98% of the APIs marked deprecated, as we show in Section 3.3.2.

RQ2.3: How widely do APIs infiltrate in the client programs?

Table 3.7 lists the numbers of releases and the average numbers of the reference types in the client programs of Apache and Eclipse frameworks. Generally, the internal client programs

are larger than those from third-party. Figure 3.15 shows their IRs. Similar to their sizes, internal client programs have higher IRs than third-party client programs in general, but both can reach 100%, which means that all the reference types of these client programs could be affected by API changes. In Figure 3.15, the Y axis represents the values of IRs of client programs while the width of the graph in X axis represents the number of client programs with corresponding IR values. For example, from Figure 3.15(a), we can learn that the number of client programs with IR of 10% is as double as the number of those with IR of 40% in Apache internal client programs.

Framework APIs are used in more places in internal client programs than third-party client programs. The IRs are 44% and 38% in Apache, and 37% and 26% in Eclipse, for internal and third-party client programs, respectively. The average IR is 36%.

RQ2.4: How widely does API change-propagation exist in client programs?

As shown in Figure 3.16, the values of ACPRs are 22% and 24% in Apache, and 16% and 14% in Eclipse, for internal and third-party client programs, respectively. On average, ACPR is 18% for the client programs in our dataset. Developers can encapsulate the framework API usages of $U2 - U5$ with local APIs, such as using composition to replace $U2$. First, such encapsulation can reduce IRs directly by avoiding $U2 - U5$. Second, the number of $U6$ caused by $U2 - U5$ also decreases. Consequently, client programs will be affected less by framework API changes.

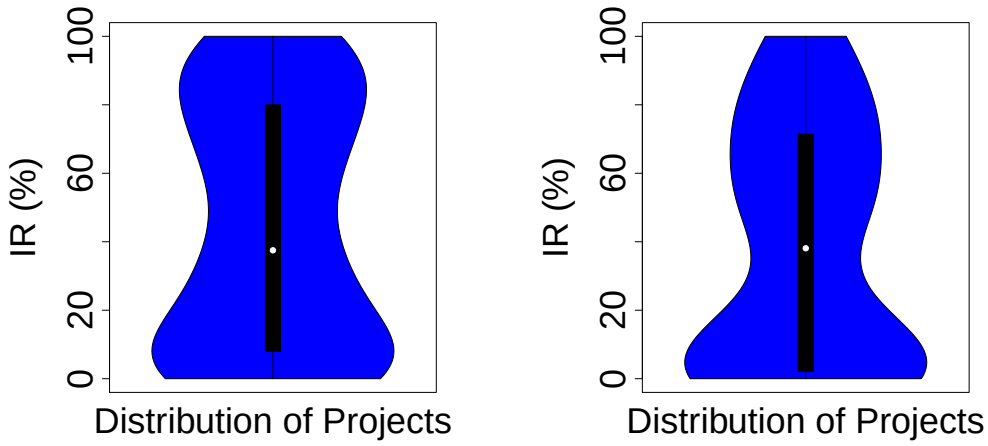
RQ2.5: How many API usages in client programs can be encapsulated?

However, the framework API usages in client programs can be reduced except $U1$, *i.e.*, inheritance for IOC. We use IIR to represent the lowest boundary of API usage in client programs. It is the ratio of client program classes and interfaces used for IOC.

As shown by the IIR distributions in Figure 3.17. The IIRs are 11% and 3% for Apache internal and third-party client programs, respectively, while those are 4% and 1% for Eclipse client programs.

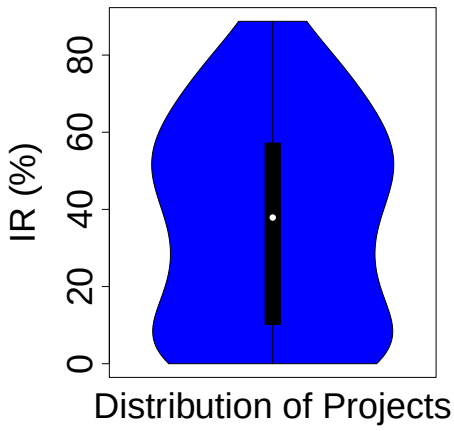
Table 3.7 Numbers of and average numbers of reference types in client program releases

Ecosystems	Apache		Eclipse	
	Internal	Third-party	Internal	Third-party
# Client Program releases	198	130	84	28
Average # Reference Type	193	140	400	72

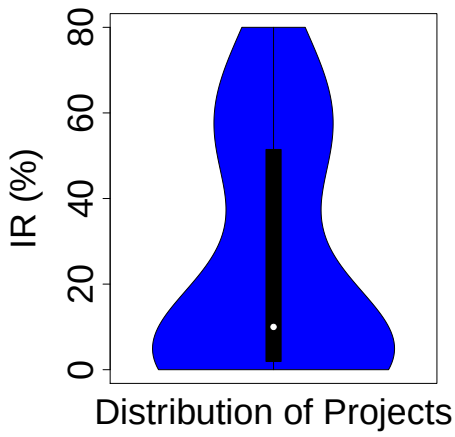


(a) Apache internal

(b) Apache third-party

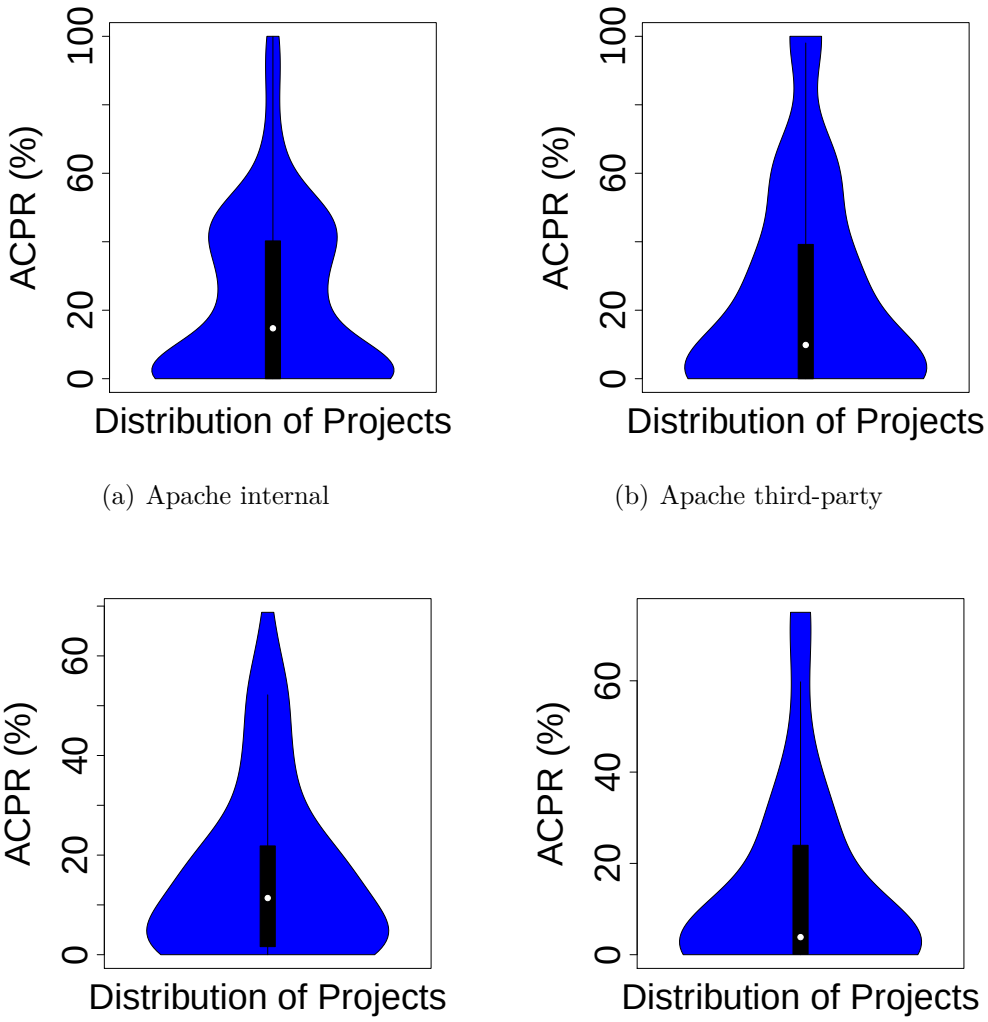


(c) Eclipse internal



(d) Eclipse third-party

Figure 3.15 Infiltration Ratios



(a) Apache internal

(b) Apache third-party

(c) Eclipse internal

(d) Eclipse third-party

Figure 3.16 API Change-Propagation Ratios

On average, the differences between IRs and IIRs are 33% and 35% in Apache, 33% and 25% in Eclipse, for internal and third-party client programs, respectively. In total, the average IR and IIR of all the client programs in our study are 36% and 5%, *i.e.*, more than 80% of API usages can be encapsulated with composition.

Besides IRs and IIRs, the distributions of the number of APIs used by each reference type in client programs can provide more detailed information of API usages in a specific client program. Figure 3.18 shows an example between *cse.green.relationship.composition* v2.5.0 using *eclipse.jdt.core* v3.2 and *anyedit.AnyEditTools* v1.8.2 using *eclipse.ui.editor* v3.2. The Y axis represents the numbers of the APIs from the two frameworks that the two client programs use in a class or interface. The width of the graph in X axis represents the proportion of classes and interfaces where the two client programs use corresponding numbers of framework APIs. Figure 3.18(a) and Figure 3.18(b) shows that *cse.green.relationship.composition* uses more APIs from *eclipse.jdt.core* than those *anyedit.AnyEditTools* uses from *eclipse.ui.editor*. Also, the API usages in *cse.green.relationship.composition* spread wider than *anyedit.AnyEditTools*, because the former has more reference types using many APIs than the latter. Therefore, *cse.green.relationship.composition* is more likely to get affected by the API changes in *eclipse.jdt.core*.

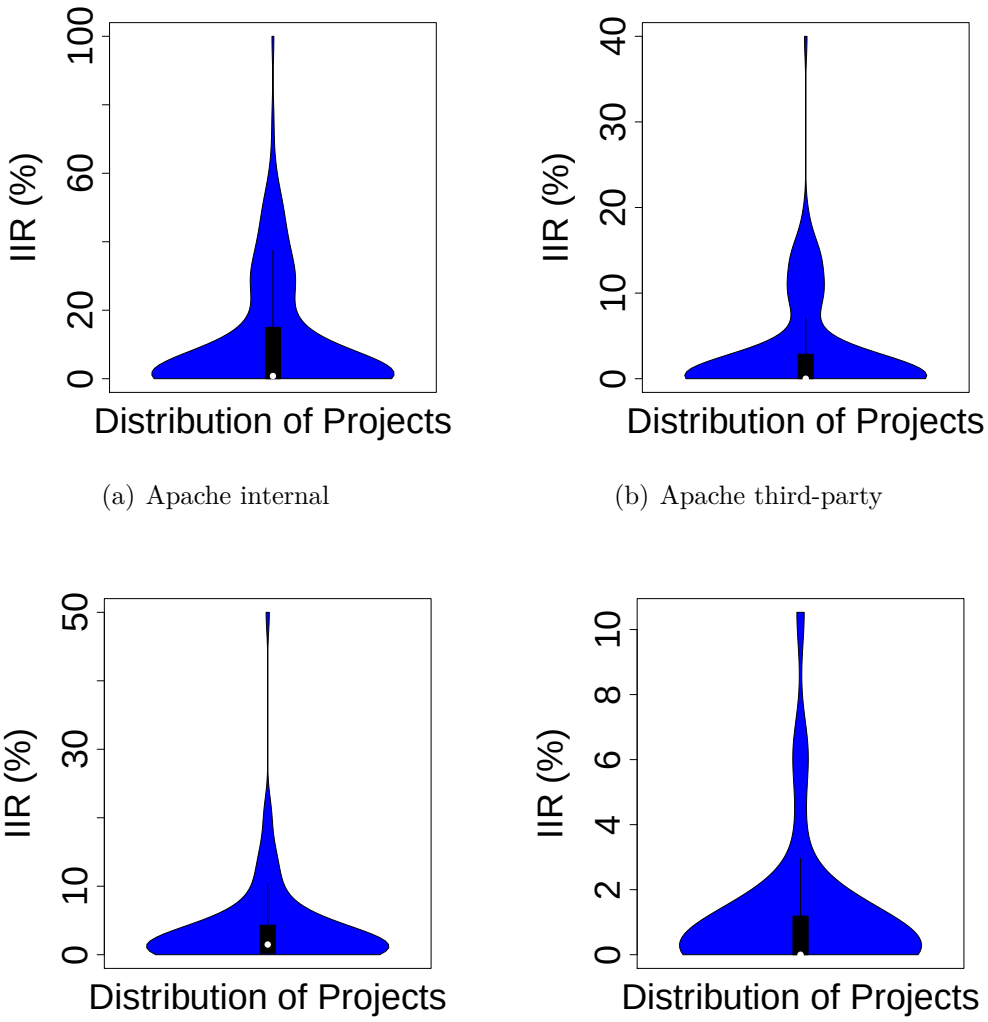
Based on the shapes of the distributions, we can learn that the Kisses shape of the usages of APIs in *anyedit.AnyEditTools* is preferred over the Vase shape of API usages in *cse.green.relationship.composition*. The former has less reference types using framework APIs, also using smaller numbers of APIs in classes and interfaces, than the latter.

Summary

Client programs only use a small part (16%) of framework APIs and do not use APIs marked as deprecated. Framework developers should have more concise design to encapsulate the unwanted APIs or provide more effective documentation to help client programs to use APIs.

Besides API changes, API usage is another factor affecting client program upgrading costs. Widely-spread API usages are more difficult to adapt to API changes. On average, framework APIs are used in 36% (IR) of the classes and interfaces of client programs and 18%(ACPR) of them are used in change-propagating ways. The results of IIR (5%) show that 80% of API usage can be reduced. Also, the low value of IIR reveals that frameworks are designed for IOC, but their APIs are accessed as libraries more often.

Client developers should keep monitoring the IRs of their client programs and keep their values as low as possible. The difference between IR and IIR shows the room for reducing



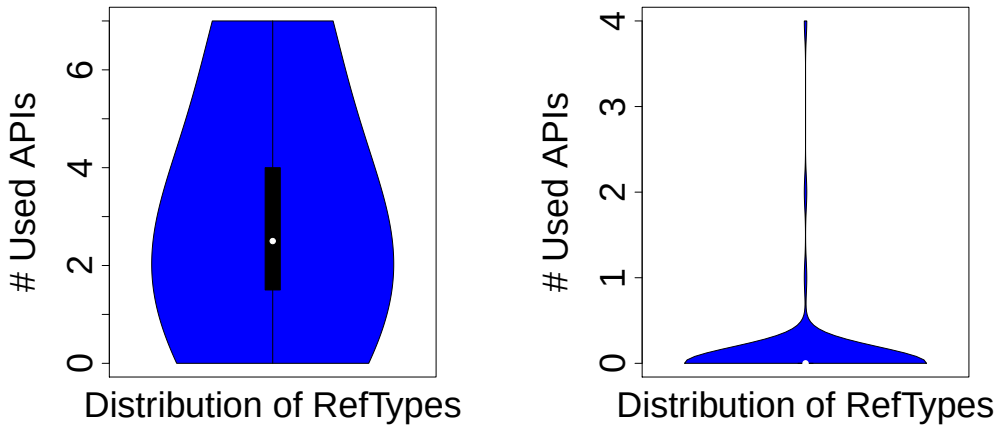
(a) Apache internal

(b) Apache third-party

(c) Eclipse internal

(d) Eclipse third-party

Figure 3.17 Ideal Infiltration Ratios



(a) cse.green.relationship.composition v2.5.0 (b) anyedit.AnyEditTools v1.8.2

Figure 3.18 Project API infiltration

API infiltrations. As recommended by Bloch (2008), composition-style usage can encapsulate framework APIs and prevent propagating their changes. Also, composition-style usage avoids *fragile base class problem* (Mikhajlov and Sekerinski, 1998) affecting client programs, *i.e.*, internal changes in base classes could break client programs, even there is no API change. Approaches and tools help developer for such specific encapsulation tasks are interesting for researchers in software engineering.

Reducing framework API usages in client programs can be further illustrated as Figure 3.19. Ideally, client programs should only keep $U1$ and $U6$ and encapsulate $U2 - U5$ with composition-style usage. Tools like ACUA can help developers by identifying $U2 - U5$ usages.

IIR is the lower boundary of IR, but it is not necessary for client programs to reach this boundary for all the frameworks. Developers decide on how tight the coupling between their programs and a framework should be. For stable frameworks, developers can stay with the current IR level because their APIs will rarely change between releases. They should reduce IRs for unstable or to-be-replaced frameworks because this will probably minimize the changes required to upgrade to new releases.

Inheritance-style usage or $U1$ is also not avoidable, because frameworks are designed for inversion of control. However, developers still can protect client programs by loosening the coupling to framework APIs (Gamma et al., 1995). Facade or Adapter patterns (Gamma et al., 1995) can provide a buffer layer between client programs and frameworks. Thus, client programs and frameworks can evolve relatively independently.

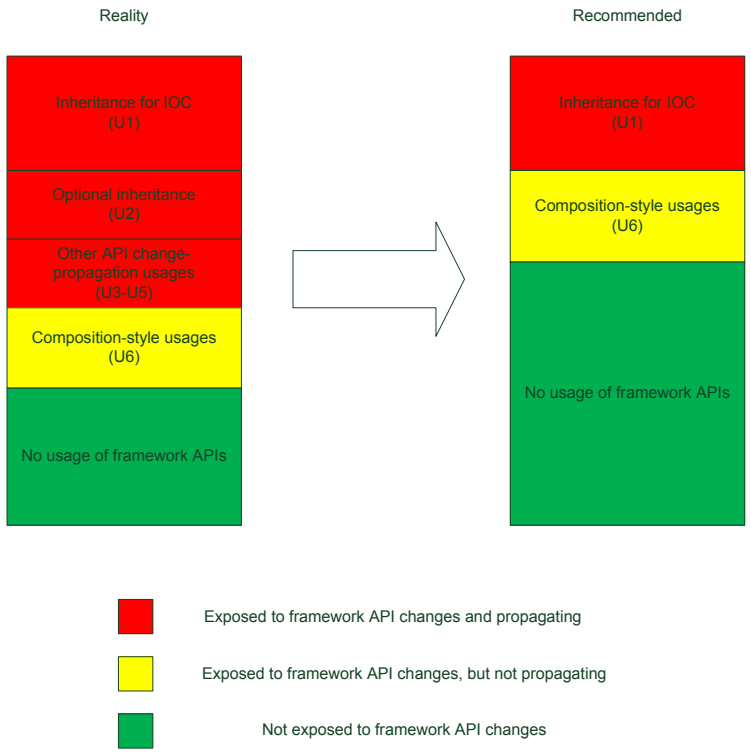


Figure 3.19 API usage recommendation

3.3.4 API Change Effects

After investigating how APIs change in frameworks and are used in client programs, we further study how the changed APIs affect client programs. First, we check API change effects at program level. On the one hand, we investigate how many upgraded client programs depend on frameworks with API changes. Such client programs may not directly use changed APIs, but may use them in future releases. On the other hand, we analyse how many upgraded client programs directly use changed APIs. Second, we check how many APIs are used at method level. Third, because different types of API changes do not occur with the same frequency in frameworks. They may not happen in client programs equally either. Knowing the frequency of different types of API changes in client programs, framework developers could decide to avoid these types of API changes whenever possible and–or provide more detailed documentation (e.g., about how to upgrade client programs accordingly) whenever they occur during a framework evolution. Researchers could also develop techniques and tools to ease the upgrading process of client programs to adapt such frequent changes.

RQ3.1: How many client programs adapt to frameworks with API changes?

For the magnitude of the effect of API changes, most upgraded frameworks change their APIs, but only in a small percentage of releases as shown in Figure 3.20. PUCF and PUCFR show that API changes exist in 59% of frameworks upgraded and 24% of their releases. On client program side, most of client programs upgraded (90% of PUCC) use frameworks with API changes in 60% (PUCCR) of client program releases.

API changes in about one third (37% of PUCUF) of upgraded frameworks with changed APIs affect client programs and 24% (PUCUFR) of the releases of frameworks upgraded with changed APIs are used by client programs, as shown in Figure 3.21. Comparing to frameworks, higher percentages of client programs and their releases affected by API changes in frameworks. 69% (PUCUC) of client programs depending frameworks with changed APIs used the changed APIs in 45% (PUCUCR) of their releases.

RQ3.2: How many changed APIs are used by client programs?

At method level, only 3% of the used APIs are changed in the next releases of frameworks, but none of them are marked as deprecated.

RQ3.3: How often is each type of API changes used in client programs?

The numbers of API changes affecting client programs are less than those in frameworks as shown in Table 3.6. We present API changes affecting client programs as *used API changes*. The API changes that occur more frequently in frameworks are not always the more often used in client programs. Figure 3.22 show the proportions of API changes at the reference-type-level that are used in client programs. ESIS (ExtendSuperInterfaceSet) is the fourth most frequent API change type in frameworks of both ecosystems, but it does not affect client programs. The other most frequent API change types in frameworks also affect client programs more often, just in different orders.

Moreover, reference-type-level API changes not only affect client program at class and interface level, but also at method-level. For example, two classes A and B from $V1$ of a framework FR do not exist in $V2$. Their API change type is MSC. One method $a1$ from class A, two methods $b1$ and $b2$ from class B are used in a client program CL of FR. At reference level, MSC affects CL two times. If we examine closer, we can notice that the effect caused by B is more serious than that caused by A, because B has two methods used by CL while A only has one. We call the methods in the changed classes and interfaces, such as $a1$, $b1$, and $b2$, as *affected methods*.

