Towards the Automation of Migration and Safety of Third-Party Libraries

Hussein Ahmed Talib Al-Rubaye
hat6622@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Dissertation is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Towards the Automation of Migration and Safety of Third-Party Libraries

by

Hussein Ahmed Talib Al-Rubaye

A dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and Information Sciences

Rochester Institute of Technology
Rochester, New York
November, 2020
Towards the Automation of Migration and Safety of Third-Party Libraries

by

Hussein Ahmed Talib Al-Rubaye

Committee Approval:
We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Mohamed Wiem Mkaouer
Dissertation Advisor

Date

Naveen Sharma
Dissertation Committee Member

Date

Mehdi Mirakhorli
Dissertation Committee Member

Date

Ali Ouni
Dissertation Committee Member

Date

Dan Phillips
Dissertation Defense Chairperson

Date

Certified by:

Pengcheng Shi
Ph.D. Program Director, Computing and Information Sciences

Date
Towards the Automation of Migration and Safety of Third-Party Libraries

by

Hussein Ahmed Talib Al-Rubaye

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Ph.D. Program in Computing and Information Sciences in partial fulfillment of the requirements for the Doctor of Philosophy Degree at the Rochester Institute of Technology

Abstract

The process of migration from one library to a new, different library is very complex. Typically, the developer needs to find functions in the new library that are most adequate in replacing the functions of the retired library. This process is subjective and time-consuming as the developer needs to fully understand the documentation of both libraries to be able to migrate from an old library to a new one and find the right matching function(s) if exists. Our goal is helping the developer to have better experiences with library migration by identifying the key problems related to this process. Based on our critical literature review, we identified three main challenges related to the automation of library migration: (1) the mining of existing migrations, (2) learning from these migrations to recommend them in similar contexts, and (3) guaranteeing the safety of the recommended migrations.
Acknowledgments

I would like to thank everyone who helped to make this dream real. Starting with my lovely wife (Rana) and our four adorable girls Jena, Laya, Dora, and Yasmine. Thanks to my amazing advisor Dr. Mohamed and the committee members. Thanks to my former manager at Xerox Corporation, Jim Russell, for making this journey possible. Not to forget, Thanks to Starbucks where I wrote all creative work there :).

Thanks everyone,
Hussein,
## Contents

1 Introduction and Background ........................................... 1
   1.1 Introduction .......................................................... 1
   1.2 The Migration Dilemma ............................................. 3
   1.3 Background and Terminology ...................................... 6
   1.4 Research agenda ..................................................... 10
   1.5 Ph.D. Publications ................................................... 11

2 Literature Review ......................................................... 12
   2.1 Abstract ............................................................... 12
   2.2 Mining software quality and comprehension .................... 12
   2.3 Library API recommendation ....................................... 14
   2.4 Library Breaking Changes .......................................... 15
   2.5 Empirical evaluation of software quality and comprehension ........ 16

1 API Mining ................................................................. 18

3 Variability in Library Evolution ........................................ 19
3.1 Introduction ................................................................. 19
3.2 Motivation .............................................................. 21
3.3 Retrieval Phase ......................................................... 23
3.4 Detection of Library Evolution ..................................... 24
   3.4.1 Library Upgrade .................................................. 25
   Incremental Version. .................................................... 25
   Repackaging. ............................................................. 26
   3.4.2 Library Migration ................................................ 26
   Instant Migration ....................................................... 27
   Delayed Migration ...................................................... 28
   3.4.3 Detection of Candidate Library Migration .................. 29
3.5 Experimental Results ............................................... 32
   3.5.1 Studied Projects .................................................. 32
   3.5.2 RQ1. Existence of Libraries ................................. 32
   3.5.3 RQ2. Evolution of Libraries ................................. 34
   3.5.4 RQ3. Variability Degree of Libraries ...................... 37
   3.5.5 Discussion and future directions ........................... 39
3.6 Threats to Validity ................................................... 39
3.7 Conclusion ............................................................. 41

4 Automate the Detection of Third-Party Library Migration 42
   4.1 Abstract ............................................................... 42
4.2 Introduction ......................................................... 43
4.3 Substitution Algorithm .............................................. 45
  4.3.1 Collection Phase ............................................. 46
  4.3.2 Detection Phase ............................................. 46
  4.3.3 Mapping Phase .............................................. 47
  4.3.4 Validation Phase ............................................ 50
4.4 Experimental Design .............................................. 51
4.5 Results ............................................................ 52
  4.5.1 Results for RQ1 ............................................. 52
  4.5.2 Results for RQ2 ............................................. 53
  4.5.3 Discussions .................................................. 54
4.6 Conclusion ......................................................... 57

5 MigrationMiner Tool ............................................... 58
  5.1 Abstract .......................................................... 58
  5.2 Introduction ...................................................... 58
  5.3 MigrationMiner ................................................... 60
    5.3.1 Data Collection ........................................... 60
    5.3.2 Migration Detector ....................................... 61
    5.3.3 Segment Detection ....................................... 62
    5.3.4 Fragment Detection ....................................... 63
    5.3.5 Documentation Collector ................................. 63
6 MigrationMapper Tool

6.1 Abstract .............................................................................. 66
6.2 Introduction ......................................................................... 66
6.3 MigrationMapper Architecture .............................................. 69
   6.3.1 Data Collection ................................................................. 69
   6.3.2 Migration Detector .............................................................. 70
   6.3.3 Segment Detection .............................................................. 71
   6.3.4 Fragment Detection .............................................................. 72
   6.3.5 Documentation Collector .................................................... 72
   6.3.6 Substitution Algorithm ....................................................... 73
6.4 Validation ............................................................................. 76
   6.4.1 MigrationMapper Correctness ........................................... 76
   6.4.2 MigrationMapper vs. MigrationMiner ............................... 76
   6.4.3 Practical Impact ................................................................. 80
6.5 Conclusion and Future Work .................................................. 81
7 The Impact of API Migration on Software Quality ................. 82

7.1 Abstract .............................................................................. 82
7.2 Introduction ......................................................................... 83
7.3 Empirical Setup ................................................................... 86
7.3.1 Data Collection .......................................................... 86
7.3.2 Metrics Measurement .................................................. 87
   Structure and size metrics ................................................. 87
   Code readability metrics ................................................. 88
   Refactoring operations collection .................................... 88
7.3.3 Ranking Model .......................................................... 89
7.4 Results ............................................................................ 90
   7.4.1 RQ1. (Design Improvement) What is the impact of library migration on the quality of software design? .................................................. 91
   7.4.2 RQ2. (Code Readability) Does migration improve the code readability? ............................ 93
   7.4.3 RQ3. (Refactoring Operations) What types of refactoring changes do developers perform during library migration? ................................. 95
   7.4.4 RQ4. (Quality Recommendation) Can we leverage design and readability metrics to recommend better code examples of migration? ......................... 98
7.5 Threats to validity .......................................................... 99
   7.5.1 Internal Validity. ........................................................ 99
   7.5.2 Construct validity ..................................................... 100
   7.5.3 External validity ..................................................... 100
7.6 Conclusion and Future Work ............................................. 101

II API Recommendation ..................................................... 102

8 Learning to Recommend Third-Party Library Migration ............... 103
   8.1 Abstract ...................................................................... 103
## CONTENTS

8.2 Introduction .......................................................................................... 103

8.3 Methodology ......................................................................................... 107
  8.3.1 Data Collection .................................................................................. 107
  8.3.2 Text Engineering .............................................................................. 108
    Information Extraction (IE) .................................................................. 108
    Text Preprocessing (TPP) ..................................................................... 109
    Vector Space Representation ............................................................... 110
  8.3.3 Feature Engineering .......................................................................... 111
    Method Description $\varphi_1$ ................................................................. 111
    Return Type Description $\varphi_2$ ......................................................... 112
    Input Parameters Description $\varphi_3$ ................................................ 112
    Input Parameters Signature $\varphi_4$ .................................................... 113
    Return Type Signature $\varphi_5$ ............................................................ 113
    Method Name $\varphi_6$ .......................................................................... 113
    Number of Input Parameters $\varphi_7$ ................................................... 114
    Package Name $\varphi_8$ ........................................................................ 114
    Class Name $\varphi_9$ ............................................................................. 114
  8.3.4 Feature selection .............................................................................. 115
  8.3.5 Classifier Model .............................................................................. 115

8.4 Experimental Design ........................................................................... 117
  8.4.1 State of the art approaches ............................................................. 119
    Learning to Rank (LTR) ...................................................................... 119
CONTENTS

TMAP ......................................................... 119
Method Signature (MS) .............................. 120

8.4.2 Parameter Tuning .............................. 121

8.5 Results ............................................... 122

8.5.1 Results for RQ1 .............................. 122
8.5.2 Results for RQ2 .............................. 126
8.5.3 Discussion and limitations ................... 126

8.6 Threats to validity ............................ 128

8.6.1 Internal validity .......................... 128
8.6.2 Construct validity ......................... 129
8.6.3 External validity ......................... 129

8.7 Conclusion ....................................... 129

9 Recommendation Using Domain And Source Code Knowledge 131

9.1 Abstract ......................................... 131
9.2 Introduction and Motivation .................. 132
9.3 Methodology .................................. 135

9.3.1 Domain knowledge ...................... 136

Extract Method Mapping Documentation .......... 136
Feature Engineering ................................. 137

9.3.2 Source code knowledge .................. 139

Extract Method Mapping Source Code .......... 139
CONTENTS

Mapping Code2Vector ................................................. 140
Principal Component Analysis ..................................... 140
CNN Learner ............................................................. 141
9.3.3 Classifier Model .............................................. 142

9.4 Experimental Design ........................................... 144
9.4.1 Parameter Tuning ............................................... 145

9.5 Results .............................................................. 146
9.5.1 Results for RQ1. ................................................ 146
9.5.2 Results for RQ2. ................................................ 148
9.5.3 Discussion and limitations ................................. 149

9.6 Conclusion .......................................................... 151

III System Behavior Safety ............................................. 152

10 System Behavior Safety of Migration ......................... 153

10.1 Introduction ...................................................... 153

10.2 Methodology ..................................................... 154
10.2.1 Migration Schema ........................................... 155
10.2.2 Parser ............................................................ 156
10.2.3 Auto Migration ................................................ 156
10.2.4 Limitations ..................................................... 156

10.3 Experiment settings ............................................. 157

10.4 Results Discussion ................................................ 158
10.5 Conclusion and Future Works ............................................................... 161
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Average time spent by developers to perform the migration between pairs of popular Java libraries.</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Migration period, clustered by developers years of experience.</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Samples of migration between json and gson.</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>PhD. timeline.</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Bird’s view of our data extraction process.</td>
<td>24</td>
</tr>
<tr>
<td>3.2</td>
<td>Upgrading log4j from 1.2.13 to 1.2.15 in the docx4j project.</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>Upgrading jetty-all library in the Apache hive project.</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Migrating from junit to testng and fest-assert-core in the ps2-parser project.</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Replacing junit functions by fest-assert-core functions in the ps2-parser project5.</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Replacing junit functions by testng functions in the ps2-parser project5.</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>Replacing org.redmine.ta.internal.json functions by to org,jason:jason functions in the redmine-java-api project.</td>
<td>29</td>
</tr>
<tr>
<td>3.8</td>
<td>Overview of the library migration detection heuristic.</td>
<td>30</td>
</tr>
<tr>
<td>3.9</td>
<td>Detection of candidate migration pairs.</td>
<td>31</td>
</tr>
<tr>
<td>3.10</td>
<td>Top actively updated libraries.</td>
<td>34</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.11</td>
<td>Topmost version-upgraded libraries.</td>
<td>36</td>
</tr>
<tr>
<td>3.12</td>
<td>Comparison of top most performed migrations between Teyton et al. and our study</td>
<td>37</td>
</tr>
<tr>
<td>3.13</td>
<td>Comparison between VD values of source and target libraries involved in the top most performed migrations</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>Approach Overview.</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>SA illustrative example using 4 fragments of migration between json and gson as input</td>
<td>49</td>
</tr>
<tr>
<td>4.3</td>
<td>f-measure using randomly created 5-1500 fragments for 5, 10, and 20 method mapping, over 30 runs.</td>
<td>54</td>
</tr>
<tr>
<td>4.4</td>
<td>Example of &quot;unresolved&quot; Fragments.</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>MigrationMiner workflow and Architecture.</td>
<td>60</td>
</tr>
<tr>
<td>5.2</td>
<td>Migration from <code>json</code> to <code>gson</code>.</td>
<td>61</td>
</tr>
<tr>
<td>5.3</td>
<td>Library Migration Detector.</td>
<td>62</td>
</tr>
<tr>
<td>5.4</td>
<td>API documentation for <code>toJson(JsonElement)</code></td>
<td>64</td>
</tr>
<tr>
<td>6.1</td>
<td>MigrationMapper workflow and Architecture.</td>
<td>69</td>
</tr>
<tr>
<td>6.2</td>
<td>Migration from <code>json</code> to <code>gson</code>.</td>
<td>70</td>
</tr>
<tr>
<td>6.3</td>
<td>Library Migration Detector.</td>
<td>71</td>
</tr>
<tr>
<td>6.4</td>
<td>API documentation for <code>toJson(JsonElement)</code></td>
<td>73</td>
</tr>
<tr>
<td>6.5</td>
<td>MigrationMiner output</td>
<td>77</td>
</tr>
<tr>
<td>6.6</td>
<td>MigrationMapper output</td>
<td>78</td>
</tr>
<tr>
<td>6.7</td>
<td>MigrationMapper website users analytics.</td>
<td>80</td>
</tr>
<tr>
<td>7.1</td>
<td>Experimental Design Overview</td>
<td>86</td>
</tr>
<tr>
<td>7.2</td>
<td>Box plots of CBO, LCOM, and average CycC values, extracted from migrated code fragments, before and after the migration (lower values are better).</td>
<td>91</td>
</tr>
<tr>
<td>7.3</td>
<td>Illustrative example of a code migration from log4j to slf4j, with a positive impact on coupling.</td>
<td>92</td>
</tr>
<tr>
<td>7.4</td>
<td>Illustrative example of a code migration from async-http-client to httpclient, with a positive impact on cohesion.</td>
<td>93</td>
</tr>
<tr>
<td>7.5</td>
<td>Box plots of BWR and SR values, extracted from migrated code fragments, before and after the migration (higher values are better).</td>
<td>94</td>
</tr>
<tr>
<td>7.6</td>
<td>Illustrative example of a code migration from json to gson, with a positive impact on readability.</td>
<td>94</td>
</tr>
<tr>
<td>7.7</td>
<td>Distribution of refactoring operations per type, before and after the migration.</td>
<td>96</td>
</tr>
<tr>
<td>7.8</td>
<td>Illustrative example of a code migration from json to gson, being supported by applying a change parameter type refactoring.</td>
<td>97</td>
</tr>
<tr>
<td>7.9</td>
<td>Percentages of the match between developers choices and the example@k.</td>
<td>99</td>
</tr>
<tr>
<td>8.1</td>
<td>The proposed RAPIM approach for method APIs mapping recommendation.</td>
<td>106</td>
</tr>
<tr>
<td>8.2</td>
<td>Comparative study between learners, in terms of accuracy.</td>
<td>115</td>
</tr>
<tr>
<td>8.3</td>
<td>Error with tuning.</td>
<td>120</td>
</tr>
<tr>
<td>8.4</td>
<td>ROC Curve for BDT with and without tuning.</td>
<td>121</td>
</tr>
<tr>
<td>8.5</td>
<td>Performance of approaches under test, in terms of accuracy, across 8 migrations.</td>
<td>122</td>
</tr>
<tr>
<td>8.6</td>
<td>Samples of method mappings between json and gson.</td>
<td>123</td>
</tr>
<tr>
<td>8.7</td>
<td>Impact of training data size.</td>
<td>127</td>
</tr>
<tr>
<td>9.1</td>
<td>The proposed DSKR approach for method APIs mapping recommendation.</td>
<td>134</td>
</tr>
<tr>
<td>9.2</td>
<td>CNN Learner</td>
<td>141</td>
</tr>
</tbody>
</table>
9.3 Comparative study Different number of features and accuracy. .................. 143
9.4 ROC Curve for BDT with and without tuning. ................................. 146
9.5 Performance of approaches under test, in terms of accuracy, across 8 migrations. .. 147
9.6 Samples of method mappings between json and gson. ....................... 148
9.7 Impact of training data size. ......................................................... 149
10.1 Auto Migration Methodology ....................................................... 154
10.2 Type Change Issue ................................................................. 157
10.3 Sample of Passed Test Case ...................................................... 160
10.4 Sample of Failed Test Case ...................................................... 160
List of Tables

3.1 Projects Overview ................................................................. 33
3.2 Topmost remodularized libraries ............................................ 35
3.3 Topmost performed migrations .............................................. 35
4.1 TPR and F-measure of the method mappings. ............................... 53
4.2 Number of library documentation TPR. ................................... 56
5.1 Accuracy of Migration Miner. .................................................. 65
6.1 Accuracy of MigrationMapper. ............................................... 76
6.2 Average fragment size. ......................................................... 79
7.1 Dataset overview. ................................................................. 87
7.2 Statistical significance of samples difference, before and after API migration, for each of the considered metrics. ................................. 90
8.1 Performance of RAPIM, in terms of Precision and Recall, across 3 migrations. ................................. 125
Chapter 1

Introduction and Background

1.1 Introduction

Many software systems depend on third-party libraries to offer robust services. These services save developers a tremendous amount of time. Developers often need to replace one library with another library while maintaining the same functionality. These new libraries are more efficient and thus, perform better. Also, they include additional features and services while being less complex to maintain. This process of replacing one library with a completely different library, while preserving the same functionality, is called library migration. This phenomenon has been studied by Teyton [77] [76]. This process should not be confused with the process of upgrading one library to a more recent version. Upgrading a library to a newer version is referred to as a library upgrade and has been studied by [43] [57].

When a developer wants to migrate from library A to library B, he/she needs to find functions in the new library that offer the same behavior of the functions used in the current library. The functionality of both libraries must be the same. This process is very time-consuming because the developer needs to fully understand how both libraries work and function. The developer needs to fully read the documentation of both libraries to understand how to map functions from the old library to functions in the new library.

Ideally, software engineers urgently need a comprehensive approach that takes as input a new library to replace an existing one then provides all the possible automated function migrations while testing the intactness of the semantics, and warns about any transformations that violate the
client code’s behavior. This can be achieved by combining multiple disciplines to solve a specific one. This is why our research aims to address the following research questions:

- **Research Question 1:** How do developers migrate between different APIs?

To answer the following research question, we have conducted a large mining study to extract existing, manually performed migrations, between famous Java third-party libraries. The extracted migrations demonstrate (1) the difficulty of finding the appropriate functions to replace the ones to be retired, (2) subjectivity of the developer choices, since multiple functions can be a candidate to replace an existing one, so developers make different decisions based on different scenarios, and (3) the possibility to replace one function from the retired library with more than one function from the new library, which raises the complexity of the mining process.

- **Research Question 2:** What API features developers rely on when migrating between APIs? How can we learn from these features to recommend better APIs for developers?

Existing studies have shown that developers use multiple preferences when choosing the appropriate API for their codebase. They mainly focus on finding the API that offers similar, yet better functionalities than the library they are retiring. Existing works focus on information retrieval and lexical similarity to promote a candidate replacing the library. We will challenge these techniques by testing them against our mined dataset of existing migrations, and we plan on using this dataset to learn what features are more relevant to this problem.

- **Research Question 3:** How do developers ensure that the code semantics are not violated when conducting the migration?

Addressing this research question is essential for the automation of migration, any introduced functions during this process need to preserve the system’s overall behavior. Mining the existing regression testing practices are necessary to better understand how the test case selection is performed.
1.2 The Migration Dilemma

The migration process between two different libraries is a hard, error-prone and time-consuming process [5,9,20,45].

![Figure 1.1: Average time spent by developers to perform the migration between pairs of popular Java libraries.](image)

To showcase the complexity of the process, we measure, based on the data we collected in our experiments, the average time spent to perform migration between different libraries as shown in Figure 1.1. We approximate the migration time by calculating the difference between timestamps of the first and last commit that contained migrations, as the migration is typically performed in multiple commits [5]. Figure 1.1 shows that, depending on how complex is the migration, developers typically spend from 2 to 42 days to migrate between libraries.

Typically, software development companies tend to assign migration tasks to developers who have more experience to reduce regression risks. For instance, Figure 1.2 shows that developers who have more than ten years of experience are expected to perform migration more often than a new developer with less than five years of experience. The figure is based on a previous migration benchmark by Alrubaye et al. [5] which contains information about the developers who have performed migration tasks previously, such as, developer names, emails, years of experience, and migration dates.

Furthermore, on an ideal setting, each source library method is replaced with one target library method (one-to-one), in each fragment. This makes their detection easier and less error-prone. In
practice, due to the differences in libraries design, separation of concerns, and naming conventions, a method may be replaced with more than one method from the target library \(\text{one-to-many}\). Furthermore, there exists different co-locations of many added and removed methods within the same source block, which makes the automated identification of individual mappings more complex.

As a motivating example, depicted in Figure 1.3, we consider three fragments from Github\(^1\) that were extracted as part of the migration from \textit{json} to \textit{gson}. Each fragment contains a replacement scenario that is described as follows:

- **One-to-one mapping.** It is the replacement of one method with another method. In Figure 1.3-(A), the method \texttt{put(key, value)} is replaced by one method, namely \texttt{addProperty(key, value)}.

- **One-to-many mapping.** Replacing one method with more than one method. In Figure 1.3-(B), the method \texttt{put(key, value)} has been replaced with two methods, namely \texttt{addProperty(key, value)}.

\(^1\)http://migrationlab.net/redirect.php?cf=icpc2019&p=1
public void addKeyValues(String key, int value) {
    checkIfKeyDescriptionExist(key);
    - keyValues.put(key, value);
    + keyValues.addProperty(key, value);
}

(A) One-to-one

public void addKeyValues(String key, Map value) {
    checkIfKeyDescriptionExist(key);
    - keyValues.put(key, value);
    + keyValues.addProperty(key, new Gson().toJson(value));
}

(B) One-to-many

- if (!endpoints.isEmpty()) {
    - node.put("endpoints", endpoints);
+ if (!endpoints.isJsonNull()) {
+    node.add("endpoints", endpoints);

(C) Many-to-many

Figure 1.3: Samples of migration between json and gson.

value), and Gson().toJson(value). To have valid input for addProperty method, the Map object needs to be converted into a json object, so another converting method was added. Note that, in Figure 1.3(A), the same method was replaced with only one method since the input Value had no mismatch.

- Many-to-many mapping. Replacing many methods (two or more) with two methods (two or more). As a real-world example, Figure 1.3-(C) shows how the two methods isEmpty(), and put(key, value) have been replaced with two different methods, namely isJsonNull() and add(key, value).

- Multiple correct mappings. A method from the removed library could be mapped to more than one method from the added library. For example, In Figure 1.3-(A), the method put(key, value) was replaced by addProperty(key, value), while in Figure 1.3-(C), the same method was replaced by add(key, value). The reason behind the developer’s decision could be related to opting a more relative method that requires fewer changes. Otherwise, if the developer
decides to replace \texttt{put(key, value)} with \texttt{addProperty(key, value)}, then another method would likely to be added, namely \texttt{getEndpoint()} to get Integer value of endpoints, so the added code would be \texttt{addProperty("endpoints", endpoints.getEndpoint())}. So there are multiple possible correct mappings for a given method. This scenario indicates how a migration is merely a subjective process, and developers tend to choose simpler \textit{one-to-one} mappings, whenever it is possible to reduce the amount of unnecessary changes. We follow the same intuition in our approach as we opt for mappings with lower cardinality, as much as possible during our mapping generation process.

As shown in the motivating example, a large number of type mappings could be extracted from one single code change, \textit{i.e.}, commit. This is a particularly challenging task to generate accurate and relevant library method mappings to support the library migration process. Indeed, existing state-of-the-art approaches relying basically on lexical similarity achieved a limited accuracy in identifying \textit{one-to-many} or \textit{many-to-many} mappings. To address these limitations, our approach intersects fragments to generate all possible mapping types between methods, then calculates the frequency of each mapping across all fragments. Intuitively, the higher the frequency of a mapping is, the more relevant it is for the migration. Furthermore, we intersect mappings with lower cardinality (\textit{one-to-one}) with those having higher cardinality (\textit{one-to-many} and \textit{many-to-many}) in order to reduce their cardinality. Additionally, we use the similarity between the documentation of API methods as it would provide a rich and meaningful information to reduce the cardinality of \textit{many-to-many} mappings by extracting a \textit{one-to-one} mapping particularly in cases where the combination of the methods documentation exhibit a strong similarity.

\section{Background and Terminology}

This section presents definitions of keywords that are used throughout this dissertation.

\textbf{Library}. A library encapsulates a set of resources, in the form of objects and functions, publicly accessible through the library’s Application Programming Interface (API). Just like any traditional software, a library has multiple releases. Note that in this study, we identify libraries by the composition of their GroupID, ArtifactID and version, but since we are interested in the in-between library evolution, we label libraries by their artifactID for the sake of simplicity.

\textbf{Library Migration}. A library migration occurs when a source library is replaced by a target library. The source library is considered retired if all of its method dependencies are removed from its client.
code. Note that the source library does not need to be physically removed from the project (e.g., the pom files for Maven projects, or local libraries repository), but it enforces that none of its methods are used in the client implementation.

**Naming Scheme.** It serves the purpose of providing unique identifiers to libraries. Apache Maven adopts the following naming scheme:

\[
< \text{GroupID} > . < \text{ArtifactID} > . < \text{Version} >
\]

**GroupID.** It identifies the developer of the library. It is required to follow the package naming convention and thus it is unique as it represents the domain or subdomain the library provider owns. e.g., org.apache.maven.

**ArtifactID.** It is the name of the library, it must be unique within the set of libraries belonging to the same provider, i.e., two libraries belonging to two different providers can have the same ArtifactID since they can be distinguished by the GroupID.

**Version.** Consider Each library \( l \in L \) has multiples releases throughout its lifecycle i.e., we consider a chronologically ordered list of version numbers \( v \) such that \( v \in R | l_v \in L \). There are several version numbering schemes recommended by the development communities, for instance, the Apache Maven project recommends the use of the following versioning strategy:

\[
< \text{major} > . < \text{minor} > . < \text{patch} > [- < \text{type} > - < \text{attempt} >]
\]

Where **major** represents a significant release of the library, such as adding new functionalities. **Minor** indicates few changes, usually related nonfunctional requirements such as performance optimization, etc. **Patch** refers to bug fixes and security patches etc. **type-attempt** refers to the maturity level of the released version, e.g., alpha, beta, stable, etc.

It is to note that offered classes and interfaces offered by the library’s API can also be packaged for organizational purposes. From variability perspective, it is important for libraries, to explicitly announce their changes for client systems since some of these changes may have consequences on the functions offered by the libraries, and thus the client software behavior. These changes are usually announced through the library’s changelog and the API documentation. As an example, **Barcode4J** is a flexible generator for barcodes written in Java, its GroupID is identified by the net.sf.barcode4j domain, while it ArtifactID is labeled barcode4j-fop-ext-0.20.5-complete. So, the library is known as
**Migration Rule.** A migration is denoted by a pair of a source (retired) library and a target (replacing) library, *i.e.*, source $\rightarrow$ target. For example, easymock $\rightarrow$ mockito represent a migration rule where the library easymock\(^2\) is migrated to the new library mockito\(^3\). Table 4.1 depicts the list of migration rules that are mined and studied in this chapter.

**Method Mapping.** A migration rule is a set of method mappings between the source and the target library. The mapping between methods is the process of replacing a least one method from the source library by one or multiple methods belonging to the target library. Figure 4.2(E) shows some examples of mappings.

**Segment.** It constitutes the migration period. It is a sequence of one or multiple code changes (*e.g.*, commits), containing each, one or multiple fragments.

**Fragment.** A block of source code that witnesses at least one mapping. It is generated by contrasting the code before and after the migration to only keep the removed (resp., added) methods linked to the source (resp., target) library. For example, Figure 1.3 depicts three fragments, each fragments contains a set of added/removed methods.

**Key Phrases.** [82] It is a process of identifying the relevant words that are emphasized on, in a text. We use it to keep only important words from a function’s description when calculating the cosine similarity descriptions of two given functions. We use the Microsoft Text Analytics API\(^4\) to extract important key phrases from text.

**Cosine Similarity.** The cosine similarity is a measurement of how similar are two vectors based on the dot product of their magnitude [71]. We used the Cosine similarity to measure how close are two methods by capturing the similarity between the key phrases of their API documentations.

**Migration Refactoring.** A refactoring operation applied in a commit in which a library migration occurs.

Refactoring is defined as the process of changing software system in such way that changes improve software quality and do not alter the software behaviour [30,58]. Refactoring is one of the commonly-used techniques to improve software quality [30,73]. There are different refactoring operations that could be used to improve software quality such as a change in parameter types, move

\(^2\)http://easymock.org
\(^3\)https://site.mockito.org
\(^4\)https://goo.gl/exSkku
attributes/methods, rename variables/parameters/attributes/methods/classes, extract methods, extract classes, etc [30].

While object oriented (OO) software quality metrics are measurable from the codebase and formally defined in the literature [19], code readability is still a human judgment of how easy the code to understand, and a readable code facilitates its maintainability and comprehension [1,15,65]. We particularly focus on the following code metrics:

**Coupling.** Measure the level of relationship between modules [72]. While designing the software, low coupling is desirable (i.e., less dependency between modules). In this case, we also used one metric to compute it, i.e., Coupling Between Objects (CBO). The higher the CBO, the higher the class coupling.

**Cohesion.** Measure the level of relationship within module [72]. While designing the software, high cohesion is desirable (i.e., strong interaction between code elements in a module) since this target helps in fostering code maintainability. We used one metric to assess the cohesion of classes, i.e., the normalized Lack of Cohesion of Methods (LCOM). We have selected the normalized LCOM metric as it has been widely recognized in the literature [18,62] as being the alternative to the original LCOM, as the latter addresses its main limitations (misperception of getters and setters, etc.). The lower the LCOM, the higher the class cohesion.

**Complexity.** A developer should reduce the complexity of the software to reduce maintenance time and efforts. Five complexity and volume metrics are used to compute this quality attribute, namely, the Cyclomatic Complexity (CycC), the Line of Code (LOC), the Line with Comments (CLOC), the Ratio of Comment Lines to Code Lines, and the Number of Blank Lines. Normally, higher values of these metrics indicate a higher of class complexity [19].

**Method-level Vector space representation.** To represent method source code implementation as vector we use code2vec [3]. code2vec [3] is an approach founded by Alon to represent method source code implementation as vector of numbers. code2vec [3] trained on thousands of label data. When we feed a method source code implementation to code2vec [3], the tool predicate 384 labels, and confident percentage for every method. Figure 9.2 shows how code2vec generate topics and code vector from `getDomain()` method source code implementation. For first method, The models think the topics are (get, Domain) with confident of 99.8%, while other topics have less confident. Also the model predicate the code vector of 384 as well for method 1. For second method, The models think the topics are (get, Domain, internal) with confident of 78.4%, while other topics have less confident. Also the model predicate the code vector of 384 for method 2 as well.
1.4 Research agenda

To address the above-mentioned research questions, we performed the following research plan:

- **API Mining**: In this phase, we detected existing migration by mining thousands of open source project that have already witnessed migrations.

- **API Recommendation**: In this phase, we learned the features developers rely on when performing these migrations to be able to recommend such migrations in similar contexts.

- **System Behavior Safety**: In this phase, we want to make sure that our recommendation did not break or change the behavior of the code. Behavior preservation can be accomplished either by verifying the code before and after the migration or by testing the code blocks that were affected by the migration.

Figure 1.4 shows our study timeline and publications. In the last four years (2017-2020), we did complete our first contribution API Mining, and the second contribution API Recommendation, and most of third contribution System Behavior Safety that we detail in the coming parts of this thesis. The dissemination of this contribution was in the form of a papers [4,5,6,7,8,9] that have discussed, in general, the challenges of API migration along with several scenarios for the automation of detection and recommendation. Thus, the first part of our research has generated the necessary dataset that will enable the next research components related to improving the accuracy of the recommendation models while guaranteeing the system’s behavior.

![Figure 1.4: PhD. timeline.](image)
1.5 Ph.D. Publications

This section outlines our achieved contributions, as part of the PhD work. These contributions are described in detail, in the coming chapters.


Chapter 2

Literature Review

2.1 Abstract

This chapter discusses the literature relevant to this work which can be divided into four main categories (i) library migration, (ii) library API recommendation, (iii) Library Breaking Changes, and (iv) empirical evaluation of software quality and comprehension.