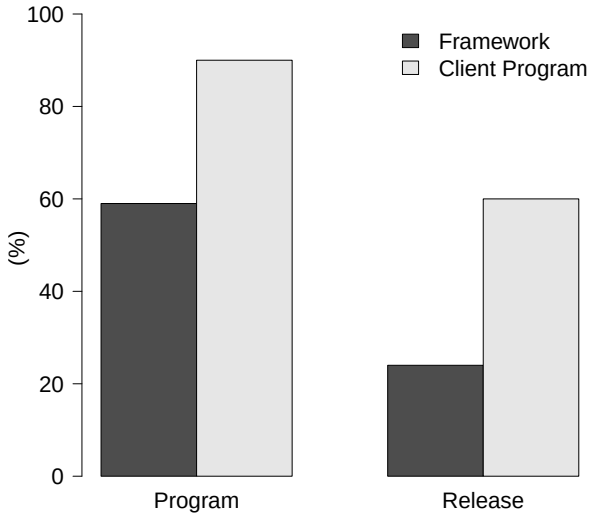


Figure 3.20 Upgraded with API changes framework and client program percentage

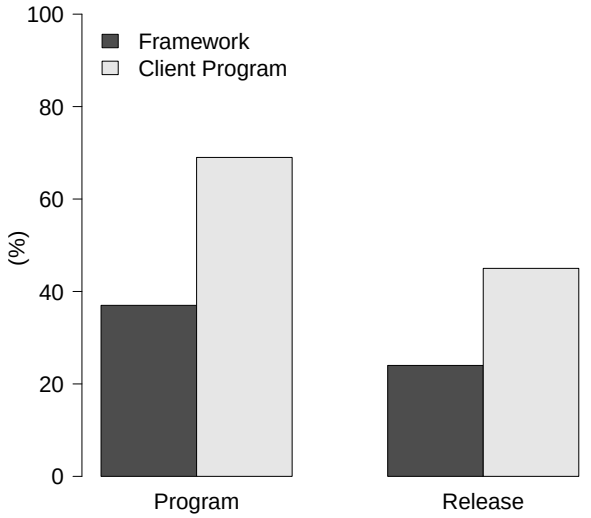


Figure 3.21 Percentages of framework and client program upgraded with changed and used APIs

Table 3.8 Numbers of reference-type-level API changes with affected methods

Framework		Apache		Eclipse	
API		Type-level without affected methods	Type-level with affected methods	Type-level without affected methods	Type-level with affected methods
Usage	Internal	493	1,453	96	232
	Third-party	77	137	35	47

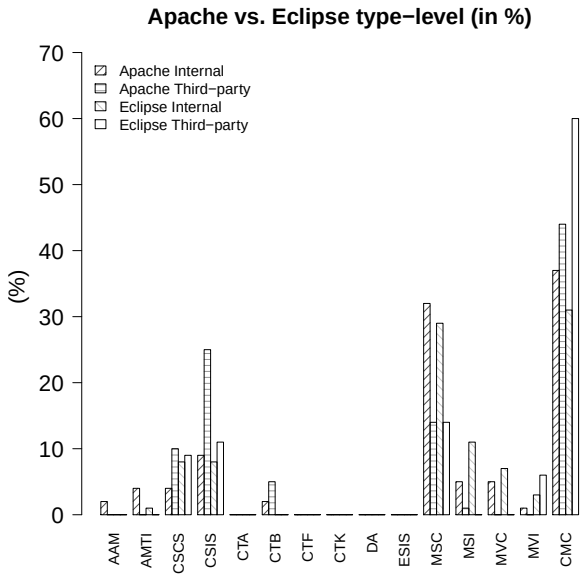


Figure 3.22 Reference-type-level API changes used in client programs

Therefore, we want to check the values of P_{U_T} at reference-type-level, when we take their affected methods in classes and interfaces into account. In our detection, we only detect method-level changes in the classes and interfaces under reference type API change WMC which have the same signatures in two releases of frameworks, because we can match two versions of methods reliably to detect detailed changes, such as those in parameters or modifiers. However, we still can count reliably how many methods in the classes and interfaces under each API change type without knowing the detailed changes in their methods. The columns under "Type-level with affected methods" in Table 3.8 show the numbers. We can see that more than one method from the changed classes and interfaces used by client programs.

Figure 3.23 shows the proportions of each reference-type-level API changes with affected methods. The top-2 API change types that are used in client programs is the same (in some cases the 1st and 2nd positions are switched) as the top-2 types of API changes that occurred the most in frameworks, *i.e.*, WMC and MSC. The other API change types used in client programs become different when we take the affected methods into account.

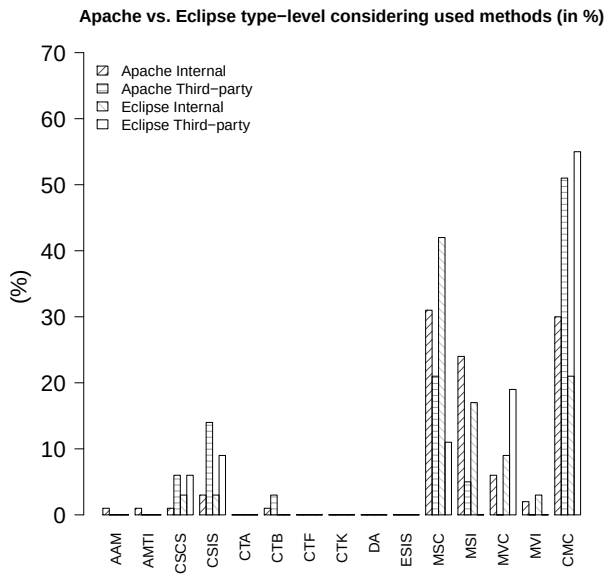


Figure 3.23 Reference-type-level API changes with affected methods used in client programs

In Apache frameworks, the order of used API changes in client programs is different than the order of API changes in frameworks. The 3rd to 5th top API change types are CSIS, ESIS, and MSI with 9%, 3%, and 3% of P_T values, respectively. In Apache internal client programs, the 3rd to 5th used API changes are MSI (Missing Interface) and MVC (Moved Class), and CSIS with P_{T_U}s of 24%, 6%, and 3%, respectively. The data show that MSI and MVC affect client programs more often than they occur in frameworks. In Apache third-party client programs, CSIS and CSCS, and MSI are the 3rd to 5th top used API changes with 13%, 6%, and 5% of P_{T_U} values. Each of these three change types only represents less than 3% of reference-type-level API changes.

The results of Eclipse frameworks and their client programs are similar. In Eclipse frameworks, CSCS, ESIS, and MVC are the 3rd to 5th top API change types with 12%, 6%, and 5% of P_T values, respectively. They are not the 3rd to 5th top used API changes in client programs. In Eclipse internal client programs, MSI, MVC, and MVI are the 3rd to 5th most used API changes with 17%, 9%, and 3% P_{T_U} values. In Eclipse third-party client programs, MVI, CSIS take the 3rd and 4th positions with 11% and 9% P_{T_U} values respectively. CSCS is only at the 5th position with 6% of P_{T_U} value.

At method-level, the top-3 API changes used in client programs are the same as those that occurred the most in the frameworks from both ecosystems, *i.e.,* MSM, CFP, and CRT. However, the ranking orders are different as shown in Figure 3.24.

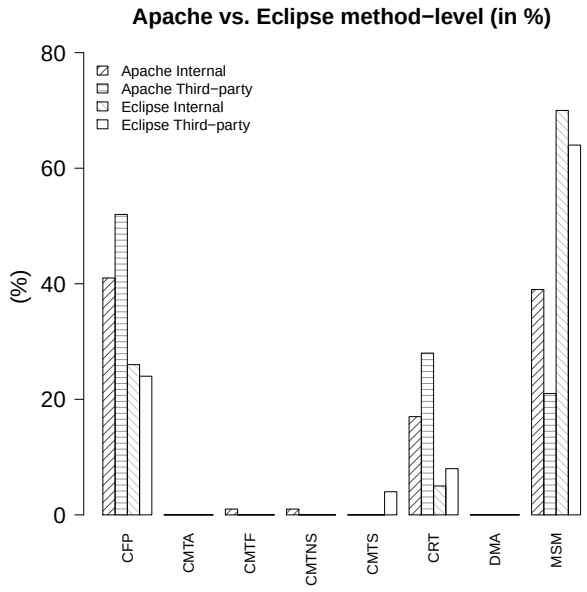


Figure 3.24 Method-level API changes used in client programs

Summary

We observe that about half of client programs are directly affected by changed APIs of small percentages of frameworks, as shown in Figure 3.25. Only 9% of total frameworks with changed APIs in 3% of framework releases are used by client programs. However, these changed APIs are used by 49% of total client programs in 21% of client program releases. Considering the number of frameworks and client programs, the magnitude of the influence of API changes is large. More than 29,000 releases of 5,845 client programs are directly affected by API changes.

The API change types at reference type level occurring more often in frameworks also affect client programs more often, except ESIS (ExtendSuperInterfaceSet) which does not used in client programs. Also, we find that some less frequent API changes at reference type level in frameworks affect client programs more often, when we consider the number of methods in the changed classes and interfaces used in client programs. For example, MSI (Missing Interface) are only in 3% of all the API change types at reference type level in the frameworks from both Apache and Eclipse. However, when we investigate the numbers of methods in such changed reference types used in client programs, it affects 24% and 17% of such methods in internal client programs from Apache and Eclipse, respectively. MSI affects 5% of the methods in the changed classes and interfaces used by Apache third-party client programs, but it does not affect Eclipse third-party client programs. This phenomenon may be caused by that

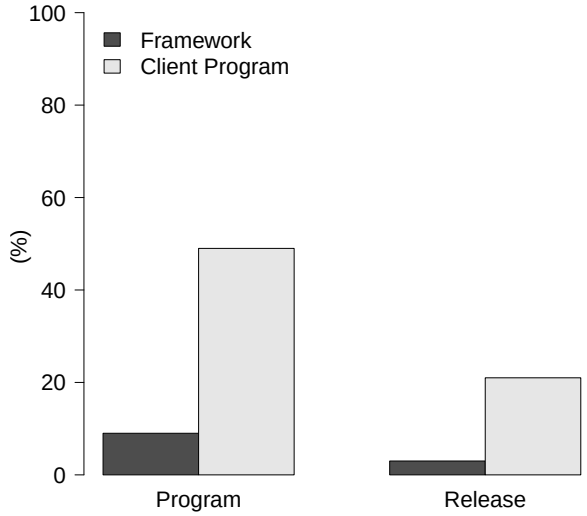


Figure 3.25 Percentages of framework and client program upgraded with changed and used APIs in all

third-party client programs use less framework APIs and Eclipse has strong restriction on API changes in its Provisional API Guidelines [7]. The API change types at method level affect the client programs with similar frequencies in which they occur in frameworks.

3.4 Discussion

In this section, we discuss the differences between special tools such as ACUA and compilers for framework upgrade tasks and the threats to validity of this study.

3.4.1 Comparison to Compilers

Compilers can be used to find the impact of API changes in client programs, but do not provide the following information needed to plan and estimate the cost of framework upgrades:

API change type: compilers can detect API changes by showing compilation errors, but they do not report which types of API changes caused the errors. Because API changes are not equally difficult to adapt, it is important to know which type of API change caused the errors, to plan program upgrade effectively. ACUA analyses client programs and framework releases and generates API usage reports that summarize which APIs are used, where they

7. http://wiki.eclipse.org/Provisional_API_Guidelines

are used (in client programs) and the types of API changes affecting them. With ACUA API usage reports, developers have a clearer picture of API change impacts.

API infiltration: compilers cannot report how widely an API is used in client programs. APIs keep evolving, developers should know how APIs infiltrate their client programs to prepare for future API changes. If large numbers of the APIs of a framework are widely used in client programs, it will be difficult for these programs to adapt to major changes in the framework or to switch to another framework with similar functions. ACUA can compute IR and IIR to tell developers the current API infiltration ratio and its possible lower boundary. The API usage data collected by ACUA can also be used to visualize API infiltration, as shown in Figure 3.15.

API change-propagation: compilers cannot inform developers of which API usages can be contained. Composition-style usage can encapsulate frameworks APIs with local APIs and reduce API infiltration. The IOC purpose of framework requires that client programs must use some APIs in inheritance-style, but not all of them. ACUA analyses API usage types and reports which APIs are not for IOC and can be encapsulated. AUCA also detects API change-propagation in client programs, *i.e.*, optional inheritance, using framework reference types or their subtypes as generic types, method return types, or formal parameter types. Reducing API change-propagation cases can control API spreading, thus brings down IR in client programs.

Such information can be obtained with special tools on framework usages and upgrades, such as Exapus (Roover et al., 2013) and ACUA (Wu et al., 2014a).

3.4.2 Threats to Validity

Some threats can limit the validity of our study. In this section, we discuss them following the guidelines provided by Wohlin *et al.* Wohlin et al. (1999).

Construct Validity Construct validity verifies that the observation really reflects the theory, *i.e.*, if the treatment reflects the cause and the outcome reflects the effect. We wanted to investigate how APIs change during framework evolution and how APIs are used in client programs. These phenomena can be observed from different perspectives. Those that we chose are suggested by experienced practitioners and summarized according to the literature. We also defined a set of metrics, such as $ANRF$, CR, UR,IR, IIR, $ACPR$, P_T, P_{T_U}, and P_{tc_tu}, to measure the outcomes. Thus, we believe that there is no construct validity threats affecting this study.

Internal Validity Internal validity verifies that the outcome is really caused by the treatment. There may be errors in the tools used to collect data in our study. We carefully tested the tools and verified their outputs. During the data analysis, we did not observe any inconsistent or conflicting result. We believe the threats to internal validity are well controlled in our study. We also described the methodology and algorithms of our study in details, in order to facilitate future replications by other researchers.

External Validity External validity verifies that the results of a study are generalizable. We identify two threats to external validity. First, we only analysed the frameworks in Maven repository. Although popular and large, these frameworks may not be representative of the general population of frameworks. Second, many client programs are also frameworks. The lack of clear difference between API usages in frameworks and pure client programs may affect our results.

3.5 Conclusion

We summarize the results of our first study, a large-scale and fine-grained analysis of the reality of API changes and usages by answering the following questions:

RQ1: How do framework APIs evolve?

Frameworks and client programs evolve with similar frequency. This phenomenon may be caused by the increasing inter-dependencies between programs (Bavota et al., 2013). Most (59%) upgraded frameworks change their APIs in a small percentage (24%) of their releases.

Most changes (80%) that framework developers made to method signatures are in API methods, although the changed APIs are a small part (10%) of the APIs. Developers only document small part (2%) of API changes. In about half (52%) of the cases, framework developers document the changes in methods for internal use.

Missing classes (MSC) and methods (MSM) are the most frequent API changes. MSC and MSM may be caused by renaming and removing. Adapting client programs to frameworks with such API changes is challenging. Framework developers should provide more detailed documentation to guide client program developers upgrading. When such documentations are missing, the imperfect change rules generated by tools, such as (Dagenais and Robillard, 2011; Kim et al., 2007; Kpodjedo et al., 2013; Meng et al., 2012; Wu et al., 2010), can help developers find the replacement of missing APIs.

RQ2: How do client programs use framework APIs?

Most client programs (78%) adapted to new releases of frameworks at least once, but only to less than half (42%) of the frameworks. It would interesting to investigate the characteristics of the frameworks with which client programs evolve.

Client programs only use a small part (16%) of framework APIs and do not use APIs marked as deprecated. Framework developers should have more concise design to encapsulate the unwanted APIs or provide more effective documentation to help client programs to use APIs.

On average, framework APIs are used in 36% (IR) of the classes and interfaces of client programs and 18% (ACPR) of them are used in change-propagating ways. The results of IIR (5%) show that 80% of API usage can be reduced through refactoring.

IIR is the lower boundary of IR, but it is not necessary for client programs to reach this boundary for all the frameworks. Developers decide on how tight the coupling between their programs and a framework should be. Because frameworks are designed for inversion of control, inheritance-style usage is not avoidable. However, developers still can protect client programs by loosening the coupling to framework APIs (Gamma et al., 1995).

RQ3: How do framework API changes affect client programs?

We observe that about half (49%) of client programs are directly affected by changed APIs of small percentages (9%) of frameworks. Considering the number of frameworks and client programs, the magnitude of the influence of API changes is large. More than 29,000 releases of 5,845 client programs are directly affected by API changes.

Missing classes and methods affect client programs more often as well, which confirms the usefulness of existing API change rule building approaches. Other API change types occurring the most in frameworks and affecting client programs the most are not always the same, and such difference should be considered when documenting API changes or developing tools to support framework upgrading.

Based on these findings, we suggest that client program developers use tools, such as ACUA, to analyse the framework API usages regularly to plan proactive framework upgrades and apply solutions, like the Adapter design patterns, to control API change-propagation. For framework developers, we suggest to avoid or provide more-detailed documentations for the API changes that occur in client programs more often and which are more difficult to adapt.

Because missing classes and methods occur in frameworks and affect client programs more often, previous change rules building approaches could help framework developers provide upgrading documentation and help client program developers identify the replacements of

these missing classes and methods. However, there are no empirical studies to quantitatively evaluate the benefits of API change rules. Consequently, we conduct our second study to evaluate the usefulness of the API change rules generated by previous approaches.

CHAPTER 4 EMPIRICAL STUDY ON THE USEFULNESS OF API CHANGE RULES

As shown by our study on the reality of API changes and usages, missing classes and methods occur more frequently in frameworks and also affect client programs more often. Frameworks are often not well documented regarding their upgrading process (Wu et al., 2014b). To ease this problem, previous approaches, such as Dagenais and Robillard (2011), Kim et al. (2007), Meng et al. (2012), Schäfer et al. (2008),Wu et al. (2010), and Xing and Stroulia (2006), generate API change rules to recommend replacements of missing APIs. However, there is no previous approaches to evaluate the usefulness of these API change rules.

To adapt the API changes of frameworks, developers usually perform four tasks: (1) find possible replacements of missing APIs, (2) find work-arounds for missing APIs if their replacements cannot be found, (3) implement the replacements or the work-arounds and, (4) test the implementation. Finding replacements for missing APIs (or identifying them as simply deleted, *i.e.*, without replacements) avoids looking for unnecessary work-arounds and–or trying wrong replacements, thus reducing the overall time to adapt to new frameworks.

An effective way to help find the replacements of missing APIs is documentation, but few companies have upgrading documentation or provide such information to the public (Wu et al., 2014b). Many approaches have been developed to recommend replacements for missing APIs. Some approaches require that the framework developers do additional work, such as providing explicit upgrade rules with annotations (Chow and Notkin, 1996) or that they record API updates to the framework (Dig et al., 2007; Henkel and Diwan, 2005; Kemper and Overbeck, 2005). However, framework developers may not be able or willing to build change rules manually or use specific tools. Thus, to avoid the extra work for framework developers, other approaches automatically identify *change rules* that describe a matching between *target methods*, *i.e.* methods existing in the old release of a framework but not in the new one, and *replacement methods* in the new release (Antoniol et al., 2004; Dagenais and Robillard, 2011; Kim et al., 2005, 2007; Meng et al., 2012; Schäfer et al., 2008; Wu et al., 2010; Xing and Stroulia, 2006). The change rules generated by these approaches are *imperfect* or not all of them are correct. Their precision varies in the frameworks analysed.

When using the change rules built by these approaches as with upgrading documents, developers do not know if the change rules are correct until having used them to modify their client programs. Based on their understanding of the frameworks, if they believe that the change rules are correct, they will modify their code accordingly and test if the replacements

indeed fix the errors and provide the expected behaviour. For the missing methods whose change rules they believe incorrect or whose replacements do not pass the tests, developers must explore the documents or source code of the releases of the framework to search for the replacements manually.

There is no empirical study of the usefulness of the imperfect change rules (generated by tools or from other sources) to show that these change rules help developers to identify the replacements more accurately and faster than without change rules or, rather, that they confuse developers because they are not all correct.

Although we could expect that using already-known all-correct change rules would help developers, it could actually slow them down in the case which developers would not be certain that the change rules are correct. According to Fagard et al. (1996), providing no information is actually better than providing the wrong information: it is less confusing and distracting. Knowing the usefulness of imperfect change rules could encourage and direct research on framework API evolution.

Therefore, we design and conduct an experiment to evaluate the usefulness of framework API evolution change rules and answer these two research questions:

— RQ_1: Is there a difference between the precision of the replacements of the target methods found by the subjects with all-correct, imperfect, and no change rules?
— RQ_2: Is there a difference between the times that the subjects spend to find replacements with all-correct, imperfect, and no change rules?

In the experiment, the subjects find the replacements of target methods with the help of all-correct, imperfect, and no change rules. Then, we measure the performance of the subjects by the precision of the replacement methods that they find and the time that they spend. To limit the influence of a specific framework on the results, we ask the subjects to analyse three medium size frameworks (JHotDraw v5.2–v5.3, JFreeChart v0.9.11–v0.9.12, and JEdit v4.1–v4.2). To limit the experiment time to about one hour, we did a pre-experiment evaluation of the required time and found that analyzing seven changed APIs in each framework can meet the experiment time requirement. Thus, we randomly select seven change rules for each framework. Among them, five correct rules and two incorrect change rules in the imperfect change rules of JEdit v4.1–v4.2 and six correct rules and one incorrect change rule for JFreeChart v0.9.11–v0.9.12 and JHotDraw v5.2–v5.3. The precision of the imperfect change rules is similar to that of the state-of-the-art approaches to framework API evolution.

We use a randomized, complete block design (Wohlin et al., 1999) in our experiment to minimize the number of subjects required and to lessen some threats to validity, discussed in

Section 4.4.7. Under this design, each subject performs experiments with all the three frameworks and the three sets of change rules, but the orders of the combinations of frameworks and change rule sets are randomized. In total, 31 subjects participated in the experiment. The statistical analysis shows that the precision of the replacements of target methods found by the subjects with all-correct, imperfect, and no change rules are significantly different with average values of 82%, 71%, and 57%, respectively. The effect size, Cliff's Delta (Grissom and Kim, 2005), of the difference in precision between the subjects with no and imperfect change rules is large and that between the subjects with imperfect and all-correct change rules is moderate. Different from the precision values, the time that the subjects with the three treatments spend to find the replacement methods is not statistically different with average values of 24, 23, and 25 minutes (1,413, 1,338, and 1,479 seconds), respectively.

These results are evidence that change rules generated by framework API evolution approaches are useful, even when some of the change rules are incorrect. Yet, as expected, the higher precision the change rules have, the more help they provide. Thus, the imperfect change rules can be used instead of unavailable documentation or as complement to partial documentation. Developers of frameworks could also use them as starting point to build upgrading documentation. The difference in accuracy between subjects with imperfect and all-correct change rules is moderate. Therefore, improving the precision of change rules will still help developers.

4.1 Study Design

The goal of our experiment is to verify that the imperfect change rules help developers to find the replacements of target methods more accurately or faster than without them. The usefulness of imperfect change rules describing API evolutions is the quality focus of our experiment and its perspective is to help developers to decide if they should use imperfect change rules when they upgrade their client programs to new releases of frameworks, when documentation is not available, complete, or up-to-date. The context of our experiment is finding the replacements of 21 target methods of three frameworks by 31 subjects with three treatments: (1) all-correct, (2) imperfect, and (3) no change rules. The correctness of the change rules is unknown to the subjects. We follow the guidelines described by Wohlin et al. (1999) to present our experiment.

4.1.1 Research Questions

We expect that the results of the experiment can answer the two research questions:

— RQ_1: Is there a difference between the precision of the replacements of the target methods found by the subjects with all-correct, imperfect, and no change rules?
— RQ_2: Is there a difference between the times that the subjects spend to find replacements with all-correct, imperfect, and no change rules?

In the experiment, although we do not give any time constraint to perform the tasks, we cannot expect all subjects to be able or willing to finish all of them. Therefore, we take both the time spent and the precision of the answers into account. The differences between our experiment and real context are discussed in Section 4.4.5.

4.1.2 Objects

The objects of our experiment are the source code of two releases of three frameworks written in Java. We choose three medium size programs as objects of our experiment. These three programs are among those that we and other researchers have analyzed: JEdit v4.1–v4.2, JFreeChart v0.9.11–v0.9.12, and JHotDraw v5.2–v5.3. JEdit is a text editor [1]. It has 37 releases between 2000 and 2013. Its API and implementation between its v4.1 and 4.2 changed dramatically. JFreeChart is a chart library [2]. There are 54 releases between 2000 to 2013. The APIs between its v0.9.11 and v0.9.12 also changed a lot and there are many APIs in v0.9.12 with very similar names. The dramatic changes in the two programs are challenging for both developers and the approaches to identify API changes. JHotDraw is a GUI framework [3] developed by Gamma *et al.* to demonstrate the application of design patterns (Gamma et al., 1995). It is less active compared to the former two programs with only 12 releases between 2001 and 2011. Yet, between v5.2 and v5.3, there are several API changes. Previous approaches to identify API changes have all very high precision when applied on these two versions of JHotDraw.