2.2 Mining software quality and comprehension

Several studies focused on understanding how developers perceive API related method changes. In the context of library updates, many studies have been proposed to capture the needed changes on the client source code applied along with API migration [43, 57, 84, 88]. Most of the existing approaches use textual similarity between the structures and method signatures as a basic technique to identify identical methods between multiple library versions. Similar approaches were tackling the problem of mapping between methods across different languages. The majority of these approaches employed information retrieval and natural language processing techniques to identify similar method usages in different languages [60, 61].

Another recent study has been conducted by Schäfer et al. [67], by analyzing changes in the method call locations to extract the fragments of added/removed methods. The authors compute associated rules from the fragments before filtering them using the similarity of method signatures. The approach allocates one method to each call. Consequently, such approach favors the one-to-one
method mapping and ignores the existence of other added (resp., removed) methods in case of
\textit{many-to-many} method mapping \textit{i.e.}, replacing one or many methods with one or many methods
within the same fragment.

Teyton et al. \cite{77} extended this to support all possible cardinalities of method mappings. They
performed the same migration process for a given input migration rule to extract all the fragments,
then they applied the Cartesian Product between the two sets of removed and added libraries.
This generates all the possible combinations of mappings that may have occurred between the
set of source library and target library methods. Then, they calculate the frequency of identical
combinations throughout all the studied projects. Finally, they define an acceptance threshold
where any combination with a higher frequency than this fixed threshold is considered a legit
method mapping.

In contrast with previous studies, this approach was similarity-agnostic since it is robust against
libraries variations in their design, naming conventions and vocabulary. On the other hand, it ex-
clusively relies on existing migrations between two given libraries to be able to provide mappings.
Lastly, its performance in terms of accuracy depends on the frequency of such migrations across
projects, as we will discuss later in Section 4.5.

Alrubaye et al. \cite{5} introduced a mining approach that extracts existing instances of library method
replacements that are manually performed by developers for a given library migration to automatic-
ically generate migration patterns in the method level. Thereafter, the proposed approach com-
bines the mined method-change patterns with method-related lexical similarity to accurately de-
tect mappings between replacing/replaced methods. Results indicate that substitution algorithm
approach significantly increases the accuracy of mining method-level mappings by an average ac-
curacy of 12\%, as well as increasing the number of discovered method mappings, in comparison
with existing state-of-the-art studies.

Similarly, Hora et al. \cite{34,36} adapted the approach of Teyton et al. \cite{77} in the context of detecting
method mappings between different releases of the same library in order to analyze the evolution
of its API. They used association rules and the frequent itemset mining technique on method call
changes between two versions of the same method. The proposed approach generated thereafter
rules to specify which old call should be replaced with a new call. The study was extended later
in \cite{35} to analyze the developers’ perception of these tracked API changes.

A dynamic analysis was also used by Gokhale et al. \cite{32} who have developed a technique to infer
likely mappings between the APIs of Java2 Mobile Edition and Android graphics. Their approach
was specific to the given libraries. Kabinna et al. [38] mined the migration of 9 logging libraries in Apache software foundation projects. Their findings show that the majority of the 49 detected migrations were successful, but the process is error-prone with an average of two post-migration bugs even when experienced developers were accomplishing the migration task.

2.3 Library API recommendation

Several recent studies proposed different API recommendation techniques based on the context of usage. Most of the API recommendation techniques are based on results returned by web search engines and crowd-sourcing, as well as the recommendation of relevant functions, was the focus of multiple studies [79, 80]. McMillan et al. [53] proposed an approach named as Portfolio, a search engine that models the developer’s behavior then looks for relevant functions based on (i) call graph similarity and (ii) querying open-source projects using natural language processing. Zhong et al. [93] proposed another approach called MAPO to select API usage patterns and then extracts common sequences that can be used to transform code snippets and make recommendations automatically. CLAN was introduced by McMillan et al. [51] and based on calculating method APIs behavioral similarity by comparing API call-graphs. Software libraries recommendation has been recently formulated as an optimization problem by Ouni et al. [59] using multi-objective search based on NSGA-II [22] to find the best trade-off between maximizing the coverage and similarity between libraries while reducing the number of recommended libraries.

Pandita et al. [61] recommend API mapping between C# and Java using the same API, different programming languages. He detects method mappings between a given source and a target library by automatically discovering possible method mappings across their APIs, using text mining on the functions textual descriptions. Their work was extended to include temporal constraints [60] and to compare text mining between various IR techniques. A dynamic analysis was also used by Gokhale et al. [32] to develop a technique to infer possible mappings between the APIs of Java2 Mobile Edition and Android graphics.

Martinez et al. [49] proposed a systematic technique to adjust software, following the object-oriented design principles in general, and whose programming language is C/C++, in particular, to mobile environments. They defined a migration process, based on the integration of interoperability tools, such as the Architecture-Driven Modernization (ADM), to transform C++ code into HAXE. One of the main benefits of defining rules using ADM, is their reusability of both transformations and its corresponding models. The authors finally provided an initial definition of C++ the metamodel
using the Ecore metamodel, through the implementation of an injector, in order to generate a C++ model.

Fazzini et al. [29] implemented \textit{AppEvolve}, to mine existing Android projects to extract examples of API updates that can be applied as patches for other projects that are in need of similar updates. Similarly, Xu and Meng [89] designed \textit{Meditor}, to extract and execute API migration edits, from history of code changes, taken from open-source projects.

2.4 Library Breaking Changes

By definition, any API change between two version of a library, which would trigger the client software to experience syntactic errors when performing library upgrade, is known as a breaking change. Breaking changes are thus a real challenge for library variability as they lock client software systems to (a) specific version(s) and prevent their automatic upgrade. Past research around breaking changes has been in the form of both, human-based and empirical studies which were primarily focused on Java-based open-source systems.

Surveys by Bogart et al. [14] indicate that the policies and principles of the development community, to which an ecosystem belongs to, has an important role to play with regards to the introduction of breaking changes. Xavier et al. [87] not only formulated the five primary reasons as to why developers to introduce breaking changes but also claimed that developers understand the repercussions caused by introducing breaking changes. A study of Android SDK related posts on StackOverflow by Linares-Vasquez et al. [46] shows that as the number of changes in an API method increase, the number of questions posed by users also increases.

An empirical study on backward compatibility of API’s by Xavier et al. [87] indicates that the percentage of breaking changes increase over time and that almost 30% of API’s are not backward compatible. Kula et al. [44] found that systems are likely to adopt latest library releases. Work by Zhou and Walker [94] have shown that use of API deprecation is sporadic and inconsistent, and emphasize the need for library developers to pay careful attention to this concept. Dig and Johnson proposed an approach to detecting code refactoring that is liable to cause breaking changes in client/dependent systems [26]. In their study of Android apps, McDonnell et al. [50] indicate that developers are reluctant to use API’s introduced in new versions of the SDK. Linares Vasquez [47], analyzed fault-and-change proneness of over 7,000 Android apps, and claim that successful apps are less fault-prone and change-prone than apps with low user ratings. An exploratory study
by Linares-Vasquez [12] resulted in a classification mechanism for the different types of changes. These studies show that developers experience in general difficulties in updating their libraries. To ensure variability, these studies offer several classifications for non-functional changes and refactorings to expose those preventing backward compatibility of libraries.

2.5 Empirical evaluation of software quality and comprehension

Recent empirical studies revisited the relationship between code changes and quality from a more developer-focused perspective. For instance, Pantiuchina et al. [62] found that there is a misperception between various popular metrics, such as coupling and cohesion, and what developers actually consider to be an improvement in their source code. Their findings show that, although developers do explicitly mention their intention in improving structural metrics, such as complexity, coupling and cohesion, the actual code changes that they perform does not necessarily improve their metrics. Similarly, Fakhoury et al. [28] have analyzed 548 commits where their developers explicitly state in their messages that they are performing readability improvements, by measuring the state-of-the-art readability metrics, on the source code, before and after committing the code changes. Similarly to [62], Fakhoury et al. found no significant correlation between the values and so the current existing readability metrics are not in line with what developers consider to be an improvement in code comprehension. Yet, their study largely inspired us to challenge the readability of code changes performed by developers during the migration process.

Our study builds on top of previous works, in the nature of its empirical setup, as we use a set of extracted commits, measure their impact on structural and comprehension metrics, and we perform statistical analysis to draw our findings. Besides targeting a different problem, our study differs from these previous studies, in the way we select our analyzed commits. Previous studies use String matching to filter out commits, while the dataset we use was constructed by finding real-world migration performed by developers and their actual mappings in the source code, regardless of whether developers do mention it explicitly in their commit messages or not. Despite these differences, we are also interested in re-challenging structural and readability metrics, on their ability to capture the side effects of the migration related code changes. Moreover, our study aims to complement existing studies by empirically investigating whether quality matters for developers, besides the correctness of migrated code. We also want to particularly raise the awareness of software engineering practitioner and researchers to the importance of considering the side effects of their proposed techniques on software quality and code comprehension. The main difference between the existing approaches and our approach, is that they tackle the prob-
lem of mapping between methods across different languages, whereas our approach recommends API mappings between different libraries belonging to the same programming language. Also our approach recommends that many-to-many method mapping.
Part I

API Mining
Chapter 3

Variability in Library Evolution

3.1 Introduction

Software systems are evolving through the composition and maintenance of various components, under one architecture, designed to serve the purpose of the client’s needs. These components are either created in-house by project developers or outsourced from third-party open-source or commercial libraries. Recently, more and more software systems rely on the use of third-party libraries by means of cutting the implementation time and effort and increase the quality of the end-product. These libraries offer services that can be (re)used as part of the core program functionalities. Thus, reusability is one of the main reasons why developers opt for libraries, and from that perspective, libraries can be seen as software artifacts that ensure variability. Variability is known as the ability of a software artifact to adjust with respect to a given context [83]. With the large number of heterogeneous environments that a software artifact can be exposed to, its variability is measured by the degree of its immunity to manual adjustments and configurations [31]. So, the more a software artifact is variability friendly, the more it is reusable in various contexts through its adaptations, these adaptations are also known as variants. From this perspective, libraries are variants that software architects must choose wisely to increase the reusability of their system and to reduce the development and to tune overhead. However, localizing the right variant, i.e., library may not be straightforward. There is a wide variety of libraries offering similar functionalities, and developers are not necessarily aware of all of them to decide on which library fits better the given project requirement(s) [92].

Moreover, the primary key challenge that faces the choice of libraries as variants is the evolution
of software systems requirements. When dealing with feature requests, developers may be forced to acquiring richer APIs, i.e., as part of maintenance practices, developers need to replace an existing library with another. This act of replacement can be categorized as either a library upgrade or migration. An upgrade occurs when the developer replaces, an outdated library version with a more recent one [20]. A migration, on the other hand, is the replacement of the library with one that was (most likely) developed by another development team, but still, performs the same (or similar) functionality as the replaced library [77]. Although the automation of library upgrades has been advancing especially with the rise of continuous integration and dynamic component frameworks [86], library migration’s automation is challenging for developers due to the high risk of introducing malfunctions into an already stable software system. Thus, from variability perspective, it is crucial for a variant, i.e., library, to be chosen based on how well-maintained it is, in a way to keep its services up-to-date, to be agnostic to deprecation, and to provide services that are compatible with environmental changes.

Our research goal, in this chapter, is to raise the awareness of libraries’ role, as an important variant, in the software lifecycle, by investigating their evolution on a wide variety of Java projects and inspecting their impact from a maintainability perspective. In this context, we analyze the relationship between libraries and software systems from two perspectives: (1) the client software using several libraries, and (2) libraries hosted on various client software systems. From the software’s perspective, developers are constantly searching for the most appropriate library that better satisfies the requirements. The search process can result in including new libraries as well as replacing existing libraries with different ones. Developers are also required to update their libraries into more reliable and stable releases constantly. Thus, developers are in need of automated tools that allow them to maintain and evolve their external libraries. From the library’s perspective, its maintainers, are also required to update the libraries functionalities to be compatible with newly appearing environments and to stay competitive with other libraries offering similar services. The update process tends to be challenging as developers must guarantee that their new releases do not introduce any breaking changes in the client systems, already deploying their old releases.

This chapter presents the results of the study which:

- 1) gives an overview of libraries evolution in a large set of software systems;
- 2) demonstrates the degree of variability among libraries researchers in the field with the main challenges and limitations that require further investigation;
- 3) enumerates the current challenges of maintaining libraries and how to support its variabil-
CHAPTER 3. VARIABILITY IN LIBRARY EVOLUTION

3.2 Motivation

Since Java libraries are widely available for reusability, software developers are provided with opportunities to include them in the core functionalities of their systems. So, measuring the existence and frequency of library usage helps in detecting upgrade and migration trends. To this end, we defined the following research questions.

RQ1. (Existence of Libraries). To what extent do developers rely on libraries in their software systems?

To answer this research question, we perform a quantitative analysis by designing a mining algorithm to detect the libraries dependencies on any given software. Mining existing dependencies allows us to uncover the spectrum of library usage regarding frequency, type of invocations and the ratio of API functions’ calls in client libraries.

RQ2. (Evolution of Libraries). How often libraries evolve within client software systems?

We monitor the evolution of libraries offered functions throughout various releases to quantify libraries upgrades and migration trends. We identify the topmost performed library upgrades and migrations along with their count. This research question complements the first research question in quantifying the developer-performed, library-related tasks. It also provides the background for the following research question that distinguishes the tasks that included changes to the client java files.

RQ3. (Variability Degree of Libraries). Does the migration process ensure better libraries from variability standpoint?

The purpose of this research question is to investigate whether the newly migrated-to libraries are considered better variants (less triggering to code changes) in comparison with ‘retired’ libraries. In order compare them, we need to formally quantify the variability of artifacts, i.e., their ability to evolve through changes without altering the client software system’s code base from syntactic and behavioral perspective. To answer this research question, we propose the Variability Degree (VD) defined as follows:
\[ VD(l_p, l_{p+1}) = 1 - \frac{1}{\text{occ}} \sum_{i=1}^{\text{occ}} CC(s_i) \]

With \[
\begin{align*}
CC(s_i) &= 1 & \text{if there exists at least one code change related to } f_{p+1} \\
CC(s_i) &= 0 & \text{if there is no code change related to } f_{p+1}
\end{align*}
\]

Where the library version \( l_p \) has been updated to \( l_{p+1} \) (\( l_p \rightarrow l_{p+1} \)) in the software S’s version \( s_i \).

\( CC(s_i) \) (Code Change) represents a function that takes as input the code base of S with the newly updated library version \( l_{p+1} \) and returns 1 if there exist at least one code change in the source files i.e., java classes related to the functions of library and 0 otherwise. It is important to note that configuration files changes are not considered as part of the code changes as we aim to measure whether the upgrade process has triggered developers to manually change their source classes to reflect the library update. Although changing configuration files may be also considered as a manual process, there are a lot of development frameworks that can automate the process of updating dependencies when new newer version of libraries are available. The automation of software configurations has been well studied in the area of variability [17].

To calculate the VD between \( l_p \) and \( l_{p+1} \), we gather all upgrades occurrences (occ), we calculate the sum of CC(), computed for each occurrence, then the sum is normalized by the total number of occurrences and then inversed. Thus, the higher the VD is, the less ‘explicit’ is the upgrade between the two versions of the library. This metric helps in measuring the effect of library evolution activities on client software systems.

As an illustrative example, Figure 3.2\textsuperscript{1} shows the upgrade of log4j from version 1.2.13 to 1.2.15. Since this upgrade is reflected in the project library configuration file without altering any class files, then their variability degree according to one occurrence is as follows: \( VD(\text{log4j.1.2.13, log4j.1.2.15}) = 1 \). As shown in Figure 3.3\textsuperscript{2}, the upgrade of jetty from 8 to 9 has triggered various changes in 5 class files, so their variability degree according to one occurrence is as follows: \( VD(\text{jetty.8.0.0, jetty.9.0.0}) = 0 \).

To answer the research questions mentioned above, we have examined the evolution of several libraries in open-source projects, the details of the retrieval and analysis are detailed in the next subsection.

\textsuperscript{1}https://github.com/plutext/docx4j/commit/d1e28e6fbb3b5ed8974572920c0899254abbd49b
\textsuperscript{2}https://github.com/apache/hive/commit/191302cf4f9eae5ef51964bdab8d8e859292aa17
3.3 Retrieval Phase

Previous studies on library evolution [77,86] have conducted their experiments on known datasets of Java and Android projects. They also analyzed a limited set of libraries within projects [78]. We have decided to extend these studies, and we have decided to consider all third-party libraries that we encounter on the projects under study. To restricted our study to projects using Maven. Maven is the Apache foundation project management tool. We have chosen Maven projects because they encapsulate their dependencies to libraries on a specific file labeled Project Object Model (POM), which facilitates the identification of libraries used by any project. Since the preliminary results have shown a very high number of candidate projects, and since we want to make sure we include projects that had enough cycles to include library evolution. Our inclusion criteria were mainly chosen to guarantee the feasibility of our study by reducing the search space among projects and by extracting more significant results by only considering well-maintained and engineered projects. This filtering process has been applied to approximately 312 137 Java projects [2], in which 53 703 projects were selected for the study. The following Figure 3.3 describes our data collection process.

As shown in Figure 3.3, We start with cloning every Java project to perform the static analysis. We scan only projects containing a POM file (step 1). Along with the cloned source code, we collect all its commits as well. For every commit, we record its information including the commitID, commitDate, developerUsername, commitText. We also keep track, for each commit, of all updated files. All this extracted data is saved on a database to facilitate its querying. After saving all projects information in the database, we chronologically scan, for each project, all its commits while comparing between the two POM files of two consecutive commits if a POM-related change is detected (step 2). Once an update on the POM file is detected, all associated libraries are shortlisted and then compared against the set of libraries in the previous POM file. It allows us to identify any added/removed library (step 3). For example, in Figure 3.4, when we compare libraries’ changes between two commits, we find that Junit was removed and testng and fest-assert-core were added.
3.4 Detection of Library Evolution

By definition, a library is being used by a project if it is (1) recognized by the project as an external resource, and (2) there is at least one function of the library being used by one of the code elements belonging to the project. In order to collect the set of libraries used by any project, we rely on firstly analyzing each project POM file to identify all its dependencies to libraries. Secondly, we need to guarantee that at least one function, belonging to the library, is invoked in the project (through method calls, reflection, etc.). For this purpose, we parse the project source code while searching for any Java file (1) importing on its header at least one of libraries API packages, and (2) at least inheriting one of its types or invoking one of its object’s functions. Once a library is identified in
the project, we scan all its commits and monitor library-related changes. These changes are then classified into the following categories that we have defined:

### 3.4.1 Library Upgrade

Just like any typical Java project, upgrading a library undergoes various possible forms of updates going from adding new classes and methods, updating existing functions, repackaging existing or newly added classes into new packages, remodularizing packages, etc. All these changes should first be reflected in the library's name, and second, it typically should not affect, without notice, how the library is being utilized at the client software side. In this context, we have distinguished two possible main upgrades:

#### Incremental Version.

The library’s upgrade is reflected by incrementing its version. For example, in Figure 3.2, `docx4j`, a library for creating and managing OpenXML packages, has upgraded `log4j` from 1.2.13 to the newly patched 1.2.15. This change is usually reflected in the POM file of `docx4j` and has no impact on the class files using `log4j`.

![Figure 3.2: Upgrading log4j from 1.2.13 to 1.2.15 in the docx4j project.](https://github.com/plutext/docx4j/commit/d1e28e6fbb3b5ed8974572920c0899254abbd49b)
Repackaging. 

Just like any regular Java project, libraries are subject to design improvement. This process includes restructuring class files into multiple packages based on various criteria including their coupling and cohesion, semantic similarity, collaboration in the functionality, etc. This process is known as software remodularization (Wiggerts 1997).

![Figure 3.3: Upgrading jetty-all library in the Apache hive project.](https://github.com/apache/hive/commit/191302cf4f9eae5ef51964bdab8d8e859292aa17)

As shown in Figure 3.3, in the Apache hive project, the jetty-all library has been remodularized. The developers of hive have removed the dependency to the library’s old ArtifactID labeled jetty-all, and they have included the newly repackaged version by including needed packages, e.g., jetty-rewrite, jetty-server, and jetty-servlet. The remodularization process is reflected in the library’s artifactID, but the GroupID usually remains intact. Unlike incremental version upgrade, developers are required to update the headers of class files referencing the old library to reference any used package; this process requires searching, in each class file, for the suitable packages that contain used the library’s classes and functions to include.

### 3.4.2 Library Migration

By definition, a library migration requires the replacement of one library by at least another one. Similarly to the upgrade process, we distinguish two types of migrations:
Instant Migration

The migration process is considered instant if the replacing library is added and replaced library is removed simultaneously. More practically, developers may perform the migration process on multiple files sequentially. This induces the existence of one or many commits containing code changes related to swapping dependencies between libraries. Once no dependency on the retired library exists, developers merely remove it from the POM file. In the following example exacted from ps2-parser project, we detected a migration from, i.e., the junit library to two replacing libraries labeled testng and fest-assert-core.

As illustrated in Figure 3.4, the project’s dependency on junit has been completely removed, and instead, the developers introduced two new libraries, testng and fest-assert-core in the POM file. It is explained by the developers’ intention of replacing the functionalities of junit by including introducing new objects and functions at the source code level, which explains the removal of all junit objects and functions from the project test files.

Figure 3.5: Replacing junit functions by fest-assert-core functions in the ps2-parser project.

---

5https://github.com/ssindelar/ps2-parser/commit/4e76b35f32011159db321e8a1540d03e004d25e8
For example, Figure 3.5\(^6\) shows how developers updated the class file `CharacterDaoTest.java` by removing the imported `junit` packages and replacing them with both `testng` and `fest-assert-core` imports. The `junit` invoked functions `Assert.assertNotNull()` and `Assert.assertEquals()` have been replaced respectively by `assertThat().isNotNull()` and `assertThat().isEqualTo()`, functions belonging to `fest-assert-core`. This example is interesting. It demonstrates that one or many functions can replace one or many functions. Such n-to-m possible migrations stand against the ease of automation of the migration process as it requires the developer’s deep understanding of both source and target APIs. More precisely, the developer is required to correctly map each function from the replaced API to a possible candidate function(s), if they exist, in the replacing API. Coming back to the illustrative example in Figure 3.5, the `CurrencyDaoTest.java` class file reflects the migration changes. In this file, `junit`’s `Assert` object has been replaced by `testng`’s `AssertJUnit` object.

As shown in Figure 3.6\(^7\), the migration process was more straightforward compared to the previous example shown in Figure 3.5; `testng` has provided a package that contains objects and functions that can immediately replace any `junit` objects and functions. It is explained by the `testng` developers’ intention of encouraging developers to migrate to their library by providing them with a richer API package that contains means of easily replacing famous and popular competitor libraries like `junit`.

### Delayed Migration

The migration is considered delayed if, in contrast with the instant migration, the retired library remains in the project POM file even if it no longer contributes to the project.

In the example shown in Figure 3.7, the developers included the new library `json` whose the exception handling object `JSONException` has replaced `JsonFormatException`, the default exception handler

\(^6\)https://github.com/ssindelar/ps2-parser/commit/4e76b35f32011159db321e8a1540d03e004d25e8

\(^7\)https://github.com/ssindelar/ps2-parser/commit/4e76b35f32011159db321e8a1540d03e004d25e8
CHAPTER 3. VARIABILITY IN LIBRARY EVOLUTION

3.4.3 Detection of Candidate Library Migration

Unlike the detection of upgrades, the detection of migration process cannot be straightforward. As shown in Figure 3.7, one or multiple libraries can be replaced by one or multiple libraries. It sophisticated the identification of the exact library replacement(s) especially that multiple migrations may also occur at the same time (same set of commits) or in a delayed fashion. For this purpose, we rely on the heuristic of Teyton et al. [78], as shown in Figure 3.8, for the approximation of possible migrations.

Figure 3.8 gives the steps used to apply to heuristic to candidate migrations between libraries. A reminder that a candidate migration (1) should contain evidence of added and possibly removed libraries at the POM file, (2) and it should trace at least one function from the removed library

previously used. Even if the retired library was removed from the files headers, the developers did not remove it from the project’s POM file. The detection of such situation tends to be very tricky because it can be similar to simply add a new library with new functionalities. We spot such situation only if the newly included library replaces at least one other library at the function level.
Figure 3.8: Overview of the library migration detection heuristic.

that has been replaced by another function from the added library. Any pair of libraries satisfying these two conditions automatically qualifies for a potential match for migration.

Since multiples libraries can be added and removed simultaneously, we apply the following heuristic: *Every added library can be a candidate of a replacement for any removed library as long as there is at least one swap at the function level.* This allows us to detect migrations pairs as follows: We apply the Cartesian Product (CP) between the set of added libraries and the set of removed libraries to generate every possible migration pair. It creates a potential link starting from a deleted library to all added libraries. Then, to verify which of these links is valid, we search the entire database for similar identified links. It allows us to calculate the number of occurrence of each link in similar migration contexts. Then we use this frequency as a voting system to champion one link over the remaining links and thus generate a final migration pair. Note that, in case of a tie between two links, they are both considered as valid links, and so the migration pair will contain one removed
library and two or more added libraries. This voting system utilizes the knowledge extracted from similar migration scenarios (the same candidate added and removed libraries) to help in distinguishing pairs. Therefore, using a broader set of projects is strongly recommended to increase the accuracy of the voting system.

To illustrate the voting process, we consider the following sets: removedLibraries = [org.json, jmock, testing] and addedLibraries = [mockito, gson, junit]. We build an indirect graph between the two sets using the CP where the starting nodes are the removed libraries and the ending nodes are the added libraries. Figure 3.9, illustrates the resulting graph.

The weight of every edge is calculated based on the frequency of occurrence of such link in the entire set. In the example above, the occurrence of org.json moving to gson is equal to 12. After determining the weights, we normalize them by dividing each weight by the maximum weight initiated from one source library to all target libraries. For example, the maximum weight of an edge leaving from org.json is 12. Thus, the normalized values between of edges linking org.json to Mockito, gson, and junit are respectively $\frac{5}{12} = 0.4$, $\frac{12}{12} = 1$, and $\frac{12}{12} = 1$. After applying this process, we filter out all links not equal to 1. In case of a tie between two or more links, we consider all of them as a migration set. For example, the detection of the migration of junit to both testng and fest-assert-core in Figure 3.4 shows the tie between both edges of the graph containing junit as a starting node and testng and fest-assert-core as part of candidate ending nodes.
CHAPTER 3. VARIABILITY IN LIBRARY EVOLUTION

3.5 Experimental Results

3.5.1 Studied Projects

This section presents the set of quantitative and comparative experiments related to answering the existence and evolution of Maven libraries in Java software systems. Our experiments were conducted on a set of libraries detected in the selected Java projects from Dataset [2]. We did not focus on a limited set of popular libraries as it may give a biased view since these libraries are well-maintained by their developers. The data collection scripts along with the dataset that was part of our study are available for replication and extension purposes\(^8\).

3.5.2 RQ1. Existence of Libraries

We highlight the existence of libraries in the studied Java projects by exploring their usage throughout the history of the software maintenance. Once a library is detected on a given project, it was difficult to distinguish whether it is newly introduced or an existing library. Another challenge consists of distinguishing external libraries and internal software functions that are invoked as a library. On these situations, we applied a heuristic that clusters libraries firstly by their GroupID, then by their ArtifactID, we ignore all subgroups and version information, and only kept the couple of GroupID and ArtifactID as the main identifier of the library. The limitation of this heuristic is its incapability to distinguish two versions of the same library in case they are identified by different GroupID and ArtifactID. However, this case is rare because sub-domain changes are hardly occurring unless the library’s ownership changes. Table 3.1 first gives an overview of the highlights of our findings in numbers.

Table 3.1 shows that the selected projects in this study are heavily dependent on third-party libraries. Also, the high number of detected upgrades and migrations show an extensive history of evolution. It was not surprising to observe that the number of upgrades, in general, is higher than the number of migrations, since developers tend to update their libraries more often and the incremental version upgrade can sometimes be automated in contrast with repackaging upgrades and migrations that tend to be manual. Roughly we found that, on average, the ratio commits of involving libraries is around 16% of the overall commits in all projects combined. Regardless of whether this ratio is considered low or high, it represents evidence of library-related maintenance activities that software engineers are responsible for. So, it is critical to account for such variant as

\(^8\)http://sevis2017-replication.alruaybe.net/
Table 3.1: Projects Overview

<table>
<thead>
<tr>
<th>Item</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloned Repositories</td>
<td>57,516</td>
</tr>
<tr>
<td>Scanned POM files</td>
<td>53,703</td>
</tr>
<tr>
<td>Libraries with unique ArtifactID</td>
<td>74,575</td>
</tr>
<tr>
<td>Libraries with unique control version</td>
<td>280,631</td>
</tr>
<tr>
<td>Commits Retrieved</td>
<td>9,583,718</td>
</tr>
<tr>
<td>Incremental version upgrades</td>
<td>8,879</td>
</tr>
<tr>
<td>repackaging upgrades</td>
<td>298</td>
</tr>
<tr>
<td>Unique upgrades combined</td>
<td>9,177</td>
</tr>
<tr>
<td>Unique migrations combined</td>
<td>2,249</td>
</tr>
<tr>
<td>All upgrade instances</td>
<td>345,541</td>
</tr>
<tr>
<td>All migration instances</td>
<td>28,509</td>
</tr>
</tbody>
</table>

it affects both software’s functionalities from the user’s perspective and it requires to be maintained from the developer’s perspective.

Figure 3.10 shows the top libraries mostly involved in all the commits involving library changes. These commits can be linked to any library-related activity such as adding a new one to including new library functions in the source code etc.

We did expect testing related libraries such as *junit* or *testng* to be the highest in this ranking since they are heavily involved in various testing practices. Therefore, developers tend to pay more attention to keeping their APIs up to date and quickly move to any testing library that shows promise in helping them with building better test suites. Instead, *springframework* was found with the highest number of commits in client software systems. This library is demanded in J2EE based projects, offering a wide variety of Spring patterns that are useful for several architectures and frameworks.

Figure 3.11 enumerates the libraries with the highest update frequency. An actively updated library may be interpreted as a sign of a healthy service that is up to date from compatibility perspective and lesser prone to bugs. On the other hand, with every update, besides the overhead of performing the physical update and any regression testing required on the source code afterward, there is always the chance of introducing behavioral or syntactic breaking changes. Developers need better strategies to quantify the pros and cons of evolving a library to better analyze the tradeoff and make a better decision; this explains why developers are sometimes reluctant to library changes.
3.5.3 RQ2. Evolution of Libraries

Table 3.2 contains the 6 most frequent remodularization instances detected, e.g., *pax-url-mvn* has been split into two packages namely *pax-logging-service*, *pax-logging-api*. This transformation has been spotted in 72 POM files. The remodularization process may also include merging existing packages, e.g., we detected that *tomcat-tribes*, *tomcat-el-api*, and *tomcat-api* had been replaced by *tomcat-embed-core*.

Table 3.3 has the cumulative count of all detected migration instances between two given libraries. We have conducted a manual analysis of the top 10 detected migrations; we observed that the highest detected migration was a false positive. In fact, *jackson-mapper-asl* has been renamed to *jackson-databind*. Its detection as a migration was due to the simultaneous change of their GroupID from *org.codehaus.jackson* to *com.fasterxml.jackson.core*. In a similar context, it is difficult to detect such false positive without an extensive manual validation, that is why previous studies have conducted qualitative analysis of all their findings to filter out any inconsistent migration [78]. We also compared the results of our migration with Teyton et al. [78]. The purpose of this comparison is to verify whether the results of Teyton et al. were specific to the set of projects under study or can
### Table 3.2: Topmost remodularized libraries

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Remodularized Library Name</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>pax-url-mvn,</td>
<td>pax-logging-service, pax-logging-api</td>
<td>72</td>
</tr>
<tr>
<td>websocket-server, fcgi-server</td>
<td>jetty-home</td>
<td>49</td>
</tr>
<tr>
<td>cditest</td>
<td>openwebbeans-impl, openwebbeans-jsf, openwebbeans-spi</td>
<td>30</td>
</tr>
<tr>
<td>tomcat-tribes, tomcat-el-api, tomcat-api</td>
<td>tomcat-embed-core</td>
<td>25</td>
</tr>
<tr>
<td>gshell-network, gshell-file, gshell-pref, gshell-artifact,</td>
<td>gshell-core</td>
<td>12</td>
</tr>
<tr>
<td>openwebbeans-openejb</td>
<td>test:cditest, cditest-owb</td>
<td>11</td>
</tr>
</tbody>
</table>

### Table 3.3: Topmost performed migrations

<table>
<thead>
<tr>
<th>Source Library</th>
<th>Target Library</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>jackson-mapper-asl</td>
<td>jackson-databind</td>
<td>344</td>
</tr>
<tr>
<td>log4j</td>
<td>slf4j</td>
<td>202</td>
</tr>
<tr>
<td>junit</td>
<td>testng</td>
<td>167</td>
</tr>
<tr>
<td>commons-logging</td>
<td>slf4j-api</td>
<td>144</td>
</tr>
<tr>
<td>commons-httpclient</td>
<td>httpclient</td>
<td>132</td>
</tr>
<tr>
<td>easymock</td>
<td>mockito</td>
<td>115</td>
</tr>
<tr>
<td>xqj-api</td>
<td>tagsoup</td>
<td>109</td>
</tr>
<tr>
<td>org.json</td>
<td>gson</td>
<td>102</td>
</tr>
<tr>
<td>jersey-server</td>
<td>pax-logging-api, pax-logging-service</td>
<td>101</td>
</tr>
<tr>
<td>Contextmenu, refresher, confirmdialog</td>
<td>rxtx</td>
<td>98</td>
</tr>
</tbody>
</table>
be generalized across a larger set which proves the existence of migration trends. The comparison results are shown in Figure 3.12.