We use these programs for four reasons. First, Java is supported by most framework API evolution approaches (Dagenais and Robillard, 2011; Kim et al., 2007; Meng et al., 2012; Schäfer et al., 2008; Wu et al., 2010; Xing and Stroulia, 2006). Second, by using programs with which we are familiar, we can more easily identify correct and imperfect change rules and better evaluate the results of the subjects on the tasks in the experiment. Third, these programs have different characteristics regarding framework API evolution. We believe that

1. `http://www.jedit.org`
2. `http://www.jfree.org/jfreechart`
3. `http://www.jhotdraw.org`

they are representative of typical API evolution patterns. Fourth, these programs are of medium sizes with numbers of lines of code ranging from 9,441 to 64,710, as shown in Table 4.1: the subjects do not have to spend too much time to explore them.

The source code of the three programs is provided to subjects in an Eclipse workspace. We also recommend the subjects to explore the source code with Eclipse. They are available online [4]. We present change rules and collect subjects' answers using a dedicated Web site described in Section 4.2.1.

Table 4.1 Object Systems

Frameworks	Releases	# Methods	# SLOC
JHotDraw	5.2	1,486	9,441
	5.3	2,265	14,612
JEdit	4.1	2,773	46,176
	4.2	3,547	59,804
JFreeChart	0.9.11	4,751	59,060
	0.9.12	5,197	64,710

4.1.3 Tasks

We also consider the difference between experiment and real context, when we design the tasks for the subjects. While finding replacements of missing methods, developers can use the compiler and their tests to verify if a replacement is correct. We do not provide client code in the experiment for two reasons. First, the client code could have helped the subjects to verify if the replacements were correct, but it cannot help developers find the correct replacements. Second, there would have been extra effort for the subjects to make the client code executable using the replacements. For example, if there was a new parameter in a replacement, to find out the proper value for that parameter would have taken time too. Such effort cannot be saved by change rules and we excluded it from our experiment.

Consequently, we design our experiment tasks as program comprehension to verify if the imperfect change rules can help subjects locate the replacements faster or more accurately than with all-correct or no rules. In our experiment, we measure both the precision of the answers of the subjects and the time that they spend. So, incorrect answers do not prevent us to evaluate subjects' performance.

4. `http://www.ptidej.net/download/experiments/emse13a`

We select a set of target methods from each framework and ask the subjects to find their replacements in the source code of the new release with all-correct, imperfect, and no change rules. For example, a subject is given a target method from JEdit v4.1, `void org.gjt.sp.jedit.gui.FloatingWindowContainer.save(org.gjt.sp.jedit.gui.DockableWindowManager.Entry)`, which does not exist in v4.2. She searches for its replacement by going through the source code of JEdit v4.1 and v4.2. According to our design, she may be provided with correct, incorrect, or no replacements methods, depending on her treatment. The subject does not know if the change rule is correct or not, therefore she must verify it by understanding the source code. In the end, she may or may not find that the actual replacement is `org.gjt.sp.jedit.GUIUtilities.saveGeometry(java.awt.Window,java.lang.String)`. We measure her performance by the precision of the replacement methods that she found and the time that she spent on the task.

4.1.4 Subjects

The subjects are volunteer software practitioners familiar with Java. Among the total 31 subjects, nine are B.Sc., four are M.Sc., and 16 are Ph.D. students in computer science or software engineering, two subjects are professional software developers; 10 subjects are female and 21 are male. Their task distribution is shown in Table 4.18 in the end of the chapter.

4.1.5 Independent Variable

The independent variable of our experiment is the kind of set of change rules provided to subjects. We have three treatments:

— TR_c: All-correct change rules.
— TR_i: Imperfect change rules.
— TR_n: No change rule.

A change rule is the mapping between a target method and their replacements in the new release of a framework. It can be correct or incorrect if the replacement methods in the change rule can replace the target method or not. A set of imperfect change rules, TR_i, represents a set of change rules mixing correct and incorrect replacement methods. Usually, the change rules generated by framework API evolution approaches are imperfect. A set of all-correct change rules, TR_c, includes change rules manually verified to be correct, *i.e.*, to provide correct replacement method(s) to a given target methods. We do not inform subjects that the change rules are imperfect or all-correct before performing the experiment. Because the change rules are randomly selected, we do not control the relations between the target methods and the type of change rules.

To build TR_c, TR_i, and TR_n, we first generate the change rules of the object systems with AURA (Wu et al., 2010). We choose AURA, because it is a state-of-the-art approach to framework API evolution. Our experiment studies the influence of all-correct, imperfect, or no change rules, using other tools or even building change rules manually do not affect the results of the study. Second, we randomly select five correct rules and two incorrect change rules for JEdit v4.1–v4.2 and six correct rules and one incorrect change rule for JFreeChart v0.9.11–v0.9.12 and JHotDraw v5.2–v5.3 to build the imperfect change rules, TR_i. The precision of TR_i is close to that of the state-of-the-art approaches to framework API evolution. Third, we manually correct the incorrect change rules to build TR_c and remove all the replacements from the change rules in TR_i to build TR_n. To limit the time of the experiment approximately to one hour, we perform a pre-experiment and choose 21 change rules, seven from each program.

4.1.6 Dependent Variable

The dependent variables of our experiment are the precision of the replacements that the subjects identify for the given target methods and the time that they spend on the task. We compute the precision using the equation below. For each target method, we compare the subjects' answers, *i.e.*, given set of methods, against the manually-validated set of replacement methods. Time is counted in seconds when subjects work on the tasks. Break time, if there were any, were excluded.

$$Precision = \frac{|\{Correct\ Replacements\}|}{|\{Given\ Target\ Methods\}|} \tag{4.1}$$

$$\tag{4.2}$$

4.1.7 Mitigating variables

We collected several mitigating variables that could influence the results of our experiment:

— Subjects' knowledge of the object programs;
— Subjects' knowledge of Java;
— Subjects' knowledge of Eclipse;
— Subjects' experience in software engineering;
— Learning effect;
— Fatigue;
— Experiment environment.

For the first four mitigating variables, we asked the subjects to fill a questionnaire to provide their levels of knowledge on five-point Likert scales (Likert, 1932) (bad, quite good, good, excellent or expert). Their knowledge levels are shown in Table 4.2. We provide a tutorial about the functions related to our experiment to minimize the influence of the knowledge of Eclipse. We verify if there is any correlation between the dependent variables and the four mitigating variables during the result analysis. We limit learning effect and fatigue with a randomized complete block design (Wohlin et al., 1999). To minimize the influence of the environment, we require that the subjects complete the tasks in a quiet environment.

Table 4.2 Subjects' Knowledge Levels

KnowledgeLevel	Topics					
	SE	Java	Eclipse	JHotdraw	JEdit	JFreechart
Bad	0	1	1	21	22	20
Quite Good	1	5	10	7	7	6
Good	18	19	16	3	2	3
Excellent	8	5	3	0	0	2
Expert	4	1	1	0	0	0

4.1.8 Hypotheses

The null hypotheses of our experiment are that there is no difference between the precision of the replacements of the target methods found by the subjects and the time that they spend with the help of all-correct, imperfect, and no change rules. For example:

— H_{p1}: There is no difference between the precision of the replacements of the target methods found by the subjects without change rules and with imperfect change rules.

The completed null hypotheses are in Table 4.3.

4.2 Study Execution

4.2.1 Experiment Web Site

To conduct the experiment and abstract the presentation of the change rules from a particular tool format, we develop a dedicated Web site. The screenshots of the main Web pages are shown in Figure 4.1. Page 4.1(a) is the page where the subjects fill in background information. Page 4.1(b) is the main task page. For each target method, the page displays

Table 4.3 Null Hypotheses

H_{p1}	There is no	precision of the replacement methods found by the subjects	without change rule	and	with imperfect change rules.
H_{p2}			without change rule	and	with correct change rules.
H_{p3}	difference		with imperfect change rules	and	with correct change rules.
H_{t1}		time spent by the subjects	without change rule	and	with imperfect change rules.
H_{t2}	between the		without change rule	and	with correct change rules.
H_{t3}			with imperfect change rules	and	with correct change rules.

the target method on the left (*e.g.*, `TextFigure.disconnect()`) and a set of candidate replacement methods, *i.e.*, obtained from a change rule, on the right-bottom corner (*e.g.*, `TextFigure.disconnect(Figure)`). The right-top box is for subjects to enter their answers.

The gathering of information from the subjects, such as background information and experiment question answers, occurs in a Web browser. All the data that subjects enter and the time that they spend on each question are stored in a database. The subjects do not need paper or writing.

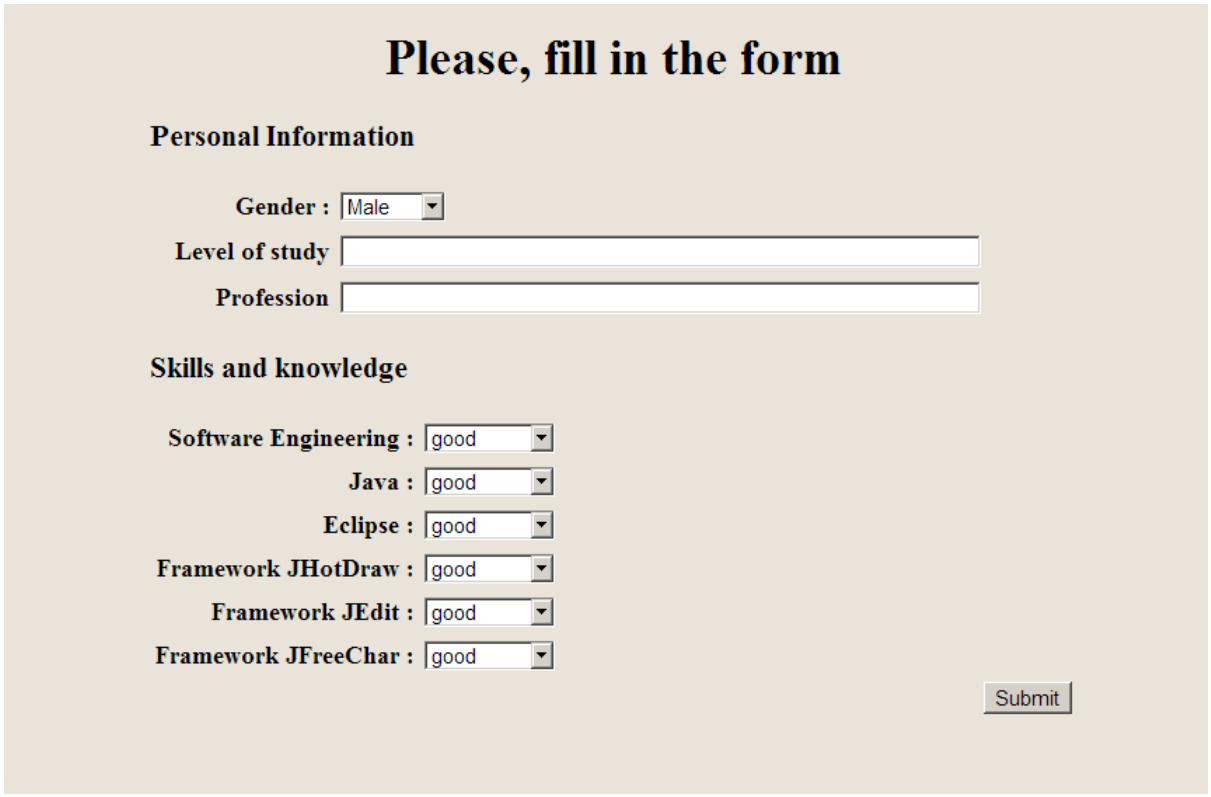
4.2.2 Experiment Workspace

Besides using the experiment Web site to read the questions, the target and suggested replacement methods (if any), and to enter their answers, the subjects had also to explore the source code of the object programs to find the replacement methods. We provided an Eclipse workspace to each subject with the source code of JEdit v4.1–v4.2, JFreeChart v0.9.11–v0.9.12, and JHotDraw v5.2–v5.3, as shown in Figure 4.2 and available online[4]. With Eclipse, the subjects could check the definitions and usage context of the target methods, read the source code of both versions of the frameworks, and identify the replacements.

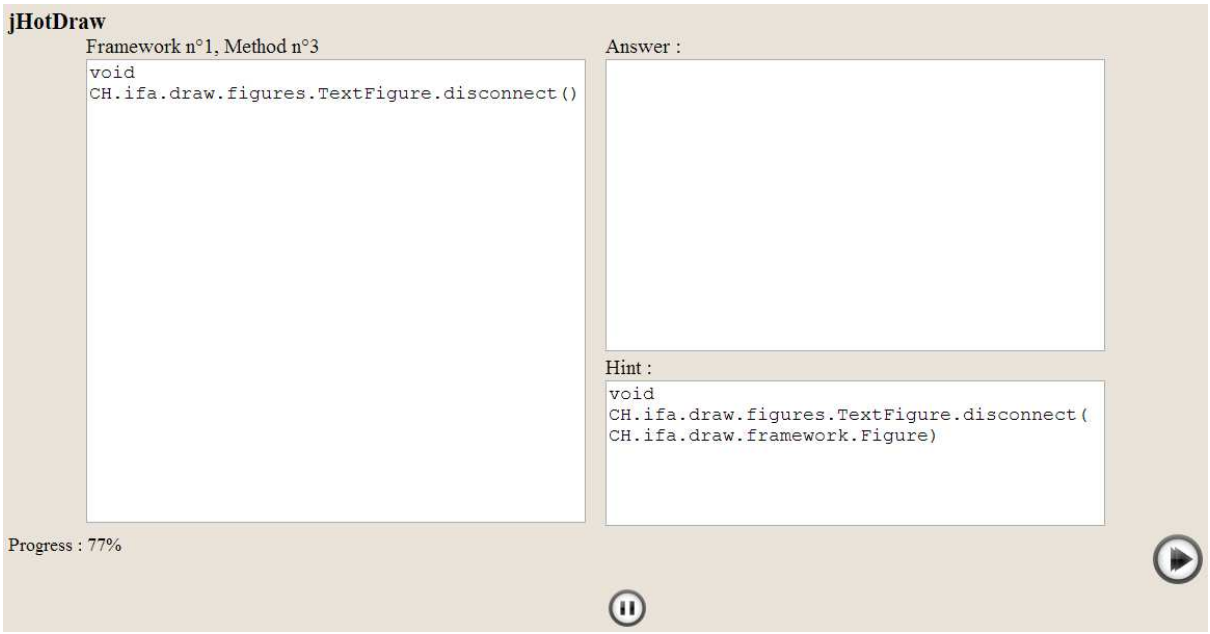
4.2.3 Experiment Process

For each program, the subjects had to find the replacements for the seven target methods with all-correct, imperfect, or no change rules.

At the beginning of the experiment, each subject received two tutorials in PDF format. One tutorial explains the procedure of the experiment without revealing the purpose of our study: what subjects should do and how they fill in the answers of the questions. The other tutorial was about the use Eclipse to explore the source code of the frameworks, in case some subjects were not familiar with it. We asked the subjects to read these two documents



(a) Background Information



(b) Question

Figure 4.1 Web Site

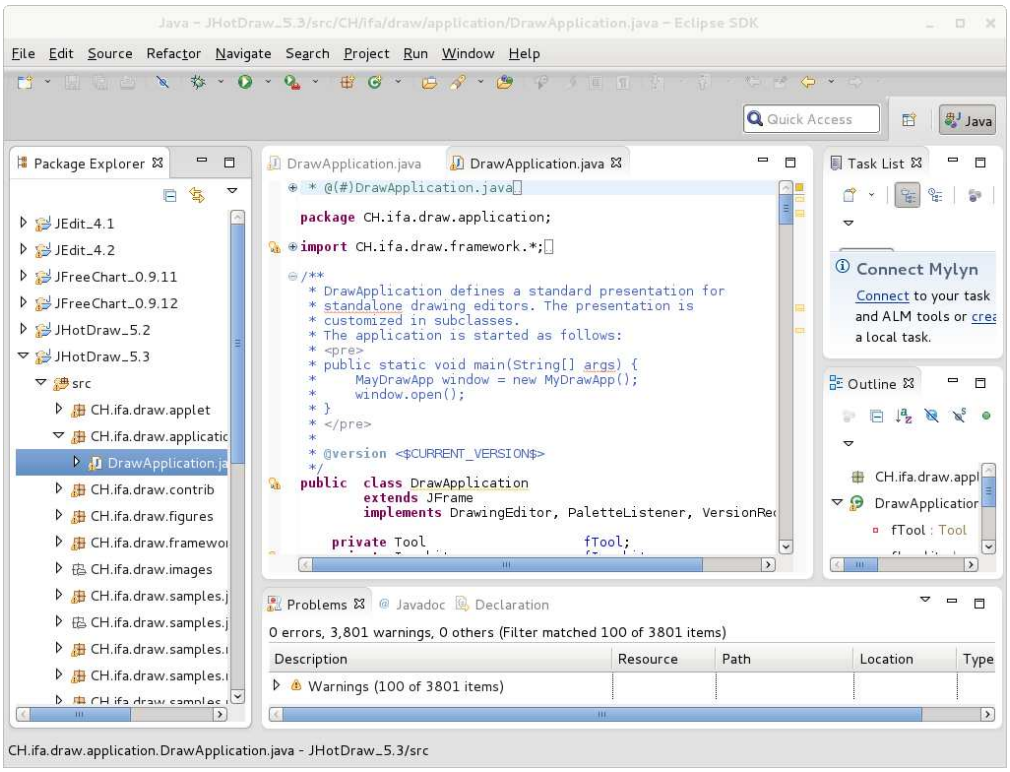


Figure 4.2 Source Code Workspace

carefully before the experiment. If the subjects had any questions, they could ask us in person if they performed the experiment in our laboratory or by email if they did it remotely.

When the subjects were familiarized with the experiment and Eclipse, we administered a pre-experiment questionnaire to collect their gender, level of study, profession, knowledge in software engineering, Eclipse, Java, and the frameworks used in the study, using the background information page of our Web site.

Then, they completed the tasks by exploring the source code of the frameworks, looking for the replacement methods of each target method displayed on the question pages. According to the treatment, they had either all-correct or imperfect replacements methods or no method at all to help them. When they found the replacement method(s), they copied the qualified names of the replacement methods from Eclipse or from the box at the bottom-right corner of the question pages and pasted them into the answer box on each question page. Eclipse provides a context menu to copy the qualified names of methods easily and our tutorial explained how to use this feature. If the subjects thought that the target method was simply deleted, *i.e.*, without any replacement, they could fill in “null" in the answer box.

4.2.4 Analysis Method

We compared the subjects' answers with the all-correct rules and classified those as wrong if they were different from the all-correct rules.

We tested our hypotheses using Kruskal-Wallis test (Wohlin et al., 1999), which is applicable for non-parametric randomized, complete block designs. The hypotheses testing results are presented as Table 4.4. We chose Kruskal-Wallis test because we did not have to make any assumption on the distribution of the data collected during the experiment. The regular α value of single comparison Kruskal-Wallis test was 0.05. Our experiment had three object programs for each treatment. Therefore, we adjusted the α value to 0.01 according to the Bonferroni correction (Miller, 1981). If the p-value of Kruskal-Wallis tests was smaller than 0.01, the results with different treatments were statistically different. We also computed the Cliff's d (Grissom and Kim, 2005) of the results with different treatments to evaluate the effect size of their differences. Cliff's Delta is also a non-parametric and it does not require any knowledge of the distribution of the data. The effect size is small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$, respectively. We computed p-values and Cliff's Deltas of the results on all three frameworks and each framework separately to distinguish the potential effect of one particular framework.

We always applied the statistical tests on the average precision values of the subject's answers and average times spent by the subjects for subjects using two different treatments, for example subjects provided with all-correct change rules and subjects provided with no change rules. In the next section, for the sake of simplicity, we talk about precision and time to mean average precision values and average time spent by subjects.

4.3 Study Results

We now report the collected data and discuss the data analysis and the results of hypothesis testing. First, we report the general results on the precision of the subjects' answers and the time that they spent. Then, we discuss the results on the three frameworks separately.

4.3.1 Overall Data Analysis

Precision The distribution of the data on precision is shown in Figure 4.3. On average, the precision value of the answers of the subjects with all-correct change rules is the highest with the value of 82%. The next is that with imperfect change rules with the value of 71%. The precision of the answers without change rule is the lowest with the value of 57%. In Table

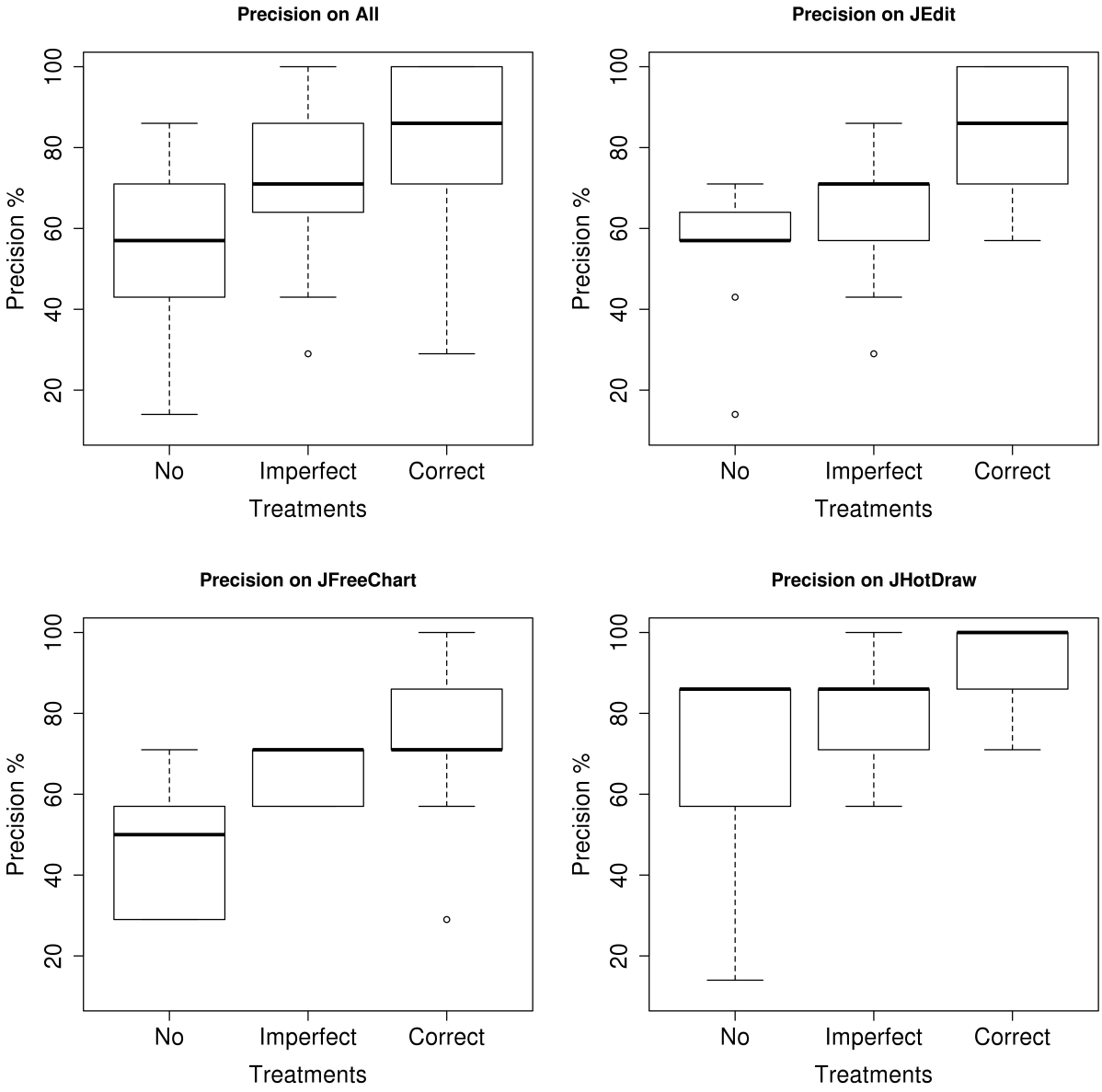


Figure 4.3 Boxplots for Precision

4.4, the hypothesis testing results show that there are statistically significant differences in the precision of the subjects' answers when they used all-correct, imperfect, and no change rules. In the third column, the value of Cliff Delta is moderate between the subjects with imperfect and all-correct rules and it is large between those with imperfect and no change rules. These results support the expectation that even imperfect change rules help developer find the correct replacements.