Based on Figure 3.12, the axis we identified the same migration rules expect for (org.json moving to gson). This can be explained by the fact that we are using a larger set of both libraries and projects. However, both studies have identified log4j to slf4j as the most popular migration pair while they disagreed on the second most popular pair. Junit to testng was ranked second in our study while it was third in Teyton et al.’s ranking, where commons-logging to slf4j-api was second but in our study, it became third. The regency of our study gives a more recent snapshot of the all occurring migrations in general, but it does not still reflect the upcoming migration of trends. Since the migration process is symmetric, then it is hard to detect which libraries are gaining popularity and which libraries are losing developers’ interest without taking into account the time factor. In general, we were able to, through this comparison, to show that migration trends exist between a set of libraries whose are similar from a functional perspective. From variability perspective, it is crucial to choose libraries that are being migrated to, rather than using a library that tends to be migrated away from. Either we have not investigated the motivation behind these migrations since
these migrations are recorded in several independent projects, it shows empirical evidence that there is a general agreement among developers, who made these migrations, on their decisions.

3.5.4 RQ3. Variability Degree of Libraries

The goal of measuring the variability degree is to show that library evolution, in general, involves various changes to the code using the functions offered by their APIs. In this context, while existing studies try to predict the type of changes accompanied with library evolution, our study aims in using the large set of detected upgrades/migrations to quantify the degree of these changes based on real-world scenarios. Such information can be valuable for developers who are seeking to upgrade libraries while reducing the change effort. Answering the third research question investigates whether the recorded migrations improve, in general, the variability of libraries. Therefore, we selected the topmost performed migrations, and for each project that belongs to any migration rule, we calculate its VD. Then we create two sets of VD values that we plot, in Figure 3.13, to verify
if the values of the target libraries are better than the source libraries.

We notice in Figure 3.13 that the target libraries have a relatively higher VD value compared to source libraries. This applies to the majority of migrations such as log4j to slf4j and junit to testing. We noticed for two migration rules that the VD values, for both source and target libraries, were similar. This fact is explained by the fact that the source and target library are historically identical, they are being seen as different libraries since they are identified by a different GroupID and ArtifactID. Another outlying rule was the swapping of jersey-server by pax-logging-api, pax-logging-service. We applied the Mann-Whitney U test with p-value = 0.05, for the migrations rules we identified, and we could not find the difference between the values of two sets to be statistically significant. Thus, we plan to qualitatively analyze the detected migrations and filter out the false positives since they negatively impact the statistical analysis by introducing identical values in both
groups under comparison.

### 3.5.5 Discussion and future directions

Our quantitative study has proven the existence of third-party libraries as an essential factor in the lifecycle of the project. We have shown the frequency of each library may evolve, some of the evolution factors are not necessarily controlled by the client project developers. To reduce the overhead of maintaining libraries, it is important to account for their change. Thus, there is a need for identifying metrics that help in ranking libraries not only from the functional and non-functional perspective but also based on their impact on client code changes. From that perspective, we developed a metric to measure the variability degree of previously evolved projects. Yet, the migration process is responsible for around 20% of the libraries evolution, and it is challenging to automate. Our initial manual classification for the commits have revealed that feature enhancement stands as the primary motivation for developers to switch to a new library. Such decisions are hard to predict and so making the right library choice early in the software design remains minimal and the necessity to account for library migration remains valid. Therefore, we plan to automate the classification of evolved libraries’ commits to reveal the reasons behind these changes. Understanding the developer’s perspective, when evolving libraries, helps in automating it.

### 3.6 Threats to Validity

In this section, we report the threats to the validity of our methodology:

**Internal Validity.** The tools we build to collect commits and detect candidate upgrades and migrations may not be accurate and may exhibit false positives (wrong upgrades, incorrect migrations, detecting a migration as an upgrade) and false negatives (miss some upgrades, migrations). To mitigate this problem, we manually selected random commits from both upgrades and migrations and we validated their correctness. We also confirmed that we were able to reproduce the same migrations found by previous existing studies ([77]). We only manually checked the most frequent migrations detected, and so, we will continue reviewing the remaining migrations and removing any identified false positives. Another way to mitigate this threat is to check if upgrades/migrations are explicitly-admitted by developers manually, but this approach, while guarantees correctness, cannot be scaled to the whole set of collected commits.
Another key limitation of this work is our assumption, when measuring the degree of variability, that developers make an effort, to accommodate new functions, only through code changes. Moreover, so, if there is any other type of effort that is not reflected in the source code, such as updating the software documentation, etc. it will not be considered by our metric. At the same time, our metric captures any library upgrade that prevents the client system from being syntactic-error free. Thus, any changes external to the source code may not necessarily introduce syntactic errors unless they are linked to the project configuration files. Furthermore, as part of our future work, we plan on conducting a qualitative analysis to understand better their perception of the changes they performed while upgrading their project libraries.

Another threat is related to detecting the functions of the libraries in the client code. We use a combination of syntactic and lexical analysis to identify functions belonging to the library’s API. We assume that all functions are being called through libraries APIs. If a function is being copy/pasted in the client code, we will not be able to consider it as part of the library.

We rely on Maven for detecting the evolution of libraries in general. This represents a limitation to our work, and we plan on analyzing other possible ways to automatically identify libraries in existing projects. In our study, we have not considered the intention of developers when evolving libraries, so it is hard to justify their choices regarding choosing an outdated library or replacing a known library with an unknown one (in terms of popularity). In our future investigation, we want to focus on developer’s perception of changes. This will help us in analyzing how developers perceive variability in their choices by better understanding their motives when migrating between libraries.

Construct Validity. We report in this sections the factors that may impact the measurements and metrics in reflecting real-world scenarios. When we have manually analyzed GitHub projects, we have noticed that few developers incorporate more than one project in the same repository, and with only one POM file that belongs to only one of them. This may interfere with our segment detection algorithm since it may be searching for a migration that does not necessarily exist. Still, this does not affect our results, but we did discard these repositories from our analysis. Another principal threat to our work is the human validation of our results, mainly to verify the correctness of the found migrations in general and more particularly the function mappings. We plan, in the future, to implement many state of the art, mining algorithms and perform a comparative analysis of their findings regarding function-level changes between libraries.

External Validity. In this study, we considered a broad set of projects that we believe it does represent a good spectrum of open-source Java projects. Although it may not represent commercial
and closed-box projects, we expect our study to provide similar results if these projects do use the libraries we studied. To help in reproducing and extending this study, we carefully selected the set of Java projects to analyze. They were all available on GitHub; we also analyzed public libraries. We also publicly provide dataset we collected in the form of a database that is easily queried to reproduce the results. Our approach can also be applied to closed-box and commercial tools as long as they maintain their libraries dependencies using Maven.

3.7 Conclusion

In this chapter, we conducted an exploratory study of the existence and evolution of libraries as variants in a broad set of open-source Java projects. We identified the automation challenges linked to library upgrade and migration. Library upgrade is mainly triggered by library maintainers while library migration is a design choice performed by client software developers. To estimate the frequency of these changes, we have developed several collection tools, and we examined 53703 projects to extract all their library-related changes. We classified these changes based on how they were performed in the project source files. We extended existing studies to detect candidate library migrations. We reported all our findings, and we compared some of them with results from previous studies.

Our main findings indicate the vast existence of library changes in software systems; these changes tend to be manual, requiring several interventions in multiple source files. We want to contribute to all this ongoing research by providing this dataset of all detected library changes and their location in the project files. We plan to continue the refinement of our findings to build a coherent dataset that allows automation solutions to learn from it. Moreover, we plan on expanding our dataset with including more projects. We also plan (1) on using information retrieval techniques to automatically classify commits, relevant to migration sets, in order to understand developers’ perception and motivation, (2) analyzing the impact of these migrations on class files’ proneness to bugs, and (3) comparing between the state of art existing recommendation approaches in terms of their ability to replicate existing decisions of manually performed migrations. Since our current study indicates the substantial existence of library migrations, we can quickly identify the actual developers who performed it. As part of our future directions, we will also survey Java developers to gauge their opinion about how they account for variability when choosing libraries for their projects.
Chapter 4

Automate the Detection of Third-Party Library Migration

4.1 Abstract

The migration process between different third-party libraries is hard, complex and error-prone. Typically, during a library migration, developers need to find methods in the new library that are most adequate in replacing the old methods of the retired library. This process is subjective and time-consuming as developers need to fully understand the documentation of both libraries’ Application Programming Interfaces, and find the right matching between their methods, if it exists. In this context, several studies rely on mining existing library migrations to provide developers with by-example approaches for similar scenarios. In this chapter, we introduce a novel mining approach that extracts existing instances of library method replacements that are manually performed by developers for a given library migration to automatically generate migration patterns in the method level. Thereafter, our approach combines the mined method-change patterns with method-related lexical similarity to accurately detect mappings between replacing/replaced methods. We conduct a large scale empirical study to evaluate our approach on a benchmark of 57,447 open-source Java projects leading to 9 popular library migrations. Our qualitative results indicate that our approach significantly increases the accuracy of mining method-level mappings by an average accuracy of 12%, as well as increasing the number of discovered method mappings, in comparison with existing state-of-the-art studies. Finally, we provide the community with an open source mining tool along with a dataset of all mined migrations at the method level.
CHAPTER 4. AUTOMATE THE DETECTION OF THIRD-PARTY LIBRARY MIGRATION

4.2 Introduction

Modern software systems rely heavily on third-party library functionality as a mean to save time, reduce implementation costs, and increase their software quality when offering rich, robust and up-to-date features [20,45,55]. However, as software systems evolve frequently, the need for better services and more secure, reliable and quality functionalities causes developers to often replace their old libraries with more recent ones. This process of replacing a library with a different one, while preserving the same functionality, is known as library migration [76].

The migration between two given libraries consists of a sequence of steps: It starts with retiring the current library by removing all its dependencies from the program, which includes imports and method calls. Developers are then required to find the right replacing method(s) for each removed method belonging to the retired library. Developers are also required to verify whether the newly adopted methods are delivering the same expected functionalities of the retired library’s methods. These steps tend to be subjective, time-consuming, and error-prone, as developers need to fully understand both retired and new libraries methods, and be aware of their implementation details. This includes the exploration of their Application Programming Interfaces (API) documentation and the online search for code snippet examples of their methods usage. Moreover, the matching process between the replaced and replacing methods, belonging respectively to the retired and new library, is not straightforward. Even if libraries offer similar services, they may be different in their methods design and documentation. Thus, it would be beneficial to learn from the collective experience based on the manually performed library migrations in the past.

However, the detection of such migrations is challenging. First, there is no systematic way to detect the developer’s intention of adopting a library migration. Therefore, its detection may require extensive analysis of the history of code changes while searching for specific replacement patterns between APIs. Furthermore, deciphering the pairing of removed and added methods is complex especially when many of them are co-located in the same code block. In addition, there is no strict rule about the cardinality of the pairs, i.e., one or many methods from the replaced API can be replaced by one or many methods from the new API, which makes its automated detection more challenging.

Several studies have tackled the problem of identifying the pairs of removed and added methods, also known as mappings, using Information Retrieval (IR) techniques to detect method change patterns, method signature similarity, and method graph mining [67,76,77]. These approaches have provided efficient results when finding 1-to-1 mappings between methods. However, they
are mainly challenged when identifying mappings with larger cardinality, *i.e.*, when one or many methods can be replaced with one or many methods, also known as *one-to-many* or *many-to-many* method mappings. Also, when two or more source methods, located in the same code block, are being replaced by two or more target methods, this creates another challenge to distinguish between these interleaved mappings.

This chapter builds on the existing studies by leveraging the lexical similarity with the repetitiveness of code changes in software systems in general, and in migrations in particular [56]. We mine the repetitive patterns of method replacements, *i.e.*, mappings in the code. Intuitively, the more a mapping between methods is detected across several code fragments, the more relevant this mapping becomes, *i.e.*, a pattern.

Furthermore, to cope with interleaved mappings *i.e.* mappings occurring in the same code blocks, we identify potential mappings between methods, based on how similar their signatures and API documentation.

We implement our approach in an open-source tool that identifies all method-level migration traces between given libraries throughout a set of representative projects. For a given library, our approach works as follows: (1) it first mines all projects to identify all migration segments, *i.e.*, set of code changes (e.g., commits), in which developers performed code changes related to the given libraries APIs; (2) it extracts all migration fragments *i.e.*, code diffs containing set of removed and added methods; (3) it generates all mappings between all removed and added methods, *i.e.*, each removed method belonging to the retired library will be to one or multiple added methods belonging to the new library.

As an attempt to evaluate our approach, we conduct a large scale empirical study on a benchmark of 57,447 open-source Java projects mined from GitHub. Results show that our approach outperforms three state-of-the-art approaches by achieving an average accuracy improvement of 12%, as well as increasing the number of discovered mappings by 17%. Furthermore, the quantitative analysis of our results indicates that our approach requires less number of code fragments to accurately extract all mappings.

The chapter has the following main contributions:

- We introduce a novel approach that increases the accuracy of detecting migration fragments during the library migration process.
- We conduct a large-scale empirical study on 57,447 open-source Java projects while mining
9 popular library migrations. We also conduct a comparative study between our approach with three state-of-art approaches that we adapt for the library migration problem.

- We provide an open-source tool along with the generated migration results as a dataset for the research community to better comprehend how developers practice library migrations.\(^1\)

In this chapter, we perform a comparative study between our approach, Teyton et al. [77], and Schäfer et al. [67]. We also adapted the approach of Nguyen et al. [57] in the context of detecting mappings by pairing methods that have a strong similarity in their signatures. In the next section, we provide the challenges of detecting mappings at the method level, we also show how our approach generates the existing method mappings through an illustrative example.


**Figure 4.1: Approach Overview.**

### 4.3 Substitution Algorithm

In this section, we introduce our approach for generating method mappings for library migration. Figure 4.1 provides an overview of our approach which consists of four main phases: (1) collection phase, (2) segment and fragment detection phase, (3) mapping generation phase, and (4) valida-
tion phase. In the following, we explain each of these phases. For the fragments collection and detection, we reuse dataset of our previous study [5].

4.3.1 Collection Phase

The collection phase takes as input a list of open-source Java software systems projects. It starts by cloning and checking out all commits for each project. For every commit, we collect its properties, such as commit ID, commit date, developer name, and commit’s description. We also keep track of all changes in the project library configuration file, known as Project Object Model (pom.xml). All mined projects data is recorded in a database for faster querying when conducting the identification of segments and fragments.

4.3.2 Detection Phase

The detection phase consists of the identification of (1) segments and (2) fragments.

Segment Detection: The purpose of the segments detection, i.e., migration periods, is to locate, for each migration rule its time periods in all projects. As defined in the background section, a segment could be composed of one or many fragment-related commits involved in the migration process. As shown in Figure 4.1, the segment detection phase starts with checking whether both libraries exist in the list of added/removed project libraries. Using static code analysis, this phase locates the end of the segment by scanning all commits in which all project source files are no longer dependent on the retired library.

Indeed, we perform a straightforward static code analysis because a migration does not require the physical removal of the library to be retired from the project, as the retired library may still loaded in project through its pom.xml file; however none of the library’s methods are used in the project’s source. Once the segment end is located, we keep scanning previous commits in a backward fashion, looking for the beginning commit which contains the beginning of the fragment, i.e., the first code change related to the replacement of any retired library method. After locating all segments for a given migration, it is important to keep track of source and target libraries versions for each segment to avoid backward incompatibility in case of an API change between two versions of the same library.

Fragment Detection: The fragment detection is responsible for the source code fragments related
to the library migration changes as shown in Figure 4.1. It clones the project source files that are changed in the commits belonging to the identified time segments. We apply the Git’s Unified Diff Utility command between the changed files to generate fragments, if any. A fragment is a continuous set of lines that have been changed along with contextual unchanged lines. Only fragments containing removed (resp., added) methods from the source (resp., target) library are considered valid. We retained a total of 8,938 fragments that we index in our dataset.

4.3.3 Mapping Phase

In this step, we generate method mappings from the identified fragments using our approach, which we label Substitution Algorithm (SA).

SA starts by sorting the identified code fragments using Heap Sort (cf. Algorithm 2). The sorting process is based on two attributes. First, the number of methods per code fragment which refers to the number of added and removed methods in a fragment. Second, the frequency of a fragment, i.e., how many times a fragment appears across all projects.

If two code fragments have the same number of methods, then the fragment that has a higher frequency is moved before the fragment with lower frequency. The process moves the fragments that have less number of methods to the beginning of the list. Thereafter, SA iterates through all the identified fragments, starting from those with higher frequency, and searches for intersections $ISet \leftarrow fragment_1 \cap fragments_2$ between each fragment and the remaining set of fragments. Two fragments are considered to bear a not-null intersection $ISet$, if they share at least one common added and removed method. When an intersection exists, we remove shared methods $ISet$ from $fragment_1$ using $fragment_1 \leftarrow update(fragment_1 - ISet)$, and $fragment_2$ using $fragment_2 \leftarrow update(fragment_2 - ISet)$, and we add $ISet$ as new fragment $fragments \leftarrow add(ISet)$. Then, the algorithm iterates back to the sorting process. This process continues until there are no more intersections that can be found between all fragments in the list.

The intuition behind this process resides on using fragments with one-to-one and one-to-many mappings, which are more frequent to be seen in the identified fragments, to reduce the cardinality of many-to-many fragments, by splitting based on any common one-to-one or one-to-many mappings. Sorting the fragments prior to applying the intersection gives the opportunity to split larger fragments using smaller, yet relevant, fragments instead of performing random intersections between fragments. Our approach is based on Heap Sort in this phase.
Algorithm 1 Substitution Algorithm (SA)

INPUT: fragments - List of fragments, every fragment has list of added methods, list of removed methods.
OUTPUT: List of method mapping.

1: procedure Substitution(fragments)
2: loop
3: fragments ← HeapSort(fragments)
4: for all fragment1 ∈ fragments do
5: for all fragment2 ∈ fragments do
6: ISet ← fragment1 ∩ fragment2
7: if ISet then
8: fragments ← update(fragment1 − ISet)
9: fragments ← update(fragment2 − ISet)
10: goto loop.
11: end if
12: end for
13: end for
14: newFragment ← LD(fragments)
15: if newFragment then
16: fragments ← newFragment
17: goto loop
18: end if
19: return fragments
20: end procedure

INPUT: fragments - List of fragments have N-M method mapping.
OUTPUT: newFragment - Fragment has one added, and one removed method that have highest similarity score between method’s description.

22: procedure LD(fragments)
23: maxScore ← Average of similarity score
24: for all fragment ∈ fragments do
25: for all rmFun ∈ fragment do
26: for all addFun ∈ fragment do
27: score ← CSLD(rmFunDes, addFunDes)
28: if score >= maxScore then
29: newFragment ← rmFun, addFun
30: maxScore ← score
31: end if
32: end for
33: end for
34: end for
35: return newFragment.
36: end procedure

Once all intersections are completed, LD(fragments) iterates through fragments with many-to-many mappings, with the aim of splitting them further using the lexical similarity. For each fragment, our approach calculates the similarity score $CSLD(rmFun_{Des}, addFun_{Des})$, between the description of the removed methods $rmFun_{Des}$, and the description of added methods $addFun_{Des}$ by following two steps:

First, it extracts the key phrases \[^{[82]}\] for $rmFun_{Des}$, and $addFun_{Des}$ to keep only relevant words. In our study we used the Microsoft Text Analytic API \[^{2}\] to extract important key phrases from the text.

Second, calculate Term Frequency–Inverse Document Frequency (TF-IDF) for key phrases of $rmFun_{Des}$, and $addFun_{Des}$ that generate vector of numeric numbers for $rmFun_{Des}$, and $addFun_{Des}$, then apply Cosine Similarity between two vectors that is a measurement of how similar are two vectors based

\[^{2}\]https://goo.gl/exSkku
on the dot product of their magnitude [71]. If the similarity score is greater than or equal to the average similarity of the top detected one-to-one correct mappings, then SA creates a new fragment *newFragment* that contains these two methods and restarts the intersection process.

SA terminates the search and returns a list of fragments, when there are no more intersections to be found, and no more newly created mappings based on methods similarity. Each fragment contains a unique method mapping.

As an illustrative example of SA’s workflow, we use the following migration rule *json* → *gson*, and four fragments 1, 2, 3 and 4, each fragment has a frequency of one as they have been extracted once during detection phase as it shows in Figure 4.2.

In iteration (A), the four fragments are sorted in ascending order using *HeapSort* based on the number of methods per-fragment and the frequency of fragment appearance.

Thereafter, during the intersection process, SA identifies possible intersection(s) between fragments 1, and 4, since both fragments share the same methods *get(int) → getAsLong()* . This intersection increases generates two fragments : fragment 1, which frequency increases by one and a new fragment 5 that are inserted in the current list of fragments, fragment 4 is now discarded.

In iteration (B), the current fragments are sorted again. During the search for intersections, another fragment is found between fragment 5, and fragment 2, as both contain the methods *toJSON-String() → toString()* . This intersection generates new fragments 6, 7, and 8, while fragment 5 and 2 are discarded and the other fragments remain unchanged.

Figure 4.2: SA illustrative example using 4 fragments of migration between json and gson as input.
In iteration (C), fragments are sorted again. We observe that there are no new intersections found between the current fragments, therefore SA iterates through all many-to-many fragments with the aim of finding lower cardinality, interleaved mappings that can be extracted using lexical similarity between added and removed methods. In this case, only fragment 3 is a candidate. Library Documentation (LD) takes as input the methods’ description from the API documentation to generate the key phrases [82]. Thereafter, it calculates the cosine similarity between the key phrases of all possible one-to-one combinations of the added and removed methods. LD returns the mapping with the highest similarity between the removed/added methods.

We observe from the example that the methods $\text{getString(String)} \rightarrow \text{getAsString()}$ have the highest similarity score, i.e., 53.13%, which is close to the average cosine similarity of the already identified fragments 1, 6, 7, and 8. Thus, these two methods are considered as a new fragment that is added to the list.

In iteration (D), fragments are sorted again. Then, during the intersection process, SA identifies an intersection between fragments 9, and 3, since both fragments have $\text{getString(String)} \rightarrow \text{getAsString()}$. This intersection increases the frequency of fragment 9 while creating a new fragment 10. while the other fragments remain unchanged.

In iteration (E), fragments are sorted again. Then, during the intersection process, SA cannot find any new intersection between the current fragments. Therefore, SA iterates through all many-to-many fragments. Since there are no many-to-many fragments, SA outputs each fragment as a final method mapping. In this example, the output of SA is a set of 6 mappings.

### 4.3.4 Validation Phase

Most of method mapping that generated by SA already verified as valid or not valid method mapping by study [5]. For rest of method mapping that SA generated, we conducted a manual inspection process similarly to our previous study [5] by building a publicly available web portal\(^3\) for the software engineering community that shows the list of library migration related-project commits for each method mapping. The authors then decide the correctness of the rule by manually checking the different method mappings in the list of commits, which constitute the ground truth. For example, from the project Selenium Grid Extras v1.1.9\(^4\), we observe a valid mapping between


put(key, description) and addProperty(key, description)\(^5\).

## 4.4 Experimental Design

We design our experimental study to mainly assess the accuracy of substitution algorithm (SA) to detect method mapping in compare with other approaches. We applied SA on same fragments that used by previous study [5] that compared three state-of-the-art approaches, namely Teyton et al. (FC) [77], Nguyen et al. (FS) [57], and Schäfer et al. (MC) [67]. We design our methodology to answer the two following research questions.

- **RQ1. (Accuracy)** To what extent SA is able to detect developer-performed method mappings?
- **RQ2. (Effectiveness)** How effective SA in detecting all the mappings with fewer fragments in comparison with existing approaches?

To answer RQ1, we evaluate the accuracy of our SA approach in detecting correct method mappings. We compare the SA mapping results with a ground truth set of the manually verified mappings, mined from 57,447 Java projects which provided by Allamanis et al. [2], extracted from migration rules that are manually validated and provided by Teyton et al. [78]. The accuracy of our approach is measured based on widely used metrics, TPR, and f-measure as follows:

### True Positive Rate (TPR)

It denotes the ratio of correctly extracted method mappings by all expected mappings.

\[
TPR(x) = \frac{V_x}{U_x}
\]

where \(V_x\) is the total number of valid mappings and \(U_x\) is the total set of manually validated mappings.

### f-measure

To measure which approach has better performance, we use f-measure as the weighted harmonic mean of both precision and recall.

\[
f - \text{measure}(x) = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]

---

\(^5\)Line 75 in JsonResponseBuilder.java, in the following commit: http://migrationlab.net/redirect.php?cf=icpc2019&n=1
To answer RQ2, we measure the ability of our SA approach in generating the expected mappings with fewer code fragments, in comparison with the three considered state-of-the-art approaches. Instead of providing each approach with all the code fragments, across projects, they are gradually fed with randomly selected fragments while measuring their performance in terms of extracting all the correct mappings. To perform this experiment, we used the manually validated mappings to create synthetic fragments. The performance of each approaches under study is tested in three different settings as follows:

- **(A) One-to-one**: All the randomly created fragments contain a random number of one-to-one mappings.
- **(B) Many-to-many**: All the randomly created fragments contain a random number of one-to-one, one-to-many, and many-to-many mappings.
- **(C) Library Documentation**: All the randomly created fragments contain a random number of one-to-one mappings. Breaking larger fragments with library documentation between methods is enabled.

For each setting, we perform three experiments where the size of randomly created fragments is 5, 10 and 20. We generate a different number of fragments between 5-1500. We select these ranges to cover all possible real-word scenarios. To deal with the stochastic nature of the experimentation, we run each experiment instance 30 times; then we take the average f-measure for each approach.

### 4.5 Results

#### 4.5.1 Results for RQ1.

We calculate the TPR, and f-measure of the mappings generated by SA approach as well as FC [77], FS [57], and MC [67].

Table 4.1 shows the results achieved by each of the four approaches applied on the same dataset. On average, we observe that the accuracy score (f-measure) achieved by SA is higher than the three other approaches by 12%. The performance achieved by SA could be justified by the intersection process which aims at detecting low-frequency mappings that are not detected by FC, MC, and FS. In terms of TPR, SA achieved the best average TPR score (82.1%). That is mean that SA detects a larger number of correct mapping in compare with three state-of-art approaches.
In summary, the qualitative analysis of 9 migration rules has demonstrated that SA’s accuracy (f-measure) has an average of 75.2% while the maximum accuracy scored of the other approaches is 63.3%. Thus, SA increased the accuracy of the state-of-art by 12%.

<table>
<thead>
<tr>
<th>Migration Rule</th>
<th>FC TPR</th>
<th>FC f-measure</th>
<th>MC TPR</th>
<th>MC f-measure</th>
<th>FS TPR</th>
<th>FS f-measure</th>
<th>SA TPR</th>
<th>SA f-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>logging→slf4j</td>
<td>25%</td>
<td>40%</td>
<td>13%</td>
<td>6%</td>
<td>46%</td>
<td>98%</td>
<td>93%</td>
<td></td>
</tr>
<tr>
<td>easymock→mockito</td>
<td>57%</td>
<td>51%</td>
<td>46%</td>
<td>44%</td>
<td>50%</td>
<td>50%</td>
<td>35%</td>
<td></td>
</tr>
<tr>
<td>testing→junit</td>
<td>56%</td>
<td>54%</td>
<td>49%</td>
<td>40%</td>
<td>76%</td>
<td>77%</td>
<td>90%</td>
<td></td>
</tr>
<tr>
<td>slf4j→log4j</td>
<td>78%</td>
<td>70%</td>
<td>73%</td>
<td>73%</td>
<td>86%</td>
<td>64%</td>
<td>100%</td>
<td>80%</td>
</tr>
<tr>
<td>json→gson</td>
<td>76%</td>
<td>43%</td>
<td>36%</td>
<td>33%</td>
<td>44%</td>
<td>53%</td>
<td>57%</td>
<td>53%</td>
</tr>
<tr>
<td>json-simple→gson</td>
<td>70%</td>
<td>50%</td>
<td>40%</td>
<td>40%</td>
<td>50%</td>
<td>55%</td>
<td>70%</td>
<td>56%</td>
</tr>
<tr>
<td>Collection→guava</td>
<td>78%</td>
<td>66%</td>
<td>78%</td>
<td>81%</td>
<td>73%</td>
<td>77%</td>
<td>78%</td>
<td>83%</td>
</tr>
<tr>
<td>gson→jackson</td>
<td>50%</td>
<td>36%</td>
<td>37%</td>
<td>40%</td>
<td>54%</td>
<td>65%</td>
<td>63%</td>
<td>69%</td>
</tr>
<tr>
<td>sesame→rifl4j</td>
<td>100%</td>
<td>72%</td>
<td>100%</td>
<td>72%</td>
<td>88%</td>
<td>72%</td>
<td>100%</td>
<td>88%</td>
</tr>
<tr>
<td>Average</td>
<td>65.5%</td>
<td>53.8%</td>
<td>52.4%</td>
<td>46.5%</td>
<td>58%</td>
<td>63.3%</td>
<td>82.1%</td>
<td>75.2%</td>
</tr>
</tbody>
</table>

4.5.2 Results for RQ2

Figure 4.3 shows how each approach performs on a different number of methods and fragments.

We observe from the Figure that SA clearly requires much less number of code fragments than the three other approaches to reach an f-measure score of 100% in all settings, namely (A) one-to-one, (B) many-to-many, and (C) library documentation. We notice that the achieved f-measure stabilizes after achieving 100%, regardless of the number of fragments.

Furthermore, results in Figure 4.3 indicate that FC follows a similar convergence pattern as SA, but it requires more fragments to reach 100%. In addition, we note that the FC approach achieves a relatively less f-measure in (B) many-to-many in comparison with (A) one-to-one, for the same number of methods and fragments. For example, in Figure 4.3-(A) with up to 10 methods per fragment, FC has an f-measure of 93% per 21 fragments, while it has an f-measure of 88% per 21 fragments in Figure 4.3-(B) with up to 10 methods per fragment.

We also observe that the f-measure score of MC approach cannot reach 100% regardless of the number of fragments, because this approach can not detect multiple mapping in one fragment. Increasing the number of fragments, increase the possibility of having more many-to-many fragments for that reason, f-measure goes down when we increase the number of fragments.

Another interesting observation is that the nature of FS relies on the closeness of the naming practices followed by the different library developers, to find good matching between methods. Thus, increasing the number of fragments does not increase the performance of FS, instead, involving
fragments with a larger set of methods increases the approach proneness to false positives, because it increases the probability of its inability to distinguish between methods when their signatures are similar. As shown in Figure 4.3, FS achieves an f-measure score of 69% in (A) one-to-one setting with 10 methods, then its f-measure score tends to decrease to 56%, when we increased the number of fragments to 1401 fragments. This observation applies to all experiment instances as shown in the figure. Moreover, we notice that the FS approach starts at a different f-measure score for different experiment instances, because it essentially depends on the similarity of the randomly selected mappings in the code fragments.

Figure 4.3: f-measure using randomly created 5-1500 fragments for 5, 10, and 20 method mapping, over 30 runs.

4.5.3 Discussions

In this section, we provide further discussions and insights about the obtained results.

Unresolved method mappings: Figure 4.3-(C) shows how different approaches perform when we enable the resolution of “unresolved” fragments that may have more than one valid mapping split using the similarity of method library documentation. We notice that SA is able to reach an
f-measure of 100% with a relatively less number of fragments, in comparison with its performance without the use of library documentation similarity.

Overall, the achieved results indicate that SA is less prone to false positives. Even in the cases when SA is unable to split larger fragments, it recommends them as is. For example, from the experiment instance reported in Figure 4.4-(A), we found some "unresolved" fragments that are generated by SA, but with no false positive results. The fragment is a combination of two correct method mappings toJSONString() → toString(), and get(int) → get(String). SA was unable to resolve these two fragments as it did not detect any fragment containing only either toJSONString() → toString() or get(int) → get(String).