We conclude that the subjects with all-correct change rules could find the replacements of the target methods more accurately than the subjects with the other two treatments. The subjects with imperfect change rules did not perform as well as the former, but significantly better than the subjects without change rule. However, even with all-correct change rules, the subjects still could not correctly answer all the questions. This last observation is evidence of the difficulty of framework API evolution. We further discuss this observation in Section 4.4.

Time Figure 4.4 shows the distribution of the times that subjects spent with the tree treatments. The subjects spent almost the same time to find the replacement methods with all-correct, imperfect, or no change rules. The subjects with imperfect change rules spent less time than the subjects with the other two treatments. On average, they spent 23 minutes (1,338 seconds) while the subjects with all-correct change rules and without change rule used 24 minutes (1,413 seconds) and 25 minutes (1,479 seconds), respectively. However, the hypothesis testing results do not confirm any statistically significance difference between the times spent with each treatment, as shown in Table 4.5.

Discussion We examined the outliers in precision and time. There were two subjects with precision of 14% on JHotDraw and JEdit, respectively, and three spent more than 46 minutes (2,800 seconds) on a single program: one on JHotdraw and two on JFreechart. There was no common subject between the two cases in precision and three cases in time. The results of these five subjects were "normal" when they worked on the other programs. Four of the five cases were on the first program that they analyzed in the experiment. We suspect that these outlier cases were caused by these subjects being not really familiar with the experiment tasks and tools. The other outlier case (more time on JFreechart) was on the third program. Probably, the subject productivity was compromised by tiredness.

4.3.2 Data Analysis per System

The results presented above show that the change rules helped the subjects find the replacements of the target methods more accurately but not faster than without them. We also

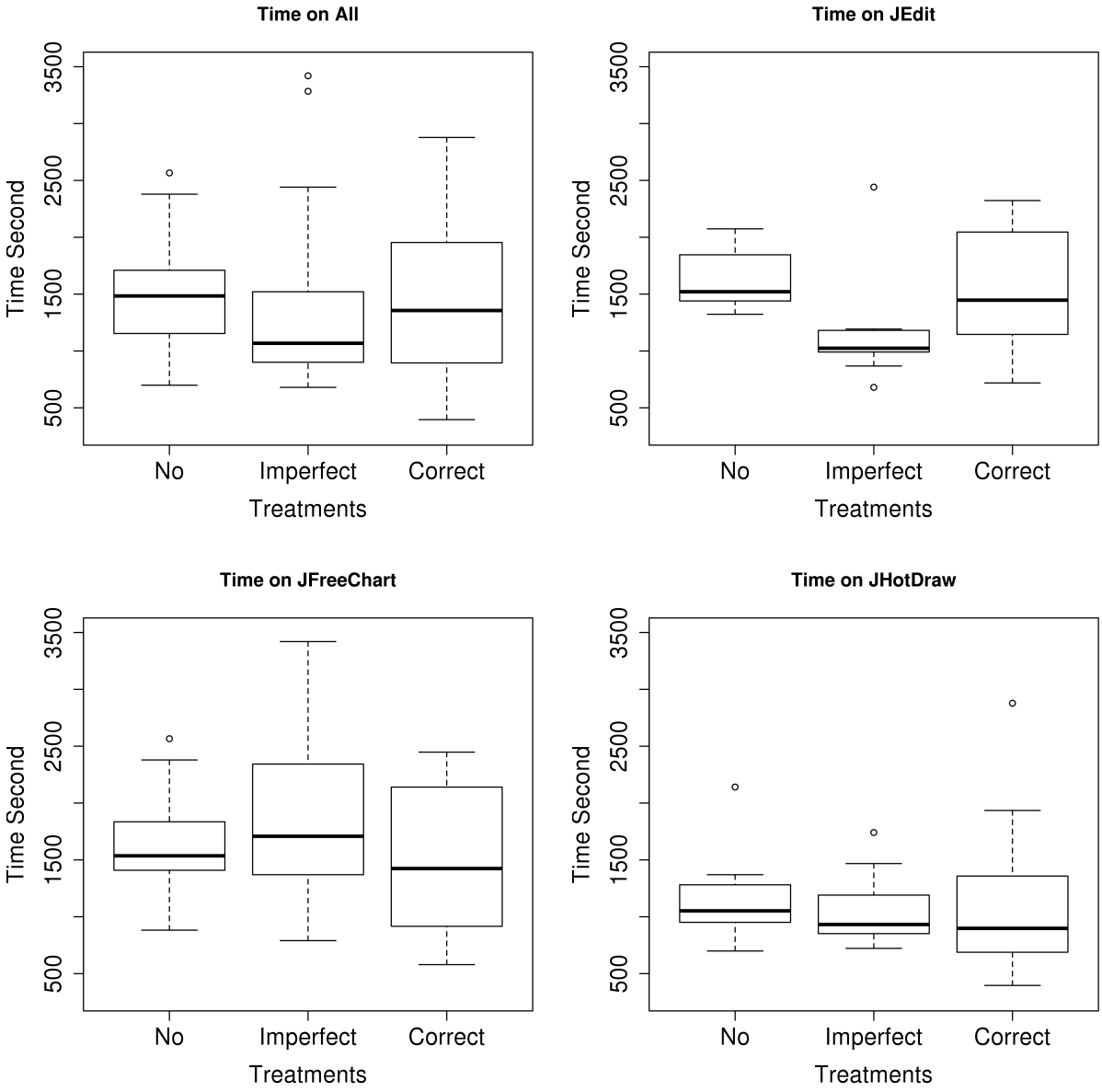


Figure 4.4 Boxplots for Time

Table 4.4 Hypothesis Testing Results for Precision

Treatements	Kruskal-Wallis p-value	Cliff's Delta
No-Imperfect-Correct	< 0.01 (4.496e-06)	N/A
No-Imperfect	< 0.01 (0.006293)	Large (0.619)
No-Correct	< 0.01 (5.269e-06)	Large (1.325)
Imperfect-Correct	< 0.01 (0.002763)	Moderate (0.443)

Table 4.5 Hypothesis Testing Results for Time

Treatements	Kruskal-Wallis p-value	Cliff's Delta
No-Imperfect-Correct	> 0.01 (0.1378)	N/A
No-Imperfect	> 0.01 (0.02611)	N/A
No-Correct	> 0.01 (0.3348)	N/A
Imperfect-Correct	> 0.01 (0.6272)	N/A

want to investigate if this conclusion applies to all three frameworks. Thus, we compare the results on different framework individually to see if the change rules helped similarly.

JHotDraw

Precision For JHotDraw, the boxplot in Figure 4.3 shows that the precision of the subjects' answers with imperfect change rules was better than that of the subjects without change rule and the precision of the subjects answers with all-correct change rules was the best, but the differences between the precision of subjects' answers with the three treatments are not statistically significant, according to the hypothesis testing results in Table 4.6.

Table 4.6 Hypothesis Testing Results for Precision on JHotDraw

Treatements	Kruskal-Wallis p-value	Cliff's Delta
No-Imperfect-Correct	> 0.01 (0.01738)	N/A
No-Imperfect	> 0.01 (0.5376)	N/A
No-Correct	> 0.01 (0.01008)	N/A
Imperfect-Correct	> 0.01 (0.02132)	N/A

Time Regarding the time that subjects spent on JHotDraw, Figure 4.4 shows that the average values of the subjects' time with the three treatments were slightly different. Similar to the general result on time, the hypothesis testing results (Table 4.7) show that the differences are not statistically significant. The change rules did not help to find the replacement methods faster or slower, while adapting to the new release of JHotDraw.

Discussion JHotDraw is a program developed by Gamma *et al.* to demonstrate the application of design patterns (Gamma et al., 1995). It was elegantly designed and consistently coded. When we studied its v5.2 and v5.3, we found that it is very straightforward to navigate and understand its source code. Between JHotDraw v5.2 and v5.3, the change rules helped subjects, but it did not take much more time for subjects to complete the task without them. Indeed, the hypothesis testing results on JHotDraw show that there is no statistically significant differences between the subjects with the three treatments.

JFreeChart

Precision The change rules helped subjects to find replacements for the target methods of JFreeChart more accurately. The boxplot in Figure 4.3 shows that the precision of the subjects' answers with correct change rules was the best and the precision of the subjects' answers with imperfect change rules is better than that without change rule. There is much less overlap between the precision values of the subjects' answers with the three treatments than that on JHotDraw and JEdit. The hypothesis testing results confirm that the differences between the subjects' answers with no vs. all-correct and no vs. imperfect change rules are statistically significant with large effect size and the difference between the subjects' answers with imperfect and all-correct change rules is not significant.

Table 4.7 Hypothesis Testing Results for Time on JHotDraw

Treatements	Kruskal-Wallis p-value	Cliff's Delta
No-Imperfect-Correct	> 0.01 (0.3632)	N/A
No-Imperfect	> 0.01 (0.2353)	N/A
No-Correct	> 0.01 (0.2207)	N/A
Imperfect-Correct	> 0.01 (0.7223)	N/A

Table 4.8 Hypothesis Testing Results for Precision on JFreeChart

Treatements	Kruskal-Wallis p-value	Cliff's Delta
No-Imperfect-Correct	< 0.01 (0.001069)	N/A
No-Imperfect	< 0.01 (0.004354)	Large (1.60919)
No-Correct	< 0.01 (0.001702)	Large (1.80291)
Imperfect-Correct	> 0.01 (0.08238)	N/A

Time Similar to JHotDraw, Figure 4.4 shows that the average values of the times spent by the subjects with the three treatments are only slightly different. The subjects with imperfect change rules have the largest value while the subjects with all-correct change rules have the smallest. The hypothesis testing results in Table 4.9 show that there are no significant differences. The change rules did not help save time to find the replacement methods while upgrading to the new release of JFreeChart.

Discussion JFreeChart is a Java chart library to generate different types of charts. Between v0.9.11 and v0.9.12, there were many methods with similar names in the two versions, such as `DefaultBoxAndWhiskerCategoryDataset.getMedianValue(int, int)` and `DefaultBoxAndWhiskerXYDataset.getMedianValue(int, int)`. It would be time-consuming

Table 4.9 Hypothesis Testing Results for Time on JFreeChart

Treatements	Kruskal-Wallis p-value	Cliff's Delta
No-Imperfect-Correct	> 0.01 (0.5209)	N/A
No-Imperfect	> 0.01 (0.5676)	N/A
No-Correct	> 0.01 (0.4288)	N/A
Imperfect-Correct	> 0.01 (0.3198)	N/A

to go over all of these similar methods to find possible replacement methods and verify them. Therefore, with the change rules, subjects used much less time to find the replacements than those without them.

Because of the differences between a real context and our experiment, the subjects without change rules spent almost the same time as the subjects with imperfect and all-correct change rules, but with lower precision in their answers. Although we asked them to take as much time as they needed, it seems that they just spent the same effort as the subjects with the other treatments and did not verify their answers thoroughly. The hypothesis testing results show that the precision of the subjects' answers with imperfect and all-correct change rules are statistically better than that without change rules, while the time that they spent is not significantly different.

JEdit

Precision Similar to the precision for JFreeChart, the boxplot in Figure 4.3 shows that the average value of precision of the subjects' answers with imperfect change rules is better than that of the subjects without change rule and that the precision of the subjects' answers with all-correct change rules is the best. In Table 4.10, the hypothesis testing results of the precision for JEdit shows that there is no statistically significant difference between the subjects' answers with imperfect and no change rules while the difference between the subjects' answers with all-correct and no change rules is statistically significant with large effective size.

Table 4.10 Hypothesis Testing Results for Precision on JEdit

Treatements	Kruskal-Wallis p-value	Cliff's Delta
No-Imperfect-Correct	< 0.01 (0.00329)	N/A
No-Imperfect	> 0.01 (0.1959)	N/A
No-Correct	< 0.01 (0.001198)	Large (1.98641)
Imperfect-Correct	> 0.01 (0.02876)	N/A

Time The distribution of the times that subjects spent on JEdit is different than those on JHotDraw and JFreeChart as shown in Figure 4.4. First, the subjects with imperfect change rules used the least time to answer. Second, the times spent by the subjects with the three treatments were very different. The hypothesis testing results (Table 4.11) confirm that only the difference between times spent by the subjects with imperfect and no change rules is significant with a large effect size. The differences between the times spent by the subjects with imperfect and all-correct change rules and with all-correct and no change rules are not statistically significant.

Discussion JEdit is a free text editor supporting plugins and syntax highlighting for more than 200 programming languages. When we analysed JEdit, we found that the implementations of v4.1 and v4.2 changed dramatically. The major differences between the two versions make the upgrading process more difficult for the subjects. The precision between the subjects with all-correct and no change rules are significantly different, but not between the subjects with imperfect and no change rules. However, the times that subjects spent with imperfect and no change rules are statistically different. The subjects with imperfect change rules spent much less time than the subject without change rule with similar precision in their answers. So, the change rules did save some effort for JEdit.

One result on JEdit that we did not expect is that the average time spent by the subjects with all-correct change rules was more than that for the subjects with imperfect change rules. To explain this observation, we first checked if there was any correlation between the results and the mitigating variables, but the answer was negative. Then, we checked if the ten subjects with all-correct change rules on JEdit also spent more time than the other subjects

Table 4.11 Hypothesis Testing Results for Time on JEdit

Treatements	Kruskal-Wallis p-value	Cliff's Delta
No-Imperfect-Correct	< 0.01 (0.008001)	N/A
No-Imperfect	< 0.01 (0.001939)	Large (0.7322)
No-Correct	> 0.01 (0.4812)	N/A
Imperfect-Correct	> 0.01 (0.04125)	N/A

when they worked on JHotDraw and JFreechart with imperfect and no change rules. We found that these subjects did spend 39% more time than the others, on average. We applied Kruskal-Wallis test to the time spent by these ten subjects and others on other tasks. The p-values are 0.20, 0.20, 0.05 and 0.83 on JHotdraw with imperfect and no change rules and JFreechart with imperfect and no change rules, respectively. These results show that, in three of the four other tasks, these ten subjects spent more time than the other subjects with more the 80% confidence. The only exception was when they worked on JFreechart with no change rule. We suspect that all these ten subjects reached the upper bound of the time that they could spend on one program, because JFreechart's code is more difficult to comprehend.

4.3.3 Summary

Based on the results of the experiment and the statistical analyses, we answer the research questions as follows:

RQ1: Is there a difference between the precision of the replacements of the target methods found by the subjects with all-correct, imperfect, and no change rules?

Answer: Yes, the precision values of the subjects' answers with the help of all-correct, imperfect, and no change rules shows statistically significant differences. The subjects with all-correct change rules have the higher value, the next is with imperfect change rules, and the subjects without change rules have the lowest value. The effect size of the differences in precision values between the subjects with no and imperfect change rules is large and that between the subjects with imperfect and all-correct change rules is moderate.

RQ2: Is there a difference between the times that the subjects spend with all-correct, imperfect, and no change rules?

Answer: No, there is no significant difference between the times spent by the subjects with all-correct, imperfect, and no change rules.

These results show that the change rules generated by framework API evolution approaches are useful. Yet, different to upgrading to new releases of frameworks in a real context, the subjects could only spend limited times on the tasks, no matter how serious they were. So, they could give wrong answers. In a real context, the change rules will help developers to adapt their client code to new releases of frameworks faster, because they must work on the upgrading until they have 100% precision. As we discussed after presenting the results on each object program, the effect of the change rules can vary on a specific framework.

4.4 Discussion

In this section, we discuss the influence of the mitigating variables and other issues impacting the results of our study.

4.4.1 Mitigating Variables

For mitigating variables like fatigue and learning effect, we used a randomized, complete block design to minimize their influences. There are five other mitigating variables whose influence we could not neutralize: gender, degree, knowledge of software engineering, Java, and Eclipse, because we did not have this information until the subjects performed the experiment. Therefore, we verified if there was any correlation between the precision of the subjects' answers (the times spent by the subjects) and the five mitigating variables, using permutations tests. A permutation test (Baker, 1995) is non-parametric and does not require normal data distribution.

We also consider gender as a mitigating variable, because the differences between male and female in problem solving activities have been studied before (Beckwith et al., 2005; Meyers-Levy, 1989; O'Donnell and Johnson, 2001; Sharafi et al., 2012). We want to see if the results of our experiment are correlated with gender.

Tables 4.12 to 4.16 show that the precisions of the subjects' answers are not correlated to the five mitigating variables. (The times that the subjects spent are not correlated to the five mitigating variables either and the results can be found online [5].)

5. http://www.ptidej.net/download/experiments/emse13a

Table 4.12 Precision: Two-way Permutation Test by Change Rule Type and Knowledge in Software Engineering

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	2	9961.83	4980.91	500000.00	<0.01
SE	1	18.39	18.39	232090.00	0.81
Treatment: SE	2	1042.32	521.16	500000.00	0.20
Residuals	87	27276.64	313.52		

Table 4.13 Precision: Two-way Permutation Test by Change Rule Type and Knowledge in Java

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	2	9961.83	4980.91	500000.00	<0.01
Java	1	64.57	64.57	500000.00	0.64
Treatment: Java	2	83.80	41.90	408504.00	0.88
Residuals	87	28188.98	324.01		

4.4.2 NASA Task Load Index

Besides the two objective measurements of subjects' performance (precision and time), we also asked the subjects to fill in the NASA Task Load Index (NASA-TLX) (Hart and Staveland, 1988) after each program to give their subjective evaluation of the effort to complete the tasks. The NASA-TLX is a subjective measurement of subjects' workload in human–machine environments, using six sub-scales: Mental Demands, Physical Demands, Temporal Demands, Own Performance, Effort and Frustration, as shown in Figure 4.5. Each sub-scale is measured from 1 to 20. Level 1 represents the lightest or the best while level 20 means heaviest or worst. We computed the average value of the six sub-scales as the overall NASA-TLX value Hart and Staveland (1988).

Because the NASA-TLX depends on the subjects' personal feeling, we did not use it as a dependent variable, but it is interesting to see the relations between NASA-TLX value and the two objective measurements of subject performance. The distribution of the NASA-TLX values (see Figure 4.6) and the hypothesis testing results on NASA-TLX values (see Table 4.17) show that there is no statistical difference between the subjects with all-correct, imperfect, and no change rules. We argue that the main reason for observing statistically-similar NASA-TLX values is that the subjects' feelings vary dramatically between subjects.

We cross-referenced the NASA-TLX values and the times that the subjects spent. For the same NASA-TLX value, the times could be very different. For example, 11 subjects gave a NASA-TLX value of 7 for their tasks, but the times that they spent varied between 14

Table 4.14 Precision: Two-way Permutation Test by Change Rule Type and Knowledge in Eclipse

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	2	9961.83	4980.91	500000.00	<0.01
Eclipse	1	55.64	55.64	469963.00	0.68
Treatment: Eclipse	2	291.67	145.84	500000.00	0.64
Residuals	87	27990.04	321.72		

Table 4.15 Precision: Two-way Permutation Test by Change Rule Type and Gender

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	2	9545.85	4772.92	500000.00	<0.01
Gender	1	582.55	582.55	500000.00	0.17
Treatment: Gender	2	156.21	78.11	500000.00	0.78
Residuals	87	27598.60	317.23		

minutes (849 seconds) and 40 minutes (2,379 seconds). Similarly, we observed a subject with a NASA-TLX value of 2 who spent 40 minutes (2,378 seconds) on the tasks.

We suspect that the randomized, complete block design of the experiment amplified the variance of the subjects' feelings regarding their workload. If we had used only one object program, the subjects could have reported consistent feelings between the treatments. However, an experimental design with one object program would have been a serious threat to the generalizability of the results. The lesson learned is that it is risky to use only subjective measurements as dependent variables in experiments. Subjects may choose different values for similar personal feeling on same tasks, especially when the experiment design is complex.

4.4.3 Change Rule Types

The change rules can be classified in different ways. While considering code element changes, the change rules can be return type , package , class, method, and parameter changes. The change rules used in the experiment cover all these code element change types. While considering the mapping between changed API and its replacements, the types of the change rules include one-replaced-by-one, one-replaced-by-many, many-replaced-by-one, and simply-deleted. In our experiment, besides one-replaced-one change rules as illustrated in Section 4.1.3, the randomly selected change rules in our experiment also include one-replaced-by-many and simply-deleted rules. For the target methods of one-replaced-by-many change rules, the answers of subjects are correct only when they find all the replacements. Subjects can answer "null" as replacement when they think that the target methods are simply-deleted.

Table 4.16 Precision: Two-way Permutation Test by Change Rule Type and Degrees

	Df	R Sum Sq	R Mean Sq	Iter	Pr(Prob)
Treatment	2	9956.99	4978.49	500000.00	<0.01
Degree	2	91.86	45.93	500000.00	0.86
Treatment: Degree	4	2014.42	503.60	500000.00	0.17
Residuals	84	26231.07	312.27		

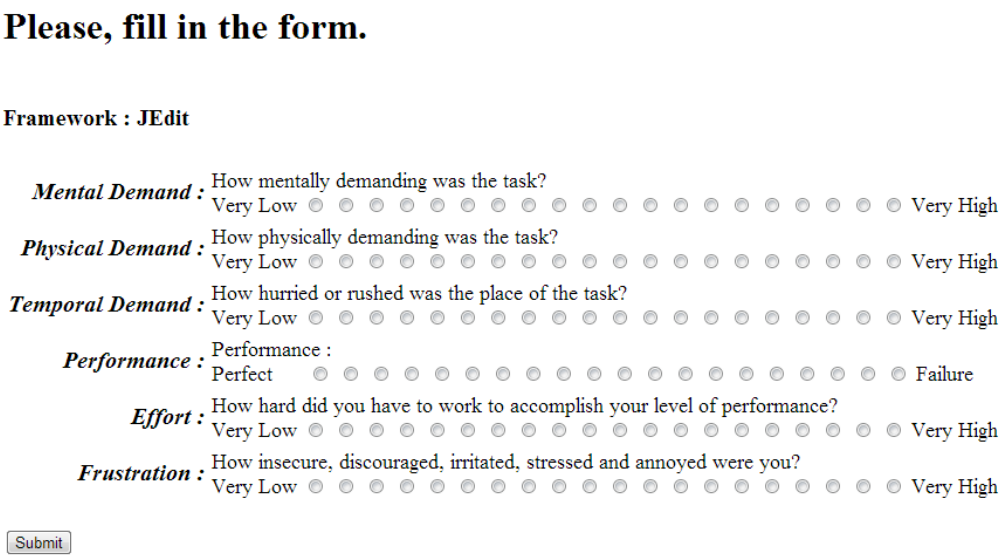
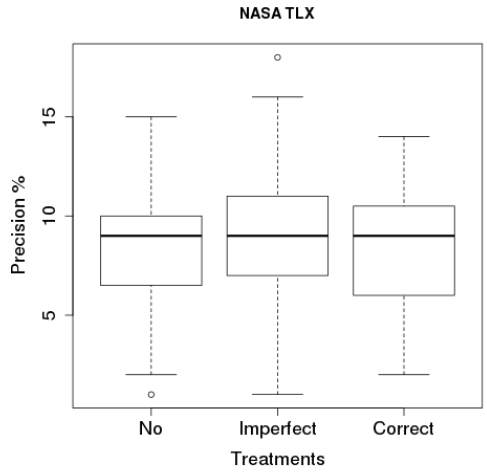


Figure 4.5 NASA-TLX

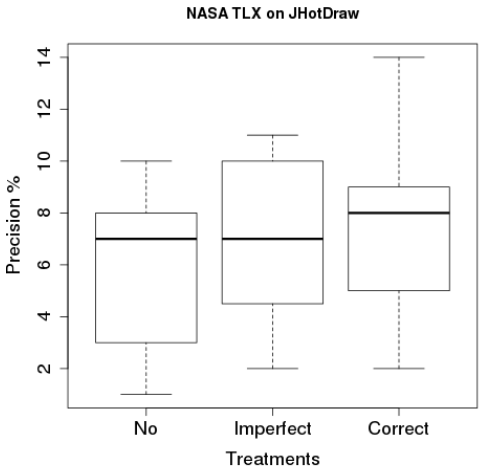
The goal of our experiment is to investigate the influence of all-correct and imperfect change rules in comparison to no change rules. The influence of different types of change rules is an interesting future study.

4.4.4 Skeptical Subjects

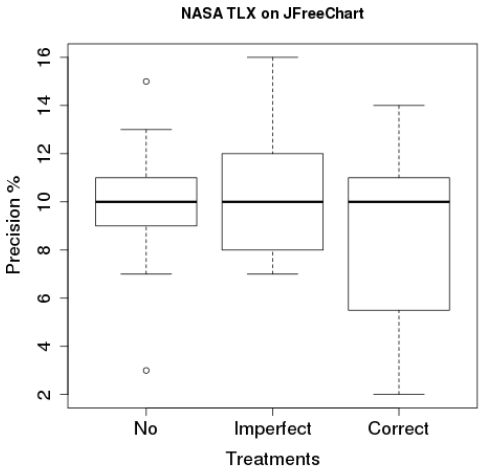
The results of the experiment show that the subjects were serious and skeptical about the provided change rules. Because they do not know a priori, even with the correct change rules, most of the subjects spent almost the same time as the subjects without them to find the replacement methods. The hypothesis testing results on time confirm this conclusion statistically. Moreover, the average precision of the subjects' answers with all correct change rules is just 82%. Even with correct change rules, the subjects did correctly answer all the questions during the time that they spent for our experiment even though we did not give any time limitation to the subjects. This observation is a further evidence of the difficulty of framework API evolution.



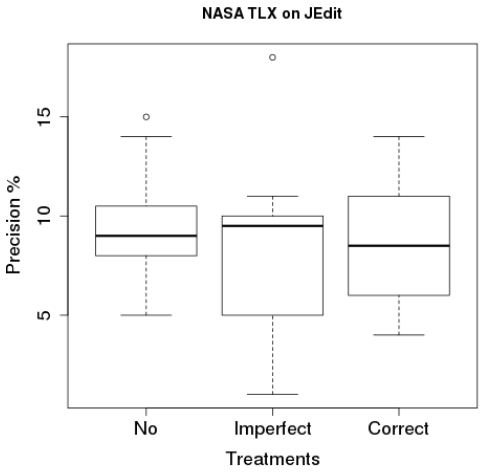
(a) General



(b) On JHotDraw



(c) On JFreeChart



(d) On JEdit

Figure 4.6 Boxplots for NASA-TLX

Table 4.17 Hypothesis Testing Results for NASA-TLX

Treatements	Kruskal-Wallis p-value			
	General	JHotDraw	JFreeChart	JEdiit
No-Correct-Imperfect	> 0.01 (0.978)	> 0.01 (0.6182)	> 0.01 (0.6777)	> 0.01 (0.8287)
No-Imperfect	> 0.01 (0.9379)	> 0.01 (0.507)	> 0.01 (1)	> 0.01 (0.7228)
No-Correct	> 0.01 (0.9211)	> 0.01 (0.3443)	> 0.01 (0.4208)	> 0.01 (0.4989)
Imperfect-Correct	> 0.01 (0.8209)	> 0.01 (0.7191)	> 0.01 (0.4954)	> 0.01 (1)

4.4.5 Experiment vs. Real Tasks

The differences between our experiment and framework upgrading tasks in a real context are caused by the goal of this study, which focuses only on a part of the framework upgrading tasks. In the experiment, the subjects are volunteers. No matter how serious they are, they can only spend a limited amount of time on the experiment. We cannot expect all of them to be able or willing to finish all the tasks correctly if they take too long. In a real context, developers first would try to find the replacements of missing methods. If they could not find any proper replacements, then they would need to find a work-around. Finally, whether with replacements or work-arounds, they would test their programs after changes. If the test passes, then the upgrading task would be done, else they would keep working on it or they would stay with the previous release of the framework.

Our study focuses on the impact of imperfect change rules on framework API evolution identification. We want to investigate if the imperfect change rules still help developers find the replacements of missing methods caused by software evolution, such as class or parameter changes. It does not cover finding work-arounds and testing. If we had asked the subjects to perform real framework upgrading tasks, finding work-arounds and testing would have increased the precision and the time spent. However, following previous works on change-rules for framework APIs (Dagenais and Robillard, 2011; Kim et al., 2007; Meng et al., 2012; Schäfer et al., 2008; Wu et al., 2010; Xing and Stroulia, 2007b), we assume that providing more accurate replacement methods can only help developers by reducing (1) the time spent understanding the new framework, (2) the time spent performing the changes necessary to adapt their programs, and (3) the overall time spent in the cycle {search-for-replacement, change, test}. Yet, identifying the replacements is only one step of the framework upgrading

process and the influence of API change rules on other steps has not been investigated. Therefore, in addition to the threats to the validity to our experiment discussed in Section 4.4.7, it is possible that other factors impact the accuracy and time of the overall cycle and future work is necessary (1) to study these factors in details, (2) to understand how time is spread throughout the cycle, and (3) to assess the impact of the change rules (and of their accuracy) on the whole cycle.

We argue that finding replacements for missing methods is a program comprehension task and, consequently, we designed our experiment as other program comprehension experiments, asking the subjects to answer questions based on their understanding of the frameworks. In our experiment, the subjects were volunteers. No matter how serious they were, they could only spend a limited amount of time on the experiment. We could not ask them to perform real framework upgrading tasks, because they could take very long time. Therefore, subjects may have given answers even if they were not 100% confident. Although we asked the subjects to take as much time as they needed, the time that the subjects spent ranges from 40 to 120 minutes and the precision of their answers lies between 14% and 100%. The average times for choosing a replacement method are 24, 23, and 25 minutes (1,413, 1,338, and 1,479 seconds). The times spent by the subjects with all-correct, imperfect and no change rules are not statistically different. These times are quite short and likely to be shorter than the times necessary to test the changes. On the one hand, such times show the importance of using API change rules to reduce the overall time of choosing and testing a change. On the other hand, such times do not show the total times needed to complete the changes to satisfy the tests in the absence of change rules, which may be longer than the average times with no change rules, because developers must find a work-around and not just provide a possible replacement, as in our experiment.

There is another difference between a real context and our experiment. In a real context, developers must have client code to test the replacements. In our experiment, we did not provide such client code for two reasons. First, the test code could have helped the subjects to verify if the replacements were correct, but it cannot help developers find the correct replacements. Second, there could have been extra effort for the subjects to make the client code executable using the replacements. For example, if there was a new parameter in a replacement, to find out the proper value for that parameter could have taken time too. Such effort cannot be saved by change rules and we excluded it from our experiment.

4.4.6 Subjects vs. Professional Developers

In our experiment, most subjects are students. From their answers, we can see that they understood the frameworks well because most of their answers are correct. Still, they are different from professional developers. We can see subjects as inexperienced developers. Change rules may help experienced and inexperienced developers in different ways. For experienced developers, their experience can help them to distinguish the correct and wrong change rules. So, change rules let them quickly focus on "difficult" changes. For inexperienced developers, change rules can reduce the general comprehension effort by guiding them to a relatively small part of the frameworks. Only further studies with professional and expert subjects could help identify differences, if any.

4.4.7 Threats to Validity

Construct Validity Construct validity verifies that the observation really reflects the theory, *i.e.*, if the treatment reflects the cause and the outcome reflects the effect. We wanted to evaluate if change rules help subjects find the replacements of target methods more accurately and faster. We used correct, imperfect, and no change rules as the treatments. We used the precision of the subjects' answers and the time that they spent as the outcomes. Some mitigating variables could affect the outcomes and we minimized their influence as described in Section 4.1.7. We did not observe correlations between the mitigating variables and the outcomes in the collected data. Thus, we argue that the treatments and outcomes reflect the cause and the effect. We use our tool, AURA (Wu et al., 2010), to collect raw change rules from which we selected 21 correct change rules, verified manually. Moreover, we used a dedicated Web site to present the change rules to the subjects. Therefore, we used AURA only to make it easier for us to build the sets TR_c and TR_i. The experiment does not depend on AURA, its algorithms or the formats of its change rules. We could have used another tool but choose AURA by convenience and due to his high precision on the three object programs.

Internal Validity Internal validity verifies that the outcome is really caused by the treatment. To overcome threats to internal validity, first, we selected three object programs to avoid that the experiment results depended on the properties of a single program. Second, we used a randomized, complete block design to avoid maturation threats, such as fatigue and learning effect. Third, we did not provide feedback to the subjects' answers, so the subjects could complete subsequent tasks using the answers of previous questions. Fourth, we chose Eclipse, a popular IDE as a tool to explore the source code to avoid instrumentation

threat. Among the 31 subjects, only one reported poor skills with Eclipse. We provided a tutorial about the Eclipse features required for the experiment to help such inexperienced subjects. Fifth, because wrong answers may be due to misunderstood tasks, we provided detailed experiment instructions and asked the subjects to read them carefully before answering questions. The experiment results showed that the subjects well understood the tasks. Sixth, asking subjects for their genders could lead to anxiety and decreased performances (stereotype threat). However, our study shows that gender could not explain any differences among subjects and, therefore, that this threat does not impact the results of our study. Last, no subject participated in the development of the experiment design, documents, tasks, and Web site. We also asked the subjects not to talk about the experiment with other people before the end of the study to avoid diffusion threat.

Conclusion Validity Conclusion validity verifies that the relation between the outcome and the treatment can be proved statistically. The null hypotheses are well defined. In total, 31 subjects participated in the experiment. We applied Kruskal-Wallis test (Wohlin et al., 1999), which is proper for randomized, complete block design to verify the significance level and Cliff's Delta (Grissom and Kim, 2005) to evaluate the effect size. Both Kruskal-Wallis test and Cliff's Delta do not make any assumption on the distribution of data collected.

External Validity External validity verifies that the results of a study are generalizable. We identify two threats to external validity. The first is the interaction between selection and treatment. In our experiment, the subjects acted as software developers. Yet, because of their limited number and specific demographics, they may not represent generally software developers accurately. However, all the 31 subjects who participated in the experiment were university-level students or had a degree in computer science or software engineering. Most of them had a good knowledge in Java, Eclipse, and software engineering. Among 29 subjects, 20 of them were graduate students. Thus, we believe that they represent junior software developers. The second threat is interaction of setting and treatment. We only used three Java programs and 21 target methods in our experiment. They may not reflect the whole framework API evolution problem. Considering the popularity of frameworks in Java and the variety of the three object programs, we believe this setting is not a major threat to external validity. We also chose the 21 target methods randomly, so there was no bias for this factor in our experiment design.

4.5 Conclusion

We conduct an experiment to evaluate the precision of the replacement methods that subjects find and the time that they spend with all-correct, imperfect, and no change rules.

The dependent variables of the experiment are the precision of the replacement methods that the subjects found and the times that they spent with and without the help of change rules. We choose randomly 21 target methods as the independent variable of the experiment and defined three treatments: all-correct, imperfect, and no change rules. To limit the influence of a specific framework on the results and to control experimental time, we choose three medium-size frameworks (JHotDraw v5.2–v5.3, JFreeChart v0.9.11–v0.9.12, and JEdit v4.1–v4.2) and seven target methods for each framework. Because of the numbers of treatments and object frameworks, we use a randomized, complete block design (Wohlin et al., 1999) to minimized the number of subjects required and to overcome some threats to validity.

In total, 31 subjects participate in the experiment. In general, the statistical analysis results showed that the precision values of subjects' answers with all-correct, imperfect, and no change rules are significantly different with average values of 82%, 71% and 57%, respectively. The effect size Cliff's Delta of the differences between the precision of the subjects' answers with no and imperfect change rules is large and that between the subjects with imperfect and correct change rules is moderate. The times that the subjects spend with the three treatments to find the replacements methods are not statistically different with average values of 24, 23, and 25 minutes (1,413, 1,338, and 1,479 seconds), respectively.

In conclusion, the results of our study show that the change rules built by previous approaches are useful. The higher precision the change rules have, the better help they provide. Thus, the imperfect change rules can be used instead of unavailable documentation or as complement to partial documentation. Developers of frameworks could also use them as starting point to build upgrading documentation.

In our first study on the reality of API changes and usages, we find that missing classes and methods are the most frequent API changes in frameworks and affecting client programs. From the experiment presented in this section, we confirm that the API change rules built by previous approaches do help developers find the replacements of these missing APIs, even these change rules are not 100% correct. The more precise the change rules are, the better they help developers. Consequently, we conduct our third study to improve previous approaches to build API change rules.

Table 4.18 Task Distribution (C-Correct, I-Imperfect, N-No, JHD-JHotDraw, JFC-JFreeChart, JE-JEdit)

Order	Treatment	Subject															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	change rules	I	I	I	N	N	N	C	N	C	C	I	I	N	N	N	N
	System	JHD	JHD	JHD	JE	JFC	JE	JFC	JHD	JE	JFC	JE	JFC	JHD	JHD	JFC	JE
2	change rules	C	C	N	I	C	I	I	C	I	I	N	N	I	C	I	C
	System	JE	JFC	JFC	JHD	JHD	JFC	JE	JE	JHD	JHD	JFC	JE	JE	JFC	JHD	JFC
3	change rules	N	N	C	C	I	C	N	I	N	N	C	C	C	I	C	I
	System	JFC	JE	JE	JFC	JE	JHD	JHD	JFC	JFC	JE	JHD	JHD	JFC	JE	JE	JHD
Order	Treatment	Subject															
		17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
1	change rules	I	C	N	I	C	C	C	N	C	N	C	C	C	I	I	
	System	JFC	JHD	JHD	JFC	JE	JFC	JHD	JHD	JE	JFC	JFC	JHD	JHD	JE	JFC	
2	change rules	N	N	C	N	N	N	I	C	N	I	N	I	I	N	N	
	System	JE	JE	JFC	JHD	JFC	JE	JE	JFC	JHD	JHD	JE	JE	JFC	JHD	JHD	
3	change rules	C	I	I	C	I	I	N	I	I	C	I	N	N	C	C	
	System	JHD	JFC	JE	JE	JHD	JHD	JFC	JE	JFC	JE	JHD	JFC	JE	JFC	JE	

CHAPTER 5 EMPIRICAL STUDY ON FEATURE USAGES IN API CHANGE RULE BUILDING

To prove our thesis, we conduct three studies on framework API evolution. The results of the first study on the reality of API changes and usages show that missing classes and methods occur more frequently in frameworks and also affect client programs more often. The second study on the usefulness of the API change rules shows that the API change rules do help developers find the replacements more accurately. The more precise the change rules are, the better they help developers. Consequently, we conduct our third study to investigate how we can improve previous approaches to build API change rules. Specifically, we investigate the effectiveness of the features used to build changes rules and multi-objective-optimization-based approaches to outperform previous approaches.

All the previous approaches use the similarity values between some features of t and m, such as call dependency (Dagenais and Robillard, 2011) or signature similarity (Kim et al., 2007). Given a target method t, these approaches use similarity values to sort the methods in the new releases of framework and suggest change rules to developers by recommending those at the top as potential replacements. Previous approaches use multiple features to detect change rules by prioritizing features. They implement prioritization in two ways. One explicitly gives high priority to certain features in their algorithm, *e.g.*, AURA (Wu et al., 2010) gives higher priority to call-dependency over signature similarity. Another assigns weights to features, such as the approach of Kim et al. (2005).

For a given program, if an API method and its replacement are not similar with respect to one feature, approaches using this feature cannot detect a correct change rule. For example, some replacement methods may be appropriate when considering only call dependency but not when considering signature similarity. Also, a specific program may require favouring one feature over the others, *i.e.*, an approach using that feature would detect more correct change rules than those using the others.

To detect more correct change rules, considering more features is promising but not straightforward, because multiple features may give contradictory information, confusing prioritizing approaches. An incorrect change rule suggested by a high-priority (heavy-weight) feature cannot be overridden by lower-priority (light-weight) features. Besides, these approaches are difficult to extend to new features, because their developers must choose the priorities, parameters, and thresholds of the new features with respect to the other features (Kim et al., 2005; Kpodjedo et al., 2013).

Although there are many approaches to build API changes using different features, the effectiveness of individual feature and of different ways to use them have not been fully investigated. Such studies would help researchers devise more accurate and extendible approaches to build API change rules.

We could use Multi-Objective OPtimization (MOOP) techniques to handle possibly contradictory information given by multiple features while building API change rules. MOOP (Sawaragi et al., 1985) is the process of finding solutions to problems with potentially conflicting objectives. No previous work uses MOOP techniques to combine features.

Consequently, we conduct a study to compare approaches using different features in different ways, including MOOP-based and prioritizing techniques, to build change rules. The goal of our study is two-fold: (1) identify features that are really beneficial to detect API change rules and (2) find the combining process that obtains the best accuracy from the features. We want to answer the following research questions:

RQ1: How effective are the features used in the literature to build change rules?
RQ2: Do the approaches using multiple features outperform those using single features?
RQ3: Do MOOP-based approaches outperform the non-MOOP approaches?
RQ4: Do MOOP-based approaches outperform state-of-the-art approaches?

We study previous works (Cossette and Walker, 2012; Dagenais and Robillard, 2011; Godfrey and Zou, 2005; Kim et al., 2005, 2007; Kpodjedo et al., 2013; Meng et al., 2012; Schäfer et al., 2008; Wu et al., 2010; Xing and Stroulia, 2007a) and select six metrics of four features used by them: call dependency similarity, comment similarity, inheritance relation, and signature similarity used by them. We use these features to build six approaches using each feature metric, five approaches prioritizing these features, and five corresponding MOOP-based approaches using the same features.

We compare these approaches on six open-source frameworks in Java. For each target method, we generate a list of candidate replacement methods with a maximum size of six methods using each approach. We chose six to abide by the human capacity to process information (seven plus or minus two) (Miller, 1956). Then, we compare the sets of change rules using two criteria: the number of change rules with correct replacements and the average position of correct replacements. We find that not all features are useful individually. Signature similarity is more effective than the other features. Not all the multi-feature approaches outperform single feature approaches and the result of the MOOP-based approaches is more stable and accurate than that of the corresponding prioritising approaches.

Also, we compared the MOOP-based approach using all the six feature metrics with a state-

of-the-art approach MADMatch (Kpodjedo et al., 2013). The MOOP-based approach build more change rules with correct replacements on two of four programs and the same number of change rules on the other two. The results showed that MOOP is an effective way to combine multiple features to build change rules, especially, when there is no prior knowledge to favour certain features.

5.1 Study Design

In this section, we describe the design and the implementation of our study. First, we present the features selected in our study. Then, we describe the implementations of the individual, non-MOOP, and MOOP-based approaches.

5.1.1 Feature Selection

In our study, we select four features: method signature, call-dependency, inheritance relation, and source code comments. The first three are the top-three features used by previous works, as shown in Section 2.2. Although source code comments are only used by UMLDiff (Xing and Stroulia, 2007a), we still include them in this study because Cossette and Walker (2012) reported that developers often use them to find the replacements of missing methods. We want to investigate the effectiveness of source code comments too.

We use six metrics to measure the features. These metrics represent different characteristics of methods. Method signature has three metrics to reflect different levels of lexical similarities. Five of the six feature metrics are used by previous approaches. LCS of inheritance relation is derived from the approach of Schäfer et al. (2008). We present how to compute these feature metrics below.

— Confidence value for call-dependency relations
— LCS for source code comments
— LCS for inheritance relations
— Method-level distance (MD) for method signatures
— LCS for method signatures
— LD for method signatures

5.1.2 Feature Metric Computation

To compute the LD and the LCS, we tokenise the strings as proposed by Lawrie et al. (2006) by splitting them at upper-case letters and other characters, such underscore, space,

punctuation. Previous works also follow the same process.

Call-dependency Relation

To compute call-dependency relation similarity, we first define an *anchor* a as a pair of methods with the same signature (including return type, declaring module, name, and parameter lists) that exist in both the old and new releases. The set of anchors A is defined as:

$$a = < a_i,\ a_j > \wedge\ j > i \tag{5.1}$$

$$A = \{a \mid (a_i \in R_i \wedge a_j \in R_j \wedge S(a_i) = S(a_j))\} \tag{5.2}$$

where the function $S(a_i)$ returns the signature of method a_i and i and j represent the releases of a program.

There are different implementations of call-dependency analyses. We choose that based on association rule mining (Agrawal et al., 1993) which is used by SemDiff (Dagenais and Robillard, 2011) and the approach of Schäfer et al. (2008). Association rule mining is a machine learning technique to build and verify the links between the items of two data sets. It uses Confidence Value (CV) and Support to measure the quality of the links. CV is a percentage to show how reliable a link. When we measure call dependency relations, the confidence value $CV(t, c)$ for a given target method t and its candidate replacement method c are computed as follows:

$$\mathbf{A}(t) = |\{\ a \mid a \in A\ \wedge a_i \rightarrow t\ \}| \tag{5.3}$$

$$\mathbf{A}(t, c) = |\{\ a \mid a \in A\ \wedge\ a_i \rightarrow t\ \wedge\ a_j \rightarrow c\ \}| \tag{5.4}$$

$$\mathbf{CV}(t, c) = \frac{\mathbf{A}(t, c)}{\mathbf{A}(t)} \tag{5.5}$$

where $a_i \rightarrow t$ represents method a_i calls method t.