For the FC approach, the unresolved fragments tend to generate large number of false positives. For instance, as shown in Figure 4.4-(B), one of the unresolved fragments achieved by FC contains a method named add(String, JsonElement) which could be replaced by any of the removed methods. This is a false positive mapping generated by FC as these two removed methods have the same frequency with these three added methods. Having these cases lead library documentation to generate false positives as well and may not detect the correct method mapping, in case the false added methods which a method that added to the fragment during the solving fragments process or by code refactoring has a high similarity with one of the removed methods.

![Fragment 1](A) Substitution Algorithm (SA)

![Fragment 1](B) Function Context (FC)

Figure 4.4: Example of "unresolved" Fragments.

For the MC approach, the size of unresolved fragments is most likely large, because MC cannot resolve many-to-many mappings. This leads to increasing the chances of false positives when com-
paring the similarity between a larger set of methods descriptions. Indeed, the less is the number of methods in code fragments, the more accurate are the results. For this reason, we did not observe a significant effect of similarity in methods descriptions for the MC approach. The methods description similarity can either increase or decrease the mappings accuracy for MC. For example, for ten methods in Figure 4.3-(B and C), MC achieved an f-measure of 5.2% in (B), when the number of the fragments reaches 101. While MC’s f-measure is 12.8% in (C), when the number of the fragments reaches 101. This indicates that the methods description similarity increases the coverage.

FS does not generate many-to-many unresolved fragments because it solves one-to-one or one-to-many method mappings only. Therefore, the methods description similarity does not have large effect on the approach’s accuracy.

<table>
<thead>
<tr>
<th>Number of methods</th>
<th>Times use library documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FC</td>
</tr>
<tr>
<td>2</td>
<td>89</td>
</tr>
<tr>
<td>5</td>
<td>497</td>
</tr>
<tr>
<td>10</td>
<td>2244</td>
</tr>
<tr>
<td>20</td>
<td>4035</td>
</tr>
</tbody>
</table>

Different approaches require a different number of times that approach use library documentation to detect method mapping from "unresolved" Fragments to reach the best TPR score. Table 4.2 shows that MC requires a large number of method calls in comparison with the three other approaches since it cannot resolve many-to-many mappings. We need to apply LD to address this limitation. While FC requires fewer method calls than MC, and greater than SA. While SA requires the minimum number of using library documentation to detect method mapping from "unresolved" Fragments in comparison with FC, and MC.

In summary, SA requires fewer fragments than existing approaches. Furthermore, the library documentation helps SA to reach 100% of f-measure score earlier. Therefore, this answers RQ2 indicating that SA is effective in detecting all the mappings, when compared to the three other approaches.

**Positive outcomes:** In scenarios where there is a sufficient fragments, both SA and FC can reach an accuracy score of 100%. However, SA is considered better as it requires less number of fragments to reach 100% accuracy.
Negative outcomes: On the other hand, an increasing number of fragments makes the accuracy of MC and FS worse. This could be mainly due to two different reasons. First, FS is able to map a limited number of methods that have a similar signature, so it does not rely on counting the number of detected mappings in the fragments. Therefore, increasing the number of fragments will not help in finding more mappings, instead of leading to more false/positives mappings. Second, MC maps blocks of code, thus increasing the number of fragments will increase the possibility of having segments with many-to-many methods, and which leads to more false positives.

In summary, it is not clear what is the best number of fragments that we should reach to reach the best accuracy for FS and MC. However, for SA and FC, it is intuitive that increasing the number of fragments helps in reaching a better accuracy.

4.6 Conclusion

This study addressed the problem of mining developer decision in migrating third-party libraries. We have described a novel approach that detects all the method mappings performed by developers when migrating between two different libraries. Our approach combines the mined method change patterns with method related documentation similarity to accurately detect mappings between removed and added methods. We evaluated our approach by mining the method-level changes of 9 popular library migrations across several open-source Java projects. The qualitative and comparative analysis of our experiments indicates that our approach significantly increases the accuracy of detecting manually-performed method-level mappings by an average accuracy of 12%, with fewer required fragments, in comparison with existing state-of-the-art studies.
Chapter 5

MigrationMiner Tool

5.1 Abstract

In this chapter we introduce, MigrationMiner, an automated tool that detects code migrations performed between Java third-party libraries. Given a list of open source projects, the tool detects potential library migration code changes and collects the specific code fragments in which the developer replaces methods from the retired library with methods from the new library. To support the migration process, MigrationMiner collects the library documentation that is associated with every method involved in the migration. We evaluate our tool on a benchmark of manually validated library migrations. Results show that MigrationMiner achieves an accuracy of 100%. A demo video of MigrationMiner is available at https://youtu.be/sAIR1HNetXc.

5.2 Introduction

The tremendous growth of available third-party libraries as being an integral part of modern software ecosystems, engendered new maintenance and evolution challenges. Typical challenges are mainly related to library APIs upgrade and migration as they often get deprecated or outdated. third-party library migration [76,77] is the process of replacing a library with a different one, while preserving the same program behavior. Unlike, upgrading a library from one version to another, the migration requires developers to explore and understand the new library’s API, its associated documentation, and its usage scenarios in order to find the right API method(s) to replace every
method, belonging to the retired library’s API.

Existing studies demonstrated that library migration is still a manual, error-prone, and time-consuming process [5,9,20,45,78]. Developers often spend a considerable time to check whether the newly adopted features do not introduce any regression in their client code. Indeed, recent studies have shown that developers typically spend up to 42 days to migrate between libraries [8]. Moreover, recent studies have shown that developers are reluctant to migrate their existing libraries, which makes their overall dependencies outdated and even vulnerable [45]. Hence, there is an urgent need to support developers in migrating their third-party libraries.

In this tool chapter, we present, MigrationMiner\(^1\), an open source tool that provides the developer with easy-to-use and comprehensive way of extracting, from given list of input projects, existing migrations between two third-party libraries using program analysis based on Abstract Syntax Tree (AST) code representation. In a nutshell, MigrationMiner (i) detects, (ii) extracts, (iii) filters, and (iv) collects code changes related to any performed migration. For a given input project, MigrationMiner detects any migration undergone between two java libraries and returns the names and versions of both retired and new libraries. Thereafter, MigrationMiner extracts the specific code changes, from the client code, and which belong to the migration changes (it should at least have one removed method from the retired library, and one added method from the new library) from all other unrelated code changes within the commits. Next, MigrationMiner filters code changes to only keep fragments that contain migration traces i.e., a code fragment, generated by the \texttt{diff} utility, which contains the removed and added methods, respectively from the retired and the new library. Finally, MigrationMiner collects the library API documentation that is associated with every method in the client code. The output of MigrationMiner, for each detected migration between two libraries, is a set of migration traces, with their code context, and their corresponding documentation.

To the best of our knowledge, there is no available open source tool that can extract migration traces between two different libraries. MigrationMiner is the first initiative to provide an open source tool and a dataset of automatically detected migrations\(^2\). Developers can use it to outsource from the \textit{wisdom of the crowd}, and extract migration patterns between two given libraries. Thus, developers can use it as a \textit{by-example} approach, to facilitate their migration process. Researchers can use it also to better understand the challenges associated with library migration and get practical insights.

\textbf{Tool, documentation and demo video.} MigrationMiner is publicly available as an open source

\(^1\)https://github.com/hussien89aa/MigrationMiner
\(^2\)http://migrationlab.net
5.3 MigrationMiner

In this section we detail the architecture and typical usage scenario of MigrationMiner as sketched in Figure 5.1. For each MigrationMiner component, we explain its input/output and workflow.

5.3.1 Data Collection

**Input.** MigrationMiner takes as input a list of open source GitHub projects, as shown in Figure 5.1. Due to the mining nature of our tool, we allow multiple project links to facilitate the automated search.

**Workflow.** The collection phase takes the list of open-source Java projects. It starts by cloning and checking out all commits for each project. For every commit, MigrationMiner collects its properties including the commit ID, commit date, developer name, and commit description. MigrationMiner keeps track of all changes in the project library configuration file, known as Project Object Model (pom.xml). All mined projects data is recorded in a SQL database for faster querying later when

---

3https://youtu.be/sAlR1HNetXc
identifying segments and fragments. As an illustrative example, Figure 5.2 shows a commit\(^4\) where \textit{json} was removed from the project while \textit{gson} was added.

**Output.** A list of potential library changes, and their corresponding commits, and projects.

```
- <groupId>org.json</groupId>
- <artifactId>json</artifactId>
- <version>20080701</version>
+ <groupId>com.google.code.gson</groupId>
+ <artifactId>gson</artifactId>
+ <version>2.3.1</version>
```

Figure 5.2: Migration from \textit{json} to \textit{gson}.

### 5.3.2 Migration Detector

**Input.** List of library changes, and their corresponding commits, and projects.

**Workflow.** Since developers may add and remove multiple libraries at the same time, there is no clear cut way to figure out the pairs of removed/added libraries. Therefore, the Cartesian Product (CP) is performed between the set of added removed libraries, in each parsed \textit{pom.xml} files, to extract all the possible combinations between removed/added libraries. Figure 5.3.A demonstrates the CP process in the form of a graph. Every node in the graph represents a library while the edge represents its potential mapping to another library. The edge is weighted by the number of times a migration is found, while parsing all commits, across all projects. For instance, the edge between \textit{json} to \textit{gson} has a weight of 12 because this migration has been identified 12 times during the data collection process. Since the CP generates every possible combination of rules, its result contains a large number of false positives. Thus, a two-step filtering process is performed:

1. In the first step, as shown in Figure 5.3.B, the weights are normalized by the highest outgoing weight per node, then the only mappings kept are those with a normalized weight that is higher than a user-defined filtering threshold value \(t_{rel} \in [0, 1]\). The value of \(t_{rel}\) controls the selection strictness. For example, when the filter \(t_{rel} = 1\), the \textit{json} to \textit{gson}, easymock to

---

\(^4\)https://github.com/vmi/selenese-runner-java/commit/641ab94e7d014cdf4fd6a83554dcf57130143d3
Mockito, and testing to Junit, migration are selected. MigrationMiner has $t_{rel} = 1$ by default, to guarantee a strict selection of rules.

2. The second filtering step is ensured by the Fragment Detection component, where only rules with actual migration traces at the method level are kept, i.e., for a rule $source \rightarrow target$, it is only kept if and only if there exists at least one or many method( )s from $source$ that has/have been replaced with one or many methods from $target$. The functionality of this component is detained in Section 5.3.4.

**Output.** Migration Rules with the highest weights.

![Diagram](image)

**Figure 5.3: Library Migration Detector.**

### 5.3.3 Segment Detection

**Input.** Migration Rules with the highest weights. Besides, list of library changes, and their corresponding commits.

**Workflow.** The purpose of the segments detection, i.e., migration periods, is to locate, for each Migration Rule, its time periods in all projects. As defined in the background section, a segment could be composed of one or many commits involved in the migration process. As shown in Figure 5.1, the segment detection phase starts with checking whether both libraries exist in the list of added/removed project libraries. Using static program analysis, MigrationMiner locates the end of the segment by scanning all commits in which all project source files are no longer dependent on the retired library.
Note that a migration does not require the physical removal of the library to be retired from the project, as the retired library may still loaded in project through its *pom.xml* file; however none of the library’s methods are used in the client code. Once a segment end is located, MigrationMiner keeps scanning previous commits in a backward fashion, looking for the start commit which contains the beginning of the migration, i.e., the first code change related to the replacement of any retired library method. After locating all segments for a given migration, it is important to keep track of source and target libraries versions for each segment to avoid backward incompatibility in case of an API change between two versions of the same library.

**Output.** Migration Rules with the highest weights, and their corresponding segments.

### 5.3.4 Fragment Detection

**Input.** Migration Rules with the highest weights, and their corresponding segments.

**Workflow.** Fragment detection generates source code fragments related to the library migration changes as shown in Figure 5.1. It clones the project source files that are changed in the commits belonging to the identified time segments. We apply the Git’s *Unified Diff Utility* command between the changed files to generate fragments, if any. A fragment is a continuous set of lines that have been changed along with contextual unchanged lines. Only fragments containing removed (resp., added) methods from the source (resp., target) library are considered valid. For example\(^5\), in Figure 5.1, for a given Migration Rule `json \rightarrow gson`, we identify that one of the fragments was converting object to string. We only keep `toJSONString()` as a removed method and `Gson()`, `toJson(Object)` as an added method. Other code changes (i.e., `String jsonText`) that do not belong to the migration, will be removed. As previously explained, all rules with no found fragment(s), will be automatically discarded.

**Output.** Filtered Migration Rules, and their corresponding fragments.

### 5.3.5 Documentation Collector

**Input.** Filtered Migration Rules, and their corresponding fragments.

**Flow.** As shown in Figure 5.1, for a given fragment, the documentation collector collects the API

---

5\(^{\text{line 180 in RuntimeConfig.java}}\), https://github.com/groupon/Selenium-Grid-Extras/commit/4d9bada8aeab5b09e7a27926fc9ecab8bb5a1b51
documentation for both source method(s) and target method(s). Based on its corresponding Migration Rule, it automatically downloads the library documentation as a \textit{jar} file for all library releases involved in migrations. Our approach relies on the libraries documentation on Maven Central Repository\textsuperscript{6}. The largest online Java library ecosystem hosting over 3,605,525 unique libraries\textsuperscript{7}, as of April 2019. The documentation collector then converts the API documentation from a \textit{jar} file to multiple \textit{HTML} source files using the \textit{doclet API}\textsuperscript{8}. It parses all of the \textit{HTML} files and collects the documentation related to class descriptions, method descriptions, parameter descriptions, return descriptions, package names, and class names. Finally, the documentation collector identifies the documentation associated with every method involved in any of the Migration Rules. For example, Figure 5.4 shows Gsons’ API documentation\textsuperscript{9} for the method String toJsonObject.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure54.png}
\caption{API documentation for \texttt{toJson(JsonElement)}}
\end{figure}

\textbf{Output.} Filtered Migration Rules, their fragments, and the documentation for each method in any fragment.

\section*{5.4 Case Study}

To evaluate the correctness of our detection process, we challenge our tool using an existing dataset provided by Teyton et al. [77]. This dataset contains 4 Migration Rules and their corresponding method mappings, detected across 16 projects, from which only 7 projects were using Maven, and

\footnotesize
\textsuperscript{6}central.maven.org
\textsuperscript{7}Statistics accessed on 5-5-2019 at https://search.maven.org/#stats
\textsuperscript{8}https://goo.gl/S3xRwk
\textsuperscript{9}https://www.javadoc.io/doc/com.google.code.gson/gson/2.2.2
so compatible with our tool. To challenge the ability of MigrationMiner to identify all the Migration Rules and their related mappings, we consider these 7 projects (that contain 3 migrations). Then, we compare our findings with the results of their manual detection to calculate the precision and recall. As shown in Table 5.1, MigrationMiner was able to detect all the Migration Rules, and all their corresponding fragments, achieving precision and recall of 100%. More interestingly, we have identified three additional Migration Rules, namely lucene → compass, jmf → gstreamer − java, jersey − client → wink − client, along with their fragments. We manually inspected and validated all the detected fragments in the client code. Thus, MigrationMiner achieved a precision of 100%. Since we did not manually investigate whether there are more unrevealed fragments, calculating the recall is not applicable for this case study.

Table 5.1: Accuracy of Migration Miner.

<table>
<thead>
<tr>
<th>Migration Rule</th>
<th>New?</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons − lang → guava</td>
<td>No</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>commons − io → guava</td>
<td>No</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>commons − lang3 → guava</td>
<td>No</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>lucene − core → compass</td>
<td>Yes</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td>jmf → gstreamer − java</td>
<td>Yes</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td>jersey − client → wink − client</td>
<td>Yes</td>
<td>100%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

5.5 Conclusion and Future Work

We presented MigrationMiner, an open source tool to detect migrations between third-party Java libraries. The evaluation of Migration Miner has shown its effectiveness in detecting manually validated migrations. MigrationMiner has been already used to detect and study the 9 various migration, detected in 57,447 projects, and this work has been published in the 27th IEEE/ACM International Conference on Program Comprehension [8]. As future work, we plan to extend MigrationMiner, to provide an interactive tool for the recommendation of library migration at the method level.
Chapter 6

MigrationMapper Tool

6.1 Abstract

Third-party software libraries reuse is becoming a common practice in software engineering. With the exponentially growing number of available and competing libraries in software ecosystems, migrating from one library to another is widely acknowledged to be a complex, time consuming and error prone activity. In this chapter, we introduce MigrationMapper, an automated tool that detects code migrations and recommends method mapping that performed between Java third-party libraries. Given a list of open source projects, the tool detects potential library migration code changes and collects the specific code fragments in which the developer replaces methods from the retired library with methods from the new library. To support the migration process, MigrationMapper builds on top of MigrationMiner [7] with a new feature that detects method-level mapping between added/removed libraries using Substitution Algorithm [8]. We evaluate our tool on a benchmark of manually validated library migrations. Results show that MigrationMapper achieves high accuracy in detecting migration code, and detect method mapping. A demo video of MigrationMapper is available at https://www.youtube.com/watch?v=D-01g2GjuTg.