Comments

Same as UMLDiff (Xing and Stroulia, 2007a), we use LCS to represent the similarity of the comments of two methods and analyse Javadoc comments in the source code, because they are directly connected with methods by the Java compiler. In the Javadoc comments for methods, we extract the textual parts and exclude the parts under annotations, such as the parts after `@param` and `@return`. Usually, the text part is the description of the functions of methods and should be relatively stable even their signatures are changed. Parameters

and return values of methods are subject to change when the signature changes, so the annotations related to them are likely to be different also, even if the functions of methods are equivalent.

Inheritance Relation

To compute the similarity of the inheritance trees between two methods, we extended that used in the approach of Schäfer et al. (2008). Schäfer *et al.* extract extension and implementation as two different inheritance relations, we combine them into one metric. First, we convert the whole inheritance tree of the class of each method into a string by traversing both inheritance trees and implementation trees in lexicographic order. We sort the interfaces in the implementation trees by their qualified names to avoid the influence of interface order.

Using a Java example in Figure 5.1, if a method m belongs to a class C, the parent class tree of C is *java.lang.Object*, $p1.P1$, and $p2.P2$. C also implements two interfaces $p1.I1$ and $p2.I2$ whose parent interfaces are $p1.PI1$ and $p2.PI2$ respectively. Then the string representing the inheritance tree of C is $p1.P1.p2.P2.p1.PI1.p1.I1.p2.PI2.p2.I2$ [1].

Then, we use LCS between the two strings representing the inheritance trees of the classes in which the two methods are defined to measure the similarity of their inheritance relations.

Method Signature

Previous works use three metrics to compare method signature similarity. Method-level Distance (MD) (Wu et al., 2010) reflects coarse-grain similarity between two methods' signatures. Levenshtein Distance (LD) (Schäfer et al., 2008; Wu et al., 2010) and Longest Common Subsequence (LCS) (Kim et al., 2007; Xing and Stroulia, 2007a) represent finer-grain methods signature similarities.

To compute Method-level Distance $MD(c,t)$, we define functions $R(c), D(c), N(c), P(c)$ to get the return type, declaring class, name, and formal parameters of method c, respectively. MD measures the similarity of method signatures in a coarse way and can prevent that a dramatic change in one from $R(c), D(c), N(c), P(c)$ misleads change rule building. The

1. In this example, we do not take *java.lang.Object* into account, because it is the root of all Java classes and adding it has no influence on the result

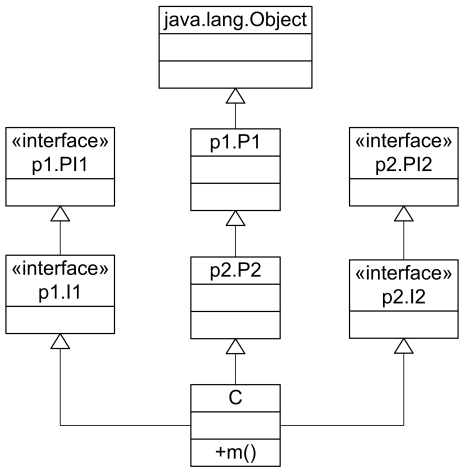


Figure 5.1 Inheritance Tree Example

method level distance between c and a target method t is:

$$E_c = \{R(c), D(c), N(c), P(c)\} \tag{5.6}$$

$$d_i(c,t) = \begin{cases} 0 \; if \; E_c(i) = E_t(i) \\ 1 \; if \; E_c(i) \neq E_t(i) \end{cases} \tag{5.7}$$

$$MD(c,t) = \sum_{i=1}^{4} d_i(c,t) \tag{5.8}$$

where $E_c(i)$ returns $R(c), D(c), N(c), P(c)$ respectively while $i = 1, ..., 4$. $MD(c,t)$ returns how many of the four elements (return types, declaring classes, names, and parameters) of the signature of c are different to those of t.

The LD and LCS of the signatures of two methods are denoted as $LD(S(c), S(t))$ and $LCS(S(c), S(t))$ respectively. A candidate method with the smallest LD does not always have the largest LCS. For example, let us assume that we want to identify the method most similar to `ab` between `a`, `abc`, and `abcd`. Both `a` and `abc` have the same LD and both `abc` and `abcd` have the same LCS. So, if we only use one of these measures, we obtain two methods with the same similarity value. Thus, first using LD, the candidate set can be narrowed to `a` and `abc`, then comparing them with LCS, `abc` is identified as the most similar to `ab`.

5.1.3 Approach Implementations

In our study, we first build the experiment approaches using the feature metrics to investigate the effectiveness of single features and of prioritising and MOOP ways to combine these features. We try to avoid influences of other factors, *e.g.*, input data granularity. As discussed

above, some approaches (Dagenais and Robillard, 2011; Kim et al., 2005; Meng et al., 2012) take software repository commits as input. Software repository commits can be seen as small releases. The fine-grained change information in the commits may help improve the precision and recall of the detected rules. Because the goal of the study is not to investigate the contributions of the fine-grained information to the improvement of precision and recall, we exclude the influence of different inputs.

Some approaches use complex processes, such as the multi-iteration algorithms in AURA (Wu et al., 2010) and UMLDiff (Xing and Stroulia, 2007a), to make the best of one feature. For example, the multi-iteration algorithms used by AURA and UMLDiff are to improve the accuracy of call-dependency and signature similarity analyses, respectively. Such complex processes can tune the usage of one feature which affects prioritized and MOOP-based approaches equally. Therefore, we exclude them from the implementations of the approaches in our study. Furthermore, many approaches use thresholds and parameters and their values are difficult to choose. We want to avoid their influences too. To demonstrate the effectiveness of MOOP techniques, we directly compare with MADMatch (Kpodjedo et al., 2013), one of the state-of-the-art approaches from previous work. The approaches that we implemented for this study are described in details below.

Individual and Prioritised Approaches

The approaches used in our study are shown in Table 5.1. Individual approaches $I1$ to $I6$ use a single feature metric to build change rules for target methods. Based on each of the six metrics described in Section 5.1.2, they compute the similarity of the potential replacement methods to a target method in subject programs, sort them and keep the six most similar candidates as potential change rules.

With six individual feature metrics, building the prioritised approaches with all possible permutations is impractical. We select five combinations shown in Table 5.1 in our experiment. AURA (Wu et al., 2010), a derived approach based on $P1$, showed that combining call-dependency similarity and method signature LCS and LD is effective. We want to verify if combining other features with method signature similarity has comparable performances.

In prioritised approaches, candidates are compared with the metrics from high to low priorities as previous approaches. For example, similar to AURA (Wu et al., 2010), with $P1$, if two candidate methods have different call-dependency similarities, the approach does not compare them by their signature LCS. If they are the same, then their similarities are decided according to the signature LCS and so on.

Using call-dependency, comment, and inheritance similarities at high priority, approaches can discover the replacements whose signatures are not alike to the target methods (Dagenais and Robillard, 2011; Schäfer et al., 2008; Wu et al., 2010). In $P5$, we combine all the four metrics used in $P1 - P4$. With the priorities from high to low, the feature metrics used are call-dependency, comment, inheritance and method-level distance with method signature LCS and LD.

MOOP Formulation

We formalise framework API evolution problem as a MOOP problem as follows. Let us assume that there is a framework with two releases R_i and R_j with $i < j$. F is the set of features that we consider to build change rules between R_i and R_j:

$$\mathbf{F} \quad = \quad \{f_1, ..., f_n\} \tag{5.9}$$

We want to identify replacement methods for the target methods in R_i but no longer in R_j, in terms of their signatures including return type, declaring class, method name, and formal parameters. Then, the target methods set T is defined as:

$$T \quad = \quad \{t \mid t \ \in \ R_i \ \wedge \ \nexists \, m \ \in R_j \ : \ S(t) = S(m)\} \tag{5.10}$$

For each target method t in T, every method in R_j is a potential replacement method, *i.e.*, candidate method. We define the candidate method set C, where x is the size of C and c_j is a method in R_j, as:

$$C \quad = \quad \{c_1, ..., c_x\} \tag{5.11}$$

Table 5.1 Individual, Prioritised and corresponding MOOP approaches

Approaches	Features		
I1	Call-dependency		
I2	Comment		
I3	Inheritance		
I4	MD (Method-level Distance)		
I5	MLCS (Method Signature LCS)		
I6	MLD (Method Signature LD)		
P1, M1	Call-dependency	MLCS	MLD
P2, M2	Comment	MLCS	MLD
P3, M3	Inheritance	MLCS	MLD
P4, M4	MD	MLCS	MLD
P5, M5	All Four in P1-P4	MLCS	MLD

For each feature f_i, we abstract the feature metrics as $sim_i(t, c_k)$ between t and c_k as the measurements of the *objectives* in the MOOP formulation. The computations of $sim_i(t, c_k)$ are implemented according the *fitness function*s described in Section 5.1.2. The set of similarities is $SIM(t, c_k)$:

$$SIM(t, c_k) \quad = \quad \{sim_1(t, c_k), ..., sim_n(t, c_k)\} \tag{5.12}$$

For two candidate methods c_p and c_q in R_j, we define $c_p \blacktriangleright c_q$ to represent that c_p dominates c_q, which means that, for each feature f_i, $sim_i(t, c_p)$ is better than or equal to $sim_i(t, c_q)$ and there is at least a feature f_z for which $sim_z(t, c_p)$ is strictly better than $sim_z(t, c_q)$:

$$c_p \blacktriangleright c_q \tag{5.13}$$

$$iff \quad sim_i(t, c_p) \geq sim_i(t, c_q) \; \forall \; i \in \{1, ..., n\} \tag{5.14}$$

$$\wedge \quad \exists \; i \in \{1, ..., n\} \mid sim_i(t, c_p) > sim_i(t, c_q) \tag{5.15}$$

We thus build the Pareto front (Zitzler and Thiele, 1999) P_t for each target method t. P_t is a subset of C that each element p_t is not dominated by the other methods in C.

$$P_t \quad = \quad \{p_t \mid \forall \; c \; \in \; C \; \wedge \; c \; \notin \; P_t \; : \; p_t \blacktriangleright c\} \tag{5.16}$$

We expect that the correct replacement methods for a target method t in T belong to its Pareto front. The Pareto fronts are the recommendation sets detected by MOOP approaches and we use recommendation sets and Pareto fronts interchangeably.

MOOP Implementation

For each prioritised approaches described in Section 5.1.3, we implement a MOOP-based approach that uses the same feature metrics and name them $M1, ..., M5$. The implementation of our MOOP-based approaches is based on jMetal (Durillo and Nebro, 2011), a package for solving MOOP problems in Java. As discussed in Section 5.1.3, the number of potential replacement methods is not large and it is feasible to traverse all of them to find the non-dominated methods. Therefore, we replace the meta-heuristic search in jMetal with a simple traversal of all solutions in our implementation.

A typical MOOP algorithm is multi-iterative. At each iteration, the first step is *population generation* to generate a pre-defined number of solutions. The second step is *population selection* that selects the non-dominated solutions and adds them to the Pareto front. When all the solutions are covered or a predefined maximum number of iteration is reached, the

algorithm reports the Pareto front.

In the implementation of our MOOP-based approaches, the solutions are the methods in the potential replacement methods. We map the feature metrics for a target method t and a potential replacement method c to the objectives and implemented the six fitness functions defined in Section 5.1.2 to compute their values.

We do not give any constraints to the objectives in our study because we want to compare the approaches as generally as possible. Many previous approaches (Dagenais and Robillard, 2011; Kim et al., 2007; Schäfer et al., 2008) use thresholds to increase the precision of detected change rules. With thresholds, approaches may not report detected change rules because the similarities of a target method and its potential replacements are not high enough.

The change operator of our MOOP implementation is traversing the candidate method list with a predefined step size s, *i.e.*, each population has s candidate methods. Because we traverse all the methods defined in the new releases of frameworks, the value of s does not have any influence on the generated Pareto front. In our implementation, we set s to six. We use the binary tournament operator to select the solutions.

Contrary to the individual and prioritised approaches, the methods in the original recommendation list of MOOP-based approaches are not sorted, because they are non-dominating each other when considering all the feature metrics. However, one can be better or worse than the others in a specific objective. In our MOOP implementation, we sort the recommendation list according to the Best Objective Number (BON), *i.e.*, the number of objectives of a method which are better than those of the others in the list. If all the recommendations are the same for a feature metric, none of them are better in this metric.

For example, if method a is the best in signature LCS and inheritance tree LCS, method b has the largest comment LCS, method c has the same CV as a and b, then the sorted recommendations are a, b, and c with BON of 2, 1, 0 respectively.

5.2 Study Execution

We now present our experiment. We first describe the subject programs and how to evaluate the results, then present the results.

5.2.1 Subject Programs

We select two releases of six open-source Java programs: Android SDK, jEdit, jFreeChart, jHotDraw, Log4j, and Struts. Table 5.2 shows their statistic data.

Table 5.2 Subject Programs

Subject Programs	Releases	# Methods	# Target Methods
Android SDK	2.1_r2.1p2	20,516	106
	2.2.3_r2	21,214	
jEdit	4.1	2,773	87
	4.2	3,547	
jFreeChart	0.9.11	4,751	30
	0.9.12	5,197	
jHotDraw	5.2	1,486	43
	5.3	2,265	
Log4j	1.0.4	906	15
	1.1.3	1,110	
Struts	1.1	5,973	91
	1.2.4	6,111	

These programs are medium size systems and have different characteristics regarding API evolutions. Android (Google, 2007) is a mobile operating system developed by Google. Android is written in several programming languages but its SDK is in Java. We analysed Android SDK releases 2.1_r2.1p2 and 2.2.3_r2 in our study. They are the latest versions of two releases of Android: Eclair (API Level 7) and Froyo (API Level 8). No previous work has evaluated the API changes of Android SDK. JEdit (jEdit, 2014) is a text editor. Its API and implementation between v4.1 and 4.2 changed dramatically. JFreeChart (jFreechart, 2014) is a chart library. The APIs between its v0.9.11 and v0.9.12 also changed a lot and there are many APIs in v0.9.12 with very similar names. The dramatic changes in the two programs are challenging for both developers and the approaches to build API changes. JHotDraw (jHotDraw, 2014) is a GUI framework developed by Gamma *et al.* to demonstrate the application of design patterns (Gamma et al., 1995). Yet, between v5.2 and v5.3, there are several API changes. Log4j (Apache, 2014a) is a logging framework and Struts (Apache, 2014b) is a Java web application framework. Both are developed by Apache. The release pairs that we choose are in their early stages and with API changes.

5.2.2 Target Methods

In our experiment, we detect and validate the change rules for public and protected target methods with changed class or method names in the new releases, because developers do need change rules to find their replacements. Most previous works (Godfrey and Zou, 2005; Kim et al., 2007, 2005; Kpodjedo et al., 2013; Meng et al., 2012; Schäfer et al., 2008; Wu et al., 2010; Xing and Stroulia, 2007a) evaluated the change rules for all the target methods. However, Des Rivières classified API changes according to program elements (Rivières, 2008), such as class or method name changes, method parameter changes, modifier changes, etc. From this classification, we can find out that only class or method name changes (including moving,

renaming, and removing) require change rules to help developers find the replacements, because developers do not have a reliable way to locate their replacements. For other type of changes, *e.g.*, only method parameter changes, developers can easily find the replacements in the new releases of frameworks based on the unchanged class names and method names.

Therefore, the evaluation on the change rules of public and protected target methods with class name and method name changes has two advantages: (1) the results reflect the effectiveness of approaches more accurately than evaluations on the change rules of all the target methods and (2) the amount of manual validation work is greatly reduced. The numbers of target methods of the six subject programs in our experiment are shown in Table 5.2.

5.2.3 Evaluation

To evaluate the change rules of the approaches in our study, we read the source code of the subject programs carefully to verify if the change rules have correct replacements. The replacement candidate method set contains the methods defined in the new releases of the subject programs and the methods called in the subject programs, but defined in other frameworks developed by the same providers. We identify the same providers using the package names. For example, if both the package names of two methods start with “org.apache", we consider them as developed by the same provider. Other methods are excluded from the candidate set. At least two of the authors agreed on the results. Then, we count the numbers of the change rules with correct replacements and the positions of the correct replacements in the recommendation lists as the metrics to compare the approaches. The approaches with larger number of the change rules with correct replacements and smaller position of correct replacement are better.

5.3 Study Results

In this section, we present the results of individual, prioritising, and MOOP-based approaches on the subject programs.

Individual Feature Metrics

To investigate more efficient ways to combine multiple features, we first observe the effectiveness of these features to build change rules individually. Table 5.3 shows the numbers of change rules with correct replacements built by approaches using individual feature metrics. Among these approaches, only $I5$ and $I6$ that use method signature LCS and method signature LD, respectively, provide stable good results. The results of others fluctuate between

programs and are worse than those of $I5$ and $I6$ except for the result of $I4$ on jHotDraw. The results of the overlaps between five [2] individual feature metrics are shown in Figure 5.2.

In Figure 5.2, each large oval represents a feature metric. The sum of the numbers in an oval is the number of the correct change rules that can be detected by that feature metric. The numbers in the small overlapped areas of the ovals represent the numbers of correct change rules that can be detected by the feature metrics represented by the overlapped ovals.

In Figure 5.2, we observe that the signature LCS detects the most correct change rules. Other feature metrics help to detect more change rules, but their numbers vary between programs. Call-dependency similarity helps detect more correct change rules for four programs while comment similarity helps for two programs.

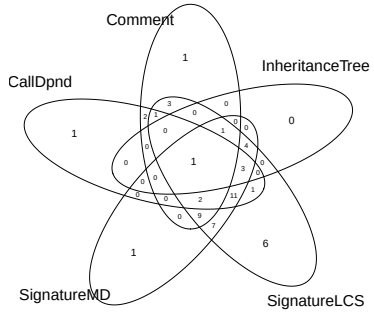
If a feature metric does not bring any new correct replacement alone, it does not mean that the feature metric cannot help detect more correct change rules when combined with other feature metrics. For example, if the correct replacement r and other methods $M = \{m_1, m_2, ..., m_n\}$ have the same comment LCS to a target method t, the approach using comment similarity only may not report r in the change rules. If r is more similar to t in signature LCS than the methods in M, combining comment LCS and signature LCS can promote r to the top and r is not necessary to be the most similar one to t in signature similarity while considering all the potential replacement methods.

The approaches using call-dependency relations or source code comments alone can only build change rules for a part of the target methods, because not all the target methods are called by anchors inside frameworks or have comments. The other approaches can detect change rules for all the target methods. Figure 5.3 shows that the percentages of target methods called by anchors and with comments vary between programs. Call-dependency similarity applies to only less than 50% of target methods and less than 5% between Struts 1.0.4 to

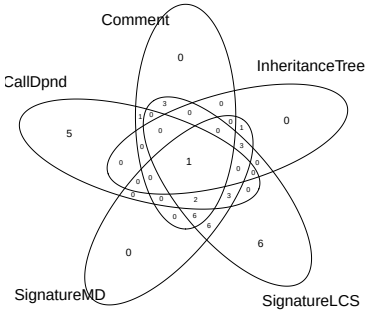
2. We do not include method signature LD, because the libraries available to draw Venn diagrams only support maximum five dimensions and the results of signature LD are similar to those of signature LCS, as we presented in Table 5.3.

Table 5.3 Numbers of change rules with correct replacements - individual feature metrics

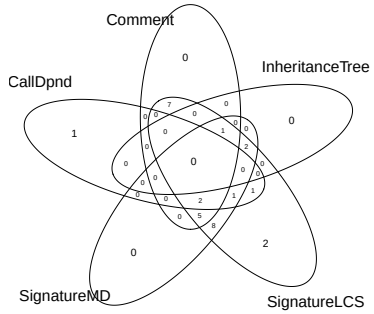
	I1:CallDependency	I2:Comment	I3:Inheritance	I4:SignatureMD	I5:SignatureLCS	I6:SignatureLD
Android	22	20	9	39	50	51
jEdit	12	13	5	22	30	27
jFreeChart	5	15	3	19	29	26
jHotDraw	14	20	14	39	36	38
Log4j	6	8	3	10	12	10
Struts	2	8	17	18	19	18



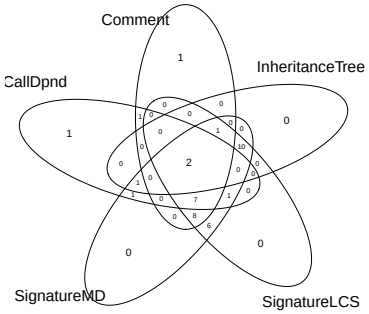
(a) Android SDK



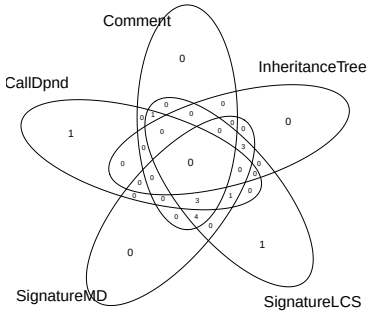
(b) jEdit



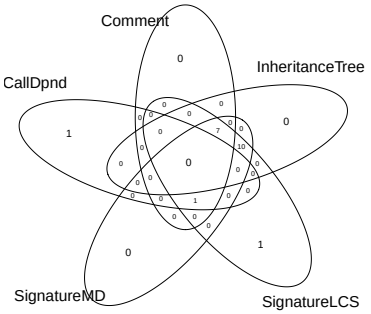
(c) jFreeChart



(d) jHotDraw



(e) Log4j



(f) Struts

Figure 5.2 Effectiveness of individual features

1.1.3 as worst case. More than 50% of target methods have comments, *i.e.*, the replacement methods of these target methods could be detected by comments similarity. On average, call-dependency similarity and comment similarity apply to 31% and to 75%, respectively, of all the target methods in our study.

Number of Change Rules with Correct Replacements

Figure 5.4 shows the numbers of change rules with correct replacements built by the approaches in this study. We observe that, MOOP-based approaches give results better than or close to those of the non-MOOP approaches using the same feature metrics, except between $P3$ and $M3$ on jHotDraw (this case will be discussed in Section 5.4.3). More important, $P5$ which prioritises all the feature metrics does not always give better results than other prioritised approaches using less features ($P1$ to $P4$). There are even large decreases in three programs (jEdit, jFreeChart, and Struts). In contrary, $M5$, the MOOP-based approach using the same feature metrics as $P5$, generally outperforms all the other approaches, only build one and two less correct change rules than $P1$ on Android SDK and jFreeChart, respectively.

In Section 5.3, we found that the results of approaches only using lexical similarities are stable, because most of the replacements have similar method signatures. Here, we observe that combining lexical similarities with other features, such as call-dependency, comment and inheritance relations, is not always helpful to detect more correct change rules. Only $P1$, which combines call-dependency similarity with signature similarity, is consistently better than the approaches using only lexical similarity. $P5$, the approach combining all the feature metrics in a prioritised way, is not consistently better than the two single feature approaches either. This result confirms that integrating multi-feature is not straightforward.

On the other hand, MOOP-based approaches provide more stable and generally good results. Combining multiple features in a MOOP way can mitigate the confusion caused by possibly contradictory information of multi-features.

Besides the number of change rules with correct replacements, the positions of the correct replacements are also important for developers to use the change rules. The correct replacements with higher ranks are easier to be picked up by developers.

Positions of Correct Replacements

The approaches in our study output maximum six recommendations. Even with the same numbers of target methods with correct recommendations, the approaches with smaller correct recommendation positions are preferable to developers. Figure 5.5 shows the average

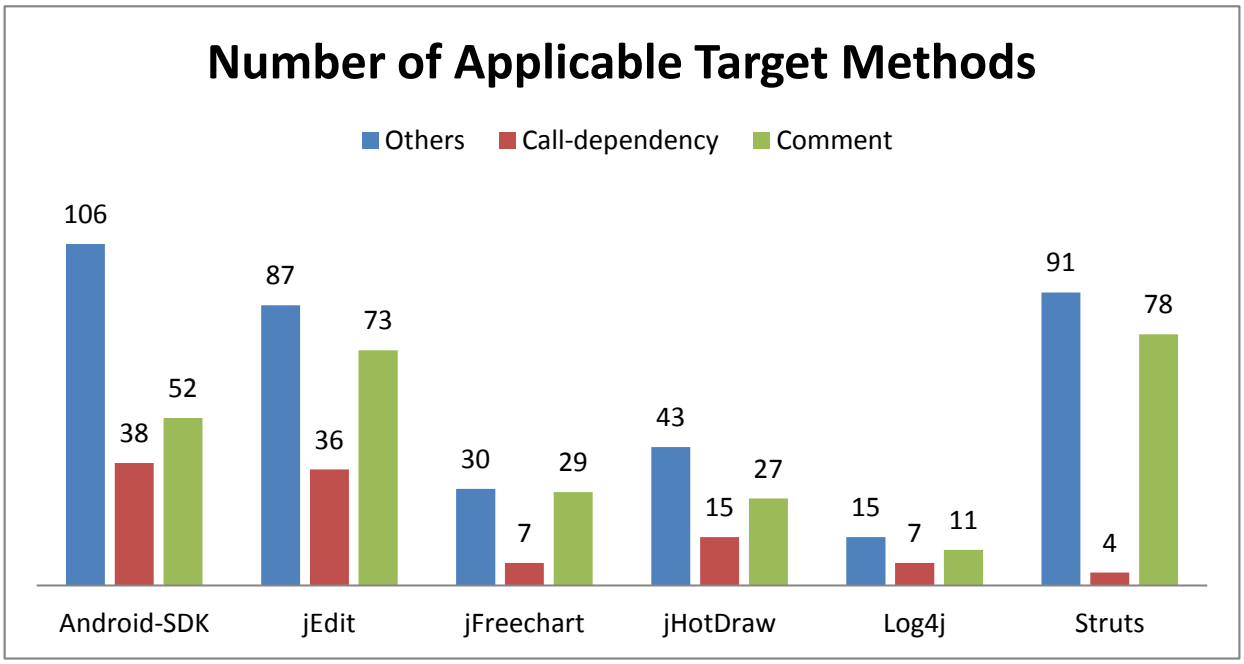


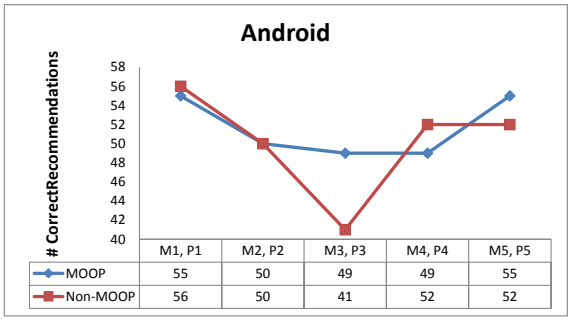
Figure 5.3 Number of change rules

correct replacement positions of the approaches in our study. Generally, the correct replacement positions of MOOP-based approaches are higher than those of the corresponding non-MOOP approaches. There are three cases where MOOP-based approaches with lower average correct replacement positions, $M2$ on jFreeChart and $M3$ on jEdit and Log4j. However, the MOOP-based approaches build 36%, 79%, and 100% more correct change rules than corresponding non-MOOP approaches, respectively in the three cases.

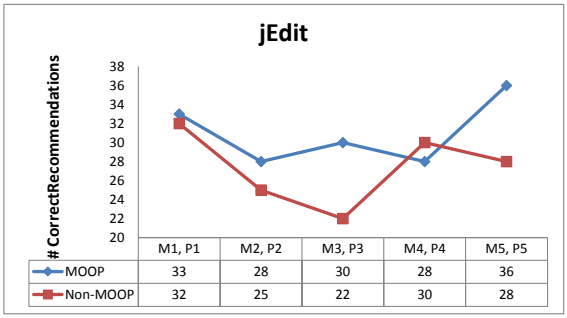
Comparison with MADMatch

MADMatch (Kpodjedo et al., 2013) is one of the state-of-the-art approaches matching program elements. It combines four features (call-dependency, inheritance, code-structure, and signature) to improve the precision and recall. The authors analysed four of the six subject programs used in this study and shared the results with us. We compared $M5$ with MADMatch on the numbers of change rules with correct replacements and the results are in Table 5.4. MADMatch and $M5$ have the same results on jFreeChart and Struts and $M5$ detects more change rules with correct replacements on jEdit and jHotDraw.

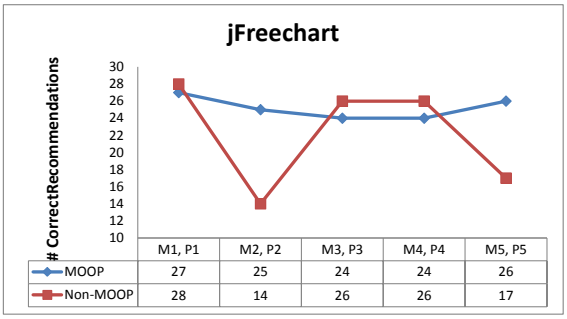
Two reasons cause that the change rules are missed by MADMatch: (1) MADMatch was mislead by the conflicting features. For example, `DisplayTokenHandler.getChunks()` is replaced by `DisplayTokenHandler.getChunkList()` between jEdit 4.1 and 4.2. This change



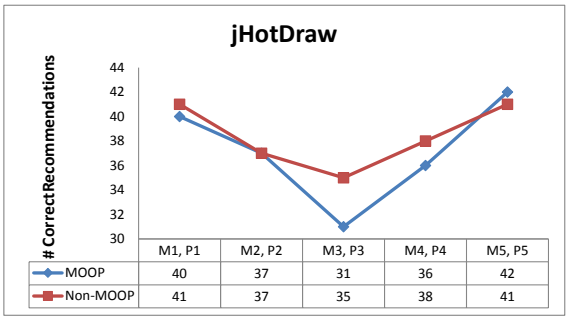
(a) Android SDK



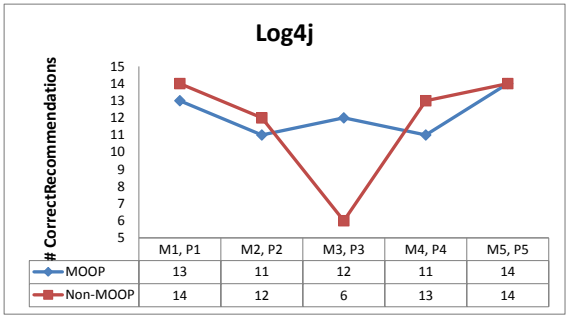
(b) jEdit



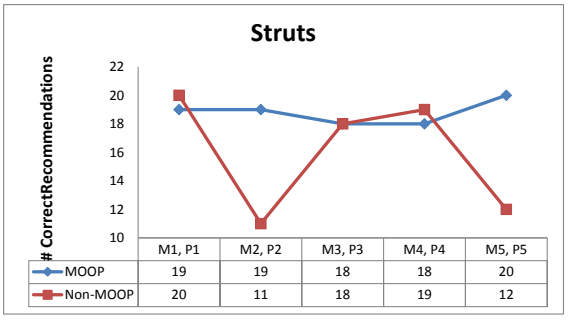
(c) jFreeChart



(d) jHotDraw

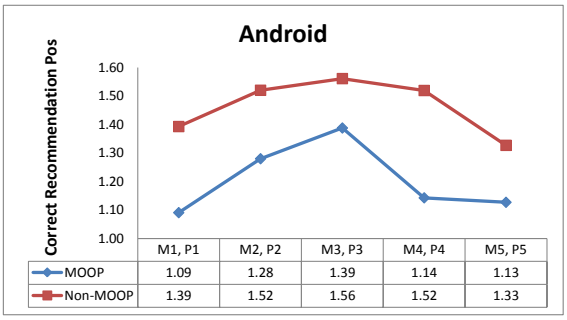


(e) Log4j



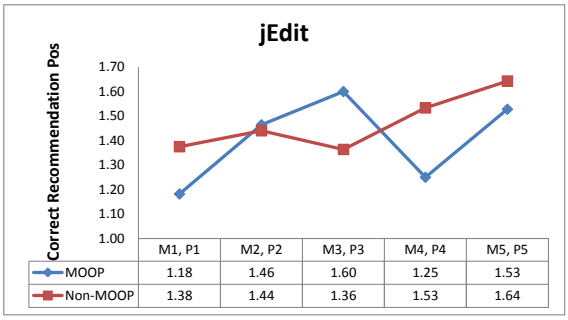
(f) Struts

Figure 5.4 Numbers of change rules with correct replacements - non-MOOP vs MOOP



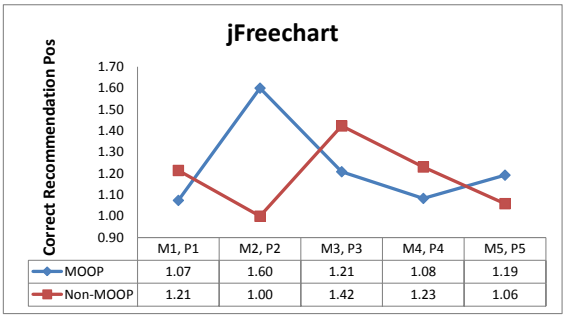
	M1, P1	M2, P2	M3, P3	M4, P4	M5, P5
MOOP	1.09	1.28	1.39	1.14	1.13
Non-MOOP	1.39	1.52	1.56	1.52	1.33

(a) Android SDK



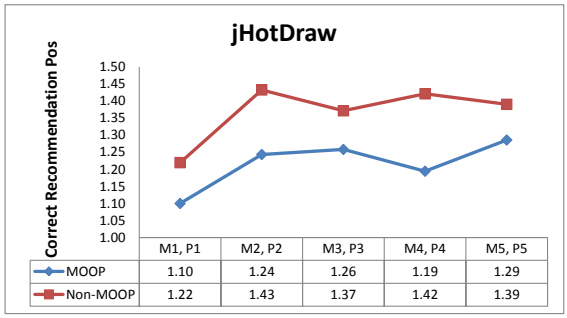
	M1, P1	M2, P2	M3, P3	M4, P4	M5, P5
MOOP	1.18	1.46	1.60	1.25	1.53
Non-MOOP	1.38	1.44	1.36	1.53	1.64

(b) jEdit



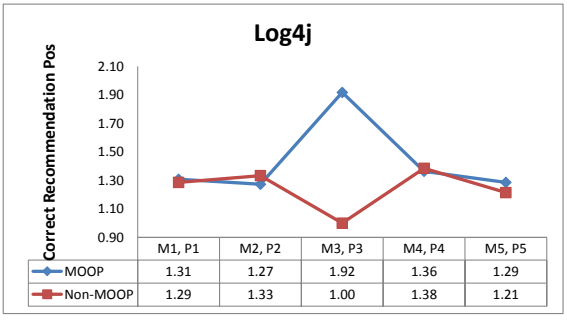
	M1, P1	M2, P2	M3, P3	M4, P4	M5, P5
MOOP	1.07	1.60	1.21	1.08	1.19
Non-MOOP	1.21	1.00	1.42	1.23	1.06

(c) jFreeChart



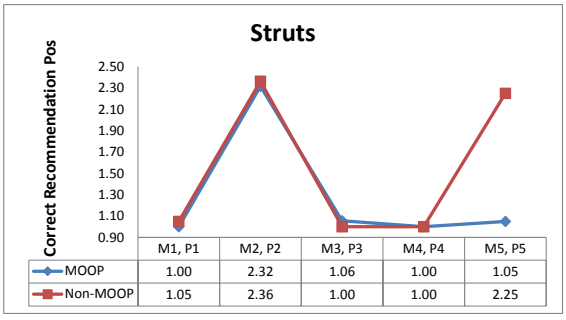
	M1, P1	M2, P2	M3, P3	M4, P4	M5, P5
MOOP	1.10	1.24	1.26	1.19	1.29
Non-MOOP	1.22	1.43	1.37	1.42	1.39

(d) jHotDraw



	M1, P1	M2, P2	M3, P3	M4, P4	M5, P5
MOOP	1.31	1.27	1.92	1.36	1.29
Non-MOOP	1.29	1.33	1.00	1.38	1.21

(e) Log4j



	M1, P1	M2, P2	M3, P3	M4, P4	M5, P5
MOOP	1.00	2.32	1.06	1.00	1.05
Non-MOOP	1.05	2.36	1.00	1.00	2.25

(f) Struts

Figure 5.5 Average correct replacement positions - non-MOOP vs MOOP

rule can be built by method signature text similarity. However, MADMatch reported `Display TokenHandler.canMerge(...)`, which has much lower lexical similarity. (2) The thresholds used prevents MADMatch building some change rules, because the results under thresholds are discarded. This is a typical limitation of using thresholds.

Because $M5$ give the recommendation list with maximum six elements and MADMatch only gives one recommendation, we cannot say that $M5$ outperforms MADMatch for all the cases. Interested readers can refer to (Kpodjedo et al., 2013) for details of the implementation of MADMatch. Yet, the MOOP-based approach can provide comparable or better results with a more straightforward way to combine features than non-MOOP approaches. More discussions about recommendation list size are in Section 5.4.1.

5.3.1 Summary

We can answer the research questions based on the results of the experiment as follows:

RQ1: How effective are the features used in the literature to build change rules?

A: Among the individual feature metrics, only method signature LCS and method signature LD can build change rules with reliable quality. These two metrics perform slightly different between programs.

RQ2: Do the approaches using multiple features outperform those using single features?

A: No, combining more features does not necessarily help build more correct change rules. It depends on the features and how to combine them. The approach combining call-dependency and signature similarities produces stable good results, while the results of other combinations very between programs.

RQ3: Do MOOP-based approaches outperform the non-MOOP approaches?

A: Yes, MOOP-based approaches generally detect more correct change rules with higher correct replacement positions, compared to the prioritised approaches using the same features. On average, MOOP-based approaches detect 13% more correct change rules with 3% smaller average correct replacement position than the corresponding prioritising approaches.

Table 5.4 Numbers of change rules with correct replacements - $M5$ vs MADMatch

	jEdit	jFreeChart	jHotDraw	Struts
M5	36	26	42	20
MADMatch	26	26	31	20

RQ4: Do MOOP-based approaches outperform state-of-the-art approaches?

A: Yes, the MOOP-based approach $M5$ detects more change rules than MADMatch on two of four subject programs and the same number of change rules on the other two.

Furthermore, the prioritised approach considering all the feature metrics, $P5$, perform differently between subject programs. Its results are less accurate than individual approaches sometimes, because of the contradictory information between the feature metrics. Therefore, considering more features is a source of decrease in accuracy. The stable results given by the MOOP-based approaches show that MOOP is an effective way to handle the potential conflicts between feature metrics. Also, MOOP-based approaches do not use context-sensitive thresholds or parameters and easy to be extended to new features.

5.4 Discussion

We now discuss issues on framework API evolution identification and approaches comparison.

5.4.1 Recommendation List Size

We argue that approaches on framework API evolution should give recommendation list, as Google search results, instead of one recommendation, because of the asymmetric costs between searching for correct replacements and identifying wrong change rules. Developers are better at understanding semantic differences between two methods than any tools, but not at searching the whole source code of the new releases of a framework to find the most similar methods to a method with respect to some features. We prefer recommendation lists with three conditions:

First, the recommendation lists should be suggested by different features. Cossette and Walker (2012) manually analysed three Java programs and discovered that only a part of the target methods can be found by one feature. Therefore, a larger recommendation set based on single feature does not give more correct change rules efficiently, because the positions of the correct replacements may be low. Developers must spend more effort to evaluate the recommendations above.

In our study, the features used represent different characteristics of API methods, including call-dependency, comment, inheritance and different level of method signature similarities. The results fluctuate when we only combine two of them. Combining them in MOOP-based approach makes the best of the diversified features and produces stable good results.

Second, the size of recommendation set must be small enough. Large data set will overwhelm

developers. Psychological research shows that human capacity to process information is around a magical number seven (plus or minus two) (Miller, 1956). If the size of a data set is much larger than seven, it would much more difficult for developer to process directly.

In our study, prioritising approaches have fixed recommendation list size of six and those of MOOP-approaches are changed between one and six depending on programs and features. Table 5.5 lists the average recommendation list sizes of the five MOOP-based approaches in our study. All of them are less than six and those of $M5$ are larger than the others because $M5$ considered more features.

Third, the positions of the correct replacements are better when closer to the top. Joachims et al. (2005) found in an eye-tracking experiment that subjects scan answer lists from top to bottom before deciding which answers to explore and about 50% of the subjects only scan the top three answers in the list. So if the ranks of correct recommendations are too low, users might not check them at all. Like how people use Google, it is fine if the relevant links are not the first one, but usually we do not check the links on the second page.

As shown in Section 5.3, the average positions of correct replacements are less than 1.6 for $M5$ even the average recommendation sizes are larger than 2.7.

5.4.2 Change Rules without Correct Replacements

Change rules without correct replacements for target methods have two categories: the approach is not able to detect the correct replacements or the target methods are simply deleted without replacement.

An approach cannot detect exiting replacements for two reasons. First, these replacements are not the most similar to the target methods according to the used features. Second, the replacements are not in the new releases of the frameworks, *e.g.*, from third party frameworks or deleted because of framework behaviour changes.

In our study, we found that most of the target methods, for which all approaches in our study cannot find the correct replacements, are simply deleted, because of behaviour changes

Table 5.5 Average recommendation list sizes

	Android	jEdit	jFreeChart	jHotDraw	Log4j	Struts
M1	1.75	2.36	1.67	1.79	1.93	1.88
M2	1.99	2.66	2.43	1.95	1.87	2.64
M3	1.93	2.39	1.57	1.81	2.47	2.23
M4	1.63	2.18	1.60	1.95	1.80	1.95
M5	2.91	3.87	2.93	2.70	4.07	3.67

between the framework releases. We proposed a heuristic treating target methods with the replacements existing also in the old releases of framework as simply-deleted in AURA (Wu et al., 2010). However, this heuristic is not 100% accurate and there is no reliable way to detect simply-deleted change rules yet.

For the target methods whose correct replacements cannot be detected by the approaches, the sizes of recommendation lists are important for developers because they must potentially traverse all of them. If two approaches cannot detect the correct change rule for a target method, the approach with smaller recommendation list size performs better than the other.

In our study, the individual (except that using call-dependency similarity) and prioritised approaches always give six recommendations because we do not use threshold to filter out the recommendations. Figure 5.6 shows the results between $M5$, the individual approach using call-dependency similarity, and the others. The recommendation list size without correct replacements of the individual approach using call-dependency similarity is the smallest, but it can only build change rules for the target methods called by anchors.

On average, the recommendation list sizes without correct replacements of $M5$ is 4.63 comparing to six of the others. For jHotDraw and Log4j, $M5$ only has one target method for each without correct replacements. These two target methods are simply-deleted in the new releases as confirmed by manual validation.

Simply-deleted methods are more difficult to adapt. Developers must validate all the recommendations of the target methods, but still are sure if the target methods are really removed or just missed by the tools. Without a reliable way to detect simply-deleted rules, approaches with high recall and smaller recommendation sizes, such as $M5$, are more desirable for developers to save effort on searching the non-existing replacements.

5.4.3 Limitation

The main limitation of MOOP-based approaches is that the replacement methods must be non-dominated by the other methods according to the features used. This limitation caused $M3$ to detect less change rules with correct replacements than the corresponding prioritised approach $P3$ on jHotDraw. Both $M3$ and $P3$ use inheritance tree LCS and method signature LCS and LD. Between jHotDraw 5.2 and 5.3, there are many target methods replaced by other methods with less signature similarity in the same classes. These replacement methods are dominated by the methods in the same declaring classes and more similar to the target methods in method signatures. Therefore, $M3$ cannot report them.

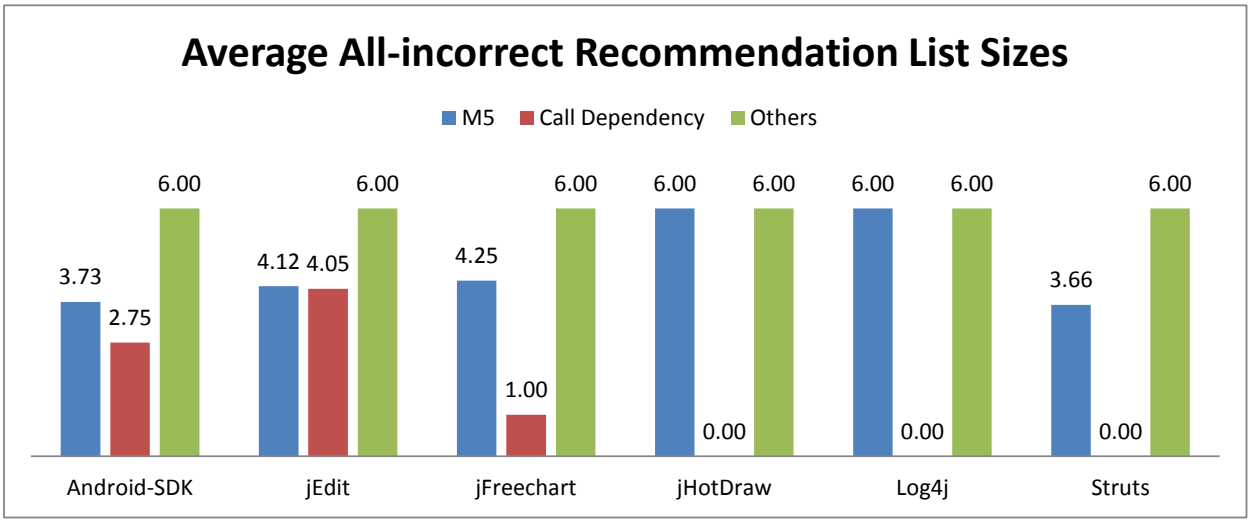


Figure 5.6 Average recommendation list sizes without correct replacements

For example, `ConnectionFigure.start()` in jHotDraw 5.2 is replaced by `ConnectionFigure.getStartConnector()` in v5.3. In class `ConnectionFigure` v5.3, another method `startFigure()` is more similar to `start()` in method signature. Because `startFigure()` and `getStartConnector()` have the same inheritance tree LCS, the former dominates the latter and $M3$ recommended `startFigure()` as the replacement. $P3$ also recommended `startFigure()` first. $P3$ has fixed recommendation list size of six and it recommends `getStartConnector()` at third position.

The results of $M5$ show that this limitation can be ameliorated by including more features. The easiness to extend to new features is an advantage of MOOP-based approaches.

Another limitation of MOOP-based approaches is that the sizes of recommendation list may increase with the numbers of features considered. However, in Figure 5.5, we can see that the average correct replacement positions do not have the same trend and can be lower than the approaches using less features.

5.4.4 Threats to Validity

Construct validity Threats to construct validity concern the relation between theory and observation. In our context, we want to see if MOOP approaches generate more correct change rules with higher positions in the recommendation list and smaller all-incorrect recommendation list sizes than individual and prioritised approaches. We use different approaches as treatments and we observe the number of correct change rules of their outputs, the average correct recommendation positions and the average recommendation list sizes. Thus, we believe that there is no threat compromise the construct validity.

Internal validity Internal validity verifies if the outcome is really caused by the treatment. We identify two threats to internal validity. First, we inspected the change rules generated by the approaches manually. We cannot rule out human error in validating results. To mitigate this threat, we first reduce the possibility of human errors by reducing the number of change rules to validate without compromising the generalizability. As we explained in Section 5.2.1, not all the API changes need to be detected by tools, but those with different class, interface, or method names. Only analysing the change rules of the latter is reliable enough to evaluate approaches and reduces the chance of making errors caused by fatigue. Also, we read the source code of the subject programs carefully and at least two of the authors evaluated and agreed on the results of each program we believe that this threat does not compromise the internal validity. Second, there may be errors in our implementations of the approaches. We carefully implemented and tested these approaches and did not observe errors in the output. Third, we only consider the methods from the new releases of the subject programs and of other programs from the same providers. During the evaluation, we found less than 1% of the replacement methods belonged to third-party frameworks. We believe that it does not harm the internal validity either.

Conclusion validity Threats to conclusion validity concern the relation between the treatment and the outcome. We used un-biased systematic measures and the data generated by the approaches. Thus, we believe that no threat to the validity of our conclusion exists.

Reliability validity Threats to reliability validity concern the possibility of replicating this study. We attempted here to provide all the necessary details of our study to replicate it. Moreover, all studied programs and data from the previous approaches are publicly available or available upon request to their authors.

External validity Threats to external validity concern the possibility to generalize our findings. We studied six programs of different sizes, belonging to different domains, and evaluated by the previous approaches. The results on them are consistent and thus, mitigate the threat to generalizability. However, we only analysed Java code. Therefore, it is possible that the results of this study would be different on other programming languages. Further validation on a larger set of programs are desirable.

5.5 Conclusion

We present a study to investigate the effectiveness of four features and of their combinations to build API change rules during framework evolution. We select six metrics of these features used by previous approaches: call-dependency confidence value, comment LCS, inheritance tree LCS, method-level distance, method signature LCS, and method signature LD. We compare the change rules built by six approaches using each feature metric, five approaches using multi-features in prioritised ways and five corresponding MOOP-based approaches using the same multi-features. We discover the features really beneficial to build API change rules and a more effective way of combining different features.

We evaluate the approaches with six open-source Java frameworks and the results showed that (1) approaches only using method signature LCS or LD have comparable results to multi-feature approaches; (2) approaches combining multi-features do not always provide better results than individual approaches; (3) approaches using more features do not always perform better than the approaches using less features. An approach prioritising all the feature metrics gives less accurate results than approaches using a single feature on some subject frameworks; (4) MOOP-based approaches provide more stable results than the corresponding prioritised approaches using the same features. On average, MOOP-based approaches detect 13% more correct change rules with 3% smaller average correct replacement position than the corresponding prioritising approaches. $M5$, the MOOP-based approach using all the features, produces the best results in most cases.

Also, we compare $M5$ with a state-of-the-art approach MADMatch and $M5$ built more change rules with correct replacements on two of four frameworks and the same number of change rules on the other two when compared to MADMatch. Thus, we conclude that the MOOP-based approach can replace advantageously a more complex approach, which is hard to extend and requires parameter tuning.

We conclude that features have different effectiveness to build API change rules. Combining different features can produces better results than using single feature but they depend on the features and the combinations thereof. MOOP is an effective way to combine multiple features, especially when there is no reason to favour certain features or specific ways to combine multiple features. A MOOP-based approach can handle the contradictory information between features and is easy to be extended to new features.

CHAPTER 6 CONCLUSION

Frameworks keep evolving and client programs must upgrade to new releases to use added features or to patch security vulnerabilities. New releases of frameworks do not always guarantee backward compatibilities and may change their APIs. It is time-consuming for client programs to adapt to the API changes.

The motivations of this dissertation are: (1) helping developers and researchers better understand the reality of API changes and usages together at large-scale and fine-grained level and (2) proposing better approaches to build API change rules for missing APIs caused by framework evolution. The thesis of this dissertation is:

> Following analyses of the reality of API changes and usages, of the usefulness of API change rules, and of the effectiveness of the features used to build these rules, we can build more effective and extendible API change-rule recommendation tools.

6.1 Contributions

To prove our thesis, we first investigate the API changes and usages in the frameworks from Maven repository and two framework ecosystems: Apache and Eclipse. In total, only 9% of frameworks with changed APIs in 3% of their releases are used by client programs. However, these changed APIs are used by about half (49%) of all the client programs in about one fifth (21%) of their releases. The magnitude of the influence of API changes is large. More than 29,000 releases of 5,845 client programs are directly affected by API changes.

At method level, 10% APIs changed during framework evolution and only 2% are marked as deprecated by framework developers. 52% of the methods marked as deprecated are not API methods. Missing classes and missing methods are the two most frequent API change types in frameworks. Client programs use 16% of the APIs provided by the frameworks and 3% them are affected by API changes, but none of them are marked as deprecated. Missing classes and missing methods affect client programs more often as well.

On average, the API of one framework is used in 36% of client classes and interfaces, and more than 80% of such usages could be reduced through refactoring. About 18% of APIs are used in change-propagating ways.

Missing classes and missing methods are the most frequent API changes in frameworks and affecting client programs. Previous approaches build API change rules to recommend the

replacements for the missing APIs. However, the usefulness of these API change rules had not been confirmed empirically. We conduct an empirical study to verify if the API change rules generated by the previous approaches can help developers find the replacements of missing APIs more accurately or quickly, especially when these API change rules are not 100% correct.

The results of our study confirm that change rules built by previous approaches are useful, even when some of the change rules are incorrect. The higher precision the change rules have, the more help they provide. Thus, imperfect change rules can be used instead of unavailable documentation or as complement to partial documentation. Developers of frameworks could also use them as starting point to build upgrading documentation.

With the confirmation of the usefulness of API change rules, we want to improve the previous approaches to build API change rules. First, we investigate how useful each feature used in previous approaches is. We find that (1) approaches only using method signature have comparable results to multi-feature approaches, (2) previous approaches combining multi-features do not always provide better results than individual approaches and, (3) previous approaches using more features do not always perform better than those using less features.

The possible decrease in the results of previous multi-feature approaches are caused by the conflicting recommendations given by the features. Thus, we propose multi-objective-optimization-based approaches to build API change rules to overcome the limitation of previous multi-feature approaches and conduct an experiment to compare these two types of approaches. The comparison shows that multi-objective-optimization-based approaches provide more stable results than the corresponding prioritising approaches using the same features. Thus, multi-objective-optimization-based approaches can replace advantageously previous approaches. Also multi-objective-optimization-based approaches require less configurations and are easy to extend with new features.

We thus conclude that it is possible to implement more efficient and extendible approaches with multi-objective-optimization techniques to build API change rules. All the replication packages related to our study can be found on line [1].

6.2 Future Work

Based on the results of our studies, the following extensions are possible future work. We present them from short-term and long-term perspectives.

1. http://www.wuweidavid.net/replications.html

6.2.1 Short-term

We suggest the four possible directions to which the current research on framework API evolution could follow.

Extensive Qualitative Analyses

Our exploratory study on API changes and usages is mainly quantitative. The results help developers better understand the scale of API changes and the severity of the changes affecting the client programs. Qualitative analyses on API changes and usages on a large scale will lead more understanding. Generally, qualitative analyses are still conducted by researchers manually and are difficult to extend to large number of frameworks and client programs. Our quantitative analyses can help researchers locate the possible interesting frameworks on which researchers would apply qualitative analyses, such as the studies of Hou and Yao (2011) and Roover et al. (2013). Also, the results of quantitative analyses can help developers conduct controlled experiments on framework API changes and usages by providing data to select experimental and control groups.

Tools to Help Framework Upgrading

The API change and usage reports generated by ACUA provide the basis for developers to estimate upgrading cost before making decisions. With the detailed information in the reports, researchers can develop dedicate tools to help framework upgrading, automatically or requiring involvement from client program developers. For example, tools to help client program developers apply Adapter or Facade patterns to control API change-propagation, or tools to help framework developers provide documentations for API changes automatically.

Inter-framework API Change Rule Building

The assumption of the current formulation of API change rule building is that the missing methods are replaced by methods defined in the new releases of frameworks. Future research could extend to find inter-framework replacements, *i.e.*, replacements defined in other frameworks. Also, if the search space of replacements is extended to multiple frameworks, exhaustive search may not be feasible and we may have to use meta-heuristic search to build API change rules. Furthermore, researchers should evaluate different multi-objective-optimization algorithms (Panerati and Beltrame, 2014) and implementations, such as PyGMO (ESA, 2013) and jMetal (Durillo and Nebro, 2011), and choose them according the application contexts.

More Effective Features to Build API Change Rules

We find that the features used in API change rule building have different effectiveness. Combining features using multi-objective-optimization techniques is more effective than previous approaches, especially when there is no reason to favour certain features or specific ways to combine them. The performance of multi-objective-optimization-based approaches still depends on the used features. If the replacement of a missing API is not dominant according to the features, multi-objective-optimization-based approaches will miss it. Thus, researches could now focus on finding which features to combine and new more effective features.

Approaches on Other Steps of Upgrading Process

Our second empirical study shows that the change rules generated by framework API evolution approaches are useful. The imperfect change rules can be used instead of unavailable documentation or as complement to partial documentation. Developers of frameworks could also use them as starting point to build upgrading documentation. API change rules mainly help developers to find replacements of missing APIs more accurately. However, identifying the replacements of missing APIs is only one step in client program upgrading process. Studies proposing approaches to help developers in other steps, such as code changing and testing, are important.

6.2.2 Long-term

Base on the literature, our studies and observations, we argue that the following changes will happen to frameworks and their APIs.

Simplification

Simplification can come in both frameworks and client programs. On the framework side, we find that only 16% of the APIs provided by frameworks are used by client programs. This phenomenon shows that the current ways to expose APIs based on visibility is not efficient. Large number of unused APIs is distracting and also increase the maintenance and documentation workload. The Provisional API Guideline of Eclipse divides the APIs of frameworks into official and internal. This practice helps developers focus on the APIs that frameworks provide by design. Not all framework providers follow the same strategy. Language-supported API visibility, besides public, protected, default, and private, could be an effective solution. On the client program side, we find that 80% of API usages in client programs can be reduced by applying refactoring. Optimisation to keep API usage level to minimum could be achieved automatically by refactoring tools at compilation time to save client program developers' effort in upgrading tasks.

Standardization

Standardization can make frameworks easier to use and provide more alternatives of the same function to client programs. As the components used in computer hardware and auto-mobile manufacture, many frameworks provide similar functions, such as database access, needed by wide ranges of client programs. However, there are still not many standards to specify the APIs of frameworks, like JDBC. If there were more such standards, client programs could switch between the frameworks providing the same functions from different suppliers easily. Nowadays, open-source software gives a catalogue of frameworks for developers to choose, but the APIs of these frameworks are still not standards. More widely-accepted standards of functions and APIs could reduce software development and maintenance costs.

Verification

Verification, specially automated, will become more feasible when frameworks are standardised. Framework providers could supply the specifications of their implementations, such as reliability, execution time, memory usage, energy consumption, *etc.* Third-party organizations could independently verify if these implementations comply with the standards and compare them. Client program developers can also use and switch frameworks as off-the-shelf products of different brands according to their requirements and budgets.

REFERENCES

M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, ser. CSMR '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 181–190.

R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1993, pp. 207–216.

N. Ali, Z. Sharafi, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study on requirements traceability using eye-tracking," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM 2012, 2012.

G. Antoniol, M. D. Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*. IEEE Computer Society, 2004, pp. 31–40.

Apache, "Struts," 2014. [Online]. Available: http://struts.apache.org/

——, "Log4j," 2014. [Online]. Available: http://logging.apache.org/log4j

V. Arnaoudova, L. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Guéhéneuc, "Repent: Analyzing the nature of identifier renamings," *IEEE Transactions on Software Engineering (TSE)*, vol. 40, no. 5, 2014.

R. D. Baker, "Modern permutation test software," in *Randomization Tests*, E. Edgington, Ed. Marcel Dekker Incorporated, 1995.

G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 280–289.

G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero, "Understanding the shape of java software," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 397–412.

L. Beckwith, M. Burnett, S. Wiedenbeck, C. Cook, S. Sorte, and M. Hastings, "Effectiveness of end-user debugging software features: Are there gender issues?" in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '05, New York, NY, USA, 2005, pp. 869–878.

B. Bentmann and J. V. Zyl, "Eclipse aether," 2012. [Online]. Available: http://www.eclipse.org/aether/

J. Bloch, *Effective Java (2nd Edition) (The Java Series)*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

E. Bruneton, E. Kuleshov, A. Loskutov, , and R. Forax, "Asm framework," 2000. [Online]. Available: http://asm.ow2.org/

M. Buchholz, "Kinds of compatibility: Source, binary, and behavioral," 2008. [Online]. Available: https://blogs.oracle.com/darcy/entry/kinds_of_compatibility

J. Businge, A. Serebrenik, and M. van den Brand, "An empirical study of the evolution of eclipse third-party plug-ins," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ser. IWPSE-EVOL '10. New York, NY, USA: ACM, 2010, pp. 63–72.

——, "Survival of eclipse third-party plug-ins," in *ICSM*, 2012, pp. 368–377.

——, "Analyzing the eclipse api usage: Putting the developer in the loop," in *CSMR*, 2013, pp. 37–46.

——, "Eclipse api usage: the good and the bad," *Software Quality Journal*, pp. 1–35, 2013.

K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *: Proceedings of the 1996 International Conference on Software Maintenance*, ser. ICSM 1996. Washington, DC, USA: IEEE Computer Society, 1996, p. 359.

J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 55:1–55:11.

CRA, "Faq on heartbleed bug," 2014. [Online]. Available: http://www.cra-arc.gc.ca/gncy/fq-hb-eng.html

B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 481–490.

——, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 4, pp. 19:1–19:35, Sep. 2011.

K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, August 2002.

J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *CSMR-WCRE*, 2014, pp. 64–73.

D. Dig and R. Johnson, "How do apis evolve? a story of refactoring: Research articles," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 83–107, 2006.

D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen, "Refactoring-aware configuration management for object-oriented programs," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 427–436.

J. J. Durillo and A. J. Nebro, "jmetal: A java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, pp. 760–771, 2011.

Eclipse, "http://www.eclipse.org," 2009. [Online]. Available: http://www.eclipse.org

——, "Eclipse project archieves," 2014. [Online]. Available: http://archive.eclipse.org/eclipse/downloads/

ESA, "Python parallel global multiobjective optimizer," 2013. [Online]. Available: http://esa.github.io/pygmo/

T. Espinha, A. Zaidman, and H.-G. Gross, "Web api growing pains: Stories from client developers and their code," in *CSMR-WCRE*, 2014, pp. 84–93.

R. H. Fagard, J. A. Staessen, and L. Thijs, "Advantages and disadvantages of the meta-analysis approach," *Journal of Hypertension*, vol. 14, no. 2, September 1996.

A. Finkelstein, M. Harman, S. A. Mansouri, J. Ren, and Y. Zhang, "A search based approach to fairness analysis in requirement assignments to aid negotiation, mediation and decision making," *Requir. Eng.*, vol. 14, no. 4, pp. 231–245, 2009.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 166–181, 2005.

Google, "Android os," 2007. [Online]. Available: http://www.android.com/

R. Grissom and J. Kim, *Effect sizes for research: a broad practical approach*. Lawrence Erlbaum Associates, 2005.

S. Gueorguiev, M. Harman, and G. Antoniol, "Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, ser. GECCO 2009. New York, NY, USA: ACM, 2009, pp. 1673–1680.

D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, Dec. 2012.

S. G. Hart and L. E. Staveland, "Nasa task load index (tlx) v 1.0," http://humansystems.arc.nasa.gov/groups/TLX/downloads/TLX.pdf, 1988.

S. G. Hart and L. E. Stavenland, "Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research," pp. 139–183, 1988.

J. Henkel and A. Diwan, "Catchup!: capturing and replaying refactorings to support api evolution," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM, 2005, pp. 274–283.

D. Hou and X. Yao, "Exploring the intent behind api evolution: A case study," in *WCRE*, 2011, pp. 131–140.

IBM, "Eclipse marketplace," 2006. [Online]. Available: http://marketplace.eclipse.org/

jEdit, "jedit - programmer's text editor," 2014. [Online]. Available: http://www.jedit.org/

jFreechart, "Jfreechart," 2014. [Online]. Available: http://www.jfree.org/jfreechart/

jHotDraw, "Jhotdraw," 2014. [Online]. Available: http://www.jhotdraw.org

T. Joachims, L. Granka, B. Pan, H. Hembrooke, and G. Gay, "Accurately interpreting clickthrough data as implicit feedback," in *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, ser. SIGIR '05. New York, NY, USA: ACM, 2005, pp. 154–161.

D. Kawrykow and M. P. Robillard, "Improving api usage through automatic detection of redundant code," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 111–122.

C. Kemper and C. Overbeck, "What's new with jbuilder," in *JavaOne Sun's 2005 Worldwide Java Developer Conference*, 2005.

M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *ICSE '07: Proceedings of the 29th international*

conference on Software Engineering. Washington, DC, USA: IEEE Computer Society, 2007, pp. 333–343.

S. Kim, K. Pan, and E. J. Whitehead, Jr., "When functions change their names: Automatic detection of origin relationships," in *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 143–152.

S. Kpodjedo, F. Ricca, P. Galinier, G. Antoniol, and Y.-G. Guéhéneuc, "Madmatch: Many-to-many approximate diagram matching for design comparison," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1090–1111, 2013.

R. Lämmel, R. Linke, E. Pek, and A. Varanovich, "A framework profile of .net," in *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, 2011, pp. 141–150.

R. Lämmel, E. Pek, and J. Starek, "Large-scale, ast-based api-usage analysis of open-source java projects," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 1317–1324.

D. Lawrie, H. Feild, and D. Binkley, "Syntactic identifier conciseness and consistency," in *Sixth IEEE International Workshop on Source Code Analysis and Manipulation.*, Sept. 2006, pp. 139–148.

D. Lawrie, C. Morrell, H. Feild, and D. W. Binkley, "Effective identifier names for comprehension and memory," *ISSE*, vol. 3, no. 4, pp. 303–318, 2007.

V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

R. Likert, "A technique for the measurement of attitudes," *Archives of Psychology*, vol. 22, no. 140, pp. 1–55, 1932.

C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceeding of the 33rd international conference on Software engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 111–120.

S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *Proceedings of 34th International Conference on Software Engineering*, ser. ICSE 2012, 2012, pp. 353–363.

J. Meyers-Levy, *Gender Differences in Information Processing: A SelectivityInterpretation.* P. Cafferata and A. Tybout, (Eds) Cognitive and Affective Responses to Advertising. Lexington Books, 1989.

L. Mikhajlov and E. Sekerinski, "A study of the fragile base class problem." in *IN EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING*, ser. ECOOP'98. Springer, 1998, pp. 355–382.

G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *The Psychological Review*, vol. 63, no. 2, pp. 81–97, March 1956.

R. G. J. Miller, *Simultaneous Statistical Inference, 2nd edition*. Springer, 1981.

J. E. Montandon, H. Borges, D. Felix, and M. T. Valente, "Documenting apis with examples: Lessons learned with the apiminer platform," in *WCRE*. IEEE, 2013, pp. 401–408.

H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 302–321.

E. O'Donnell and E. Johnson, "The effects of auditor gender and task complexity on information processing efficiency," *International Journal of Auditing*, vol. 5, no. 2, pp. 91–105, 2001.

J. Panerati and G. Beltrame, "A comparative evaluation of multi-objective exploration algorithms for high-level design," *ACM Trans. Design Autom. Electr. Syst.*, vol. 19, no. 2, p. 15, 2014.

G. C. Porras and Y. Guéhéneuc, "An empirical study on the efficiency of different design pattern representations in UML class diagrams," *Empirical Software Engineering*, vol. 15, no. 5, pp. 493–522, 2010.

S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ser. ICSM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 378–387.

D. Rivières, "Evolving java-based apis 2," 2008. [Online]. Available: http://wiki.eclipse.org/Evolving_Java-based_APIs_2

R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation?: The case of a smalltalk ecosystem," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 56:1–56:11.

C. D. Roover, R. Lammel, and E. Pek, "Multi-dimensional exploration of api usage," in *ICPC*, 2013, pp. 152–161.

M. O. Saliu and G. Ruhe, "Bi-objective release planning for evolving software systems," in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, 2007, pp. 105–114.

Y. Sawaragi, H. Nakayama, and T. Tanino, *Theory of multiobjective optimization.* Academic Press, 1985.

T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *ICSE '08: Proceedings of the 30th international conference on Software engineering.* New York, NY, USA: ACM, May 2008, pp. 471–480.

Z. Sharafi, Z. Soh, Y.-G. Guéhéneuc, and G. Antoniol, "Women and men - different but equal: On the impact of identifier style on source code reading," in *ICPC*, 2012, pp. 27–36.

B. Sharif and J. I. Maletic, "An eye tracking study on camelcase and underscore identifier styles," in *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ser. ICPC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 196–205.

Z. Soh, Z. Sharafi, B. V. den Plas, G. C. Porras, Y. Guéhéneuc, and G. Antoniol, "Professional status and expertise for UML class diagram comprehension: An empirical study," in *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, 2012, pp. 163–172.

E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010, pp. 336–345.

S. Thummalapenta and T. Xie, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 327–336.

J. van Zyl and Apache, "The maven central repository," 2005. [Online]. Available: http://search.maven.org/

C. Wohlin, P. Runeson, and M. Höst, *Experimentation in Software Engineering: An Introduction.* Springer, 1999.

W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 325–334.

W. Wu, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "Acua: Api change and usage auditor," in *The 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM'14, 2014.

W. Wu, A. Serveaux, Y.-G. Guéhéneuc, and G. Antoniol, "The impact of imperfect change rules on framework api evolution identification:an empirical study," *Empirical Software Engineering*, july 2014.

Z. Xing and E. Stroulia, "Refactoring detection based on umldiff change-facts queries," in *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 263–274.

——, "API-evolution support with diff-CatchUp," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 818 – 836, December 2007.

——, "Differencing logical uml models," *Autom. Softw. Eng.*, vol. 14, no. 2, pp. 215–259, 2007.

S. Yusuf, H. Kagdi, and J. I. Maletic, "Assessing the comprehension of uml class diagrams via eye tracking," in *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ser. ICPC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 113–122.

Y. Zhang, M. Harman, and S. A. Mansouri, "The multi-objective next release problem," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ser. GECCO 2007. New York, NY, USA: ACM, 2007, pp. 1129–1137.

E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, November 1999.