6.2 Introduction

The tremendous growth of available third-party libraries as being an integral part of modern software ecosystems, engendered new maintenance and evolution challenges. Typical challenges are
mainly related to library APIs upgrade and migration as they often get deprecated or outdated. *third-party library migration* [76,77] is the process of replacing a library with a different one, while preserving the same program behavior. Unlike, upgrading a library from one version to another, the migration requires developers to explore and understand the new library’s API, its associated documentation, and its usage scenarios in order to find the right API method(s) to replace every method, belonging to the retired library’s API.

Existing studies demonstrated that library migration is still a manual, error-prone, and time-consuming process [4,5,6,9,20,45,78]. Developers often spend a considerable time to check whether the newly adopted features do not introduce any regression in their client code. Indeed, recent studies have shown that developers typically spend up to 42 days to migrate between libraries [8]. Moreover, recent studies have shown that developers are reluctant to migrate their existing libraries, which makes their overall dependencies outdated and even vulnerable [45]. Another study shows that the library migration tasks that given to developers who have many years of coding experiences more than migration tasks that given to new developers who have fewer years of coding experience, to reduce the possibility of introduction regression while performing the migration [6]. Hence, there is an urgent need to support developers in migrating their third-party libraries.

In this tool chapter, we present, MigrationMapper¹, an open source tool that provides the developer with easy-to-use and comprehensive way of extracting, from given list of input projects, existing method-level mappings between two third-party libraries using program analysis based on Abstract Syntax Tree (AST) code representation. MigrationMapper builds on top our previous tool MigrationMiner [7] by adding Substitution Algorithm [8] to detect method mapping from migration’s code changes (Fragments). MigrationMiner has shown good accuracy in detecting migration opportunities between a source and target library, i.e., library-to-library migration, and returns the list of source library methods $L_s = \{m_{s1},...,m_{sn}\}$ that are mapped to a list target library methods $L_t = \{m_{t1},...,m_{tm}\}$. However, while MigrationMiner provides a valuable support to developers to migrate from a library $L_s$ to $L_t$, it still requires extra effort to identify the particular method-to-method mappings, e.g., to which particular target method(s) from $L_t$, a source method $m_{s1}$ should be mapped to. To make the life of developers easier finding mappings in a lower level of granularity would be crucial, i.e., method-to-method mappings. MigrationMapper aims at identifying specific method-to-method mappings between a source and target library, $L_s$ and $L_t$, respectively. It returns a set of method mappings as follows: $M = \{<m_{s1}, m_{t1}>, <m_{s2}, m_{t2}, m_{t3}>, ..., <m_{sn}, m_{tn}>, \}$, meaning that the method $m_{s1}$ corresponds to $m_{t1}$ and the library $m_{s2}$ corresponds to both $m_{t2}$ and $m_{t3}$, etc. This method-to-

¹https://github.com/hussien89aa/MigrationMapper
method mapping would help the developer to find the specific correspondent target method(s) to a given method from his source library.

In a nutshell, MigrationMapper (i) detects, (ii) extracts, (iii) filters, (iv) collects code changes related to any performed migration, and (v) identifies mappings between source and target library’s methods. For a given input project, MigrationMapper detects any migration undergone between two java libraries and returns the names and versions of both retired and new libraries. Thereafter, MigrationMapper extracts the specific code changes, from the client code, and which belong to the migration changes (it should at least have one removed method from the retired library, and one added method from the new library) from all other unrelated code changes within the commits. Next, MigrationMapper filters code changes to only keep fragments that contain migration traces i.e., a code fragment, generated by the diff utility, which contains the removed and added methods, respectively from the retired and the new library. Then, MigrationMapper collects the library API documentation that is associated with every method in the client code. Finally, MigrationMapper runs Substitution Algorithm [8] that takes the extracted fragments and documentation as input to generate method mappings between the added and removed libraries. The output of MigrationMapper, for each detected migration between two libraries, is a set of method mapping traces, and their corresponding documentation as shown in Figure 6.6.

To the best of our knowledge, there is no available open source tool that can extract migration traces between two different libraries and detect method-to-method mappings. MigrationMapper is the first initiative to provide an open source tool and a dataset of automatically detected migrations\(^2\). Developers can use it to outsource from the wisdom of the crowd, and extract migration patterns between two given libraries. Thus, developers can use it as a by-example approach, to facilitate their migration process. Researchers can use it also to better understand the challenges associated with library migration and get practical insights.

This chapter extends our previous tool chapter [7] in the following ways:

- We introduce MigrationMapper, an extended version of MigrationMiner, that extracts method-to-method mappings from a source to a target library using a substitution algorithm.

- We conduct an experiment to evaluate the correctness of MigrationMapper in detecting method-to-method library migration opportunities and compare its advantages over MigrationMiner.

- We provide usage analytics of our original tool showing the usefulness from a broad range

\(^2\)http://migrationlab.net/index.php?cf=scp2020
CHAPTER 6. MIGRATIONMAPPER TOOL

of practitioners.

Tool, documentation and demo video. MigrationMapper is publicly available as an open source tool\(^1\), with a demo video\(^3\).

\[\text{Figure 6.1: MigrationMapper workflow and Architecture.}\]

6.3 MigrationMapper Architecture

In this section, we detail the architecture and typical usage scenario of MigrationMapper as sketched in Figure 6.1. For each MigrationMapper component, we explain its input/output and workflow.

6.3.1 Data Collection

Input. MigrationMapper takes as input a list of open source GitHub projects, as shown in Figure 6.1. Due to the mining nature of our tool, we allow multiple project links to facilitate the automated search.

\(^1\)https://www.github.com

\(^3\)https://www.youtube.com/watch?v=D-01g2GjuT
**Workflow.** The collection phase takes the list of open-source Java projects. It starts by cloning and checking out all commits for each project. For every commit, MigrationMapper collects its properties including the commit ID, commit date, developer name, and commit description. MigrationMapper keeps track of all changes in the project library configuration file, known as Project Object Model (`pom.xml`). All mined projects data is recorded in a SQL database for faster querying later when identifying segments and fragments. As an illustrative example, Figure 6.2 shows a commit 4 where `json` was removed from the project while `gson` was added.

**Output.** A list of potential library changes, and their corresponding commits, and projects.

```
- <groupId>org.json</groupId>
- <artifactId>json</artifactId>
- <version>20080701</version>
+ <groupId>com.google.code.gson</groupId>
+ <artifactId>gson</artifactId>
+ <version>2.3.1</version>
```

Figure 6.2: Migration from `json` to `gson`.

### 6.3.2 Migration Detector

**Input.** List of library changes, and their corresponding commits, and projects.

**Workflow.** Since developers may add and remove multiple libraries at the same time, there is no clear cut way to figure out the pairs of removed/added libraries. Therefore, the Cartesian Product (CP) is performed between the set of added removed libraries, in each parsed `pom.xml` files, to extract all the possible combinations between removed/added libraries. Figure 6.3.A demonstrates the CP process in the form of a graph. Every node in the graph represents a library while the edge represents its potential mapping to another library. The edge is weighted by the number of times a migration is found, while parsing all commits, across all projects. For instance, the edge between `json` to `gson` has a weight of 12 because this migration has been identified 12 times during the data collection process. Since the CP generates every possible combination of rules, its result contains a large number of false positives. Thus, a two-step filtering process is performed:

1. In the first step, as shown in Figure 6.3.B, the weights are normalized by the highest outgoing weight per node, then the only mappings kept are those with a normalized weight that is

---

4https://github.com/vmi/selenese-runner-java/commit/641ab94e7d014cd6f4d6a8554dceff57130143d3
higher than a user-defined filtering threshold value $t_{rel} \in [0,1]$. The value of $t_{rel}$ controls the selection strictness. For example, when the filter $t_{rel} = 1$, the json to gson, easymock to Mockito, and testing to Junit, migration are selected. MigrationMapper has $t_{rel} = 1$ by default, to guarantee a strict selection of rules.

2. The second filtering step is ensured by the Fragment Detection component, where only rules with actual migration traces at the method level are kept, i.e., for a rule $source \rightarrow target$, it is only kept if and only if there exists at least one or many method(s) from $source$ that has/have been replaced with one or many methods from $target$. The functionality of this component is detained in Section 6.3.4.

Output. Migration Rules with the highest weights.

![Figure 6.3: Library Migration Detector.](image)

6.3.3 Segment Detection

Input. Migration Rules with the highest weights. Besides, list of library changes, and their corresponding commits.

Workflow. The purpose of the segments detection, i.e., migration periods, is to locate, for each Migration Rule, its time periods in all projects. As defined in the background section, a segment could be composed of one or many commits involved in the migration process. As shown in Figure 6.1, the segment detection phase starts with checking whether both libraries exist in the list of added/removed project libraries. Using static program analysis, MigrationMapper locates the end of the segment by scanning all commits in which all project source files are no longer dependent on the retired library.
Note that a migration does not require the physical removal of the library to be retired from the project, as the retired library may still loaded in project through its pom.xml file; however none of the library’s methods are used in the client code. Once a segment end is located, MigrationMapper keeps scanning previous commits in a backward fashion, looking for the start commit which contains the beginning of the migration, i.e., the first code change related to the replacement of any retired library method. After locating all segments for a given migration, it is important to keep track of source and target libraries versions for each segment to avoid backward incompatibility in case of an API change between two versions of the same library.

**Output.** Migration Rules with the highest weights, and their corresponding segments.

### 6.3.4 Fragment Detection

**Input.** Migration Rules with the highest weights, and their corresponding segments.

**Workflow.** Fragment detection generates source code fragments related to the library migration changes as shown in Figure 6.1. It clones the project source files that are changed in the commits belonging to the identified time segments. We apply the Git’s Unified Diff Utility command between the changed files to generate fragments, if any. A fragment is a continuous set of lines that have been changed along with contextual unchanged lines. Only fragments containing removed (resp., added) methods from the source (resp., target) library are considered valid. For example\(^5\), in Figure 6.1, for a given Migration Rule json $\rightarrow$ gson, we identify that one of the fragments was converting object to string. We only keep JSONObject(String), toString() as a removed method and Gson(), toJson(Object) as an added method. Other code changes (i.e., String args) that do not belong to the migration, will be removed. As previously explained, all rules with no found fragment(s), will be automatically discarded.

**Output.** Filtered Migration Rules, and their corresponding fragments.

### 6.3.5 Documentation Collector

**Input.** Filtered Migration Rules, and their corresponding fragments.

**Flow.** As shown in Figure 6.1, for a given fragment, the documentation collector collects the API

\(^5\)line 52 in RollupRule.java, https://github.com/vmi/selenese-runner-java/commit/641ab94e7d014cdf4fd6a83554dce57130143d3
documentation for both source method(s) and target method(s). Based on its corresponding Migration Rule, it automatically downloads the library documentation as a jar file for all library releases involved in migrations. Our approach relies on the libraries documentation on Maven Central Repository\(^6\). The largest online Java library ecosystem hosting over 3,605,525 unique libraries\(^7\), as of April 2019. The documentation collector then converts the API documentation from a jar file to multiple HTML source files using the doclet API\(^8\). It parses all of the HTML files and collects the documentation related to class descriptions, method descriptions, parameter descriptions, return descriptions, package names, and class names. Finally, the documentation collector identifies the documentation associated with every method involved in any of the Migration Rules. For example, Figure 6.4 shows Gson’s API documentation\(^9\) for the method String toJson(JsonElement).

**Output.** Filtered Migration Rules, their fragments, and the documentation for each method in any fragment.

![Figure 6.4: API documentation for toJson(JsonElement)](https://example.com/figure6.4.png)

### 6.3.6 Substitution Algorithm

**Input.** Filtered Migration Rules, their fragments, and the documentation for each method in any fragment.

**Flow.** In this step, we generate method mappings from the identified fragments using our approach, which we label Substitution Algorithm (SA).

---

\(^6\)central.maven.org

\(^7\)Statistics accessed on 5-5-2019 at https://search.maven.org/#stats

\(^8\)https://goo.gl/S3xRwk

\(^9\)https://www.javadoc.io/doc/com.google.code.gson/json/2.2.2
Algorithm 2 Substitution Algorithm (SA)

**INPUT:** fragments - List of fragments, every fragment has list of added methods, list of removed methods.

**OUTPUT:** List of method mapping.

1: procedure Substitution(fragments)
2: loop:
3:     fragments ← HeapSort(fragments)
4:     for all fragment1 ∈ fragments do
5:         for all fragment2 ∈ fragments do
6:             ISet ← fragment1 ∩ fragment2
7:             if ISet then
8:                 fragments ← update(fragment1 - ISet)
9:                 fragments ← update(fragment2 - ISet)
10:                fragments ← add(ISet)
11:         goto loop.
12:     end if
13: end for
14: end for
15: newFragment ← LibraryDocumentation(fragments)
16: if newFragment then
17:     fragments ← newFragment
18:     goto loop
19: end if
20: return fragments
21: end procedure

Algorithm 3 Library Documentation (LD)

**INPUT:** fragments - List of fragments have N-M method mapping.

**OUTPUT:** newFragment - Fragment has one added, and one removed method that have highest similarity score between method’s description.

1: procedure LibraryDocumentation(fragments)
2:     maxScore ← Average of similarity score
3:     for all fragment ∈ fragments do
4:         for all rmFun ∈ fragment do
5:             for all addFun ∈ fragment do
6:                 score ← sim(rmFunDesc, addFunDesc)
7:                 if score >= maxScore then
8:                     newFragment ← rmFun, addFun
9:                     maxScore ← score
10:                end if
11:            end for
12:        end for
13:    end for
14: return newFragment
15: end procedure

SA starts by sorting the identified code fragments using Heap Sort (cf. Algorithm 2). The sorting process is based on two attributes. *First*, the number of methods per code fragment which refers to the number of added and removed methods in a fragment. *Second*, the frequency of a fragment, i.e., how many times a fragment appears across all projects.

If two code fragments have the same number of methods, then the fragment that has a higher frequency is moved before the fragment with lower frequency. The process moves the fragments that have less number of methods to the beginning of the list. Thereafter, SA iterates through all the identified fragments, starting from those with higher frequency, and searches for intersections $ISet \leftarrow fragment_1 \cap fragment_2$ between each fragment and the remaining set of fragments.
Two fragments are considered to bear a not-null intersection \( ISet \), if they share at least one common added and removed method. When an intersection exists, we remove shared methods \( ISet \) from fragment \( 1 \) using \( \text{fragment}_1 \leftarrow \text{update}(\text{fragment}_1 - ISet) \), and fragment \( 2 \) using \( \text{fragment}_2 \leftarrow \text{update}(\text{fragment}_2 - ISet) \), and we add \( ISet \) as new fragment \( \text{fragments} \leftarrow \text{add}(ISet) \). Then, the algorithm iterates back to the sorting process. This process continues until there are no more intersections that can be found between all fragments in the list.

The intuition behind this process resides on using fragments with \textit{one-to-one} and \textit{one-to-many} mappings, which are more frequent to be seen in the identified fragments, to reduce the cardinality of \textit{many-to-many} fragments, by splitting based on any common \textit{one-to-one} or \textit{one-to-many} mappings. Sorting the fragments prior to applying the intersection gives the opportunity to split larger fragments using smaller, yet relevant, fragments instead of performing random intersections between fragments. Our approach is based on \textit{Heap Sort} in this phase.

Once all intersections are completed, \textit{LibraryDocumentation} (fragments) (cf. Algorithm 3) iterates through fragments with \textit{many-to-many} mappings, with the aim of splitting them further using the lexical similarity. For each fragment, our approach calculates the similarity score \( \text{sim}(\hat{\text{rmFun}}_{\text{Des}}, \hat{\text{addFun}}_{\text{Des}}) \) between the description of the removed methods \( \hat{\text{rmFun}}_{\text{Des}} \), and the description of added methods \( \hat{\text{addFun}}_{\text{Des}} \) by following three steps:

\textit{First}, preprocess text (TPP) for removed methods \( \hat{\text{rmFun}}_{\text{Des}} \), and added methods \( \hat{\text{addFun}}_{\text{Des}} \) to clean text and put all words on root format. This process reduces noise when we calculate the similarity score.

\textit{Second}, calculate \textit{Term Frequency–Inverse Document Frequency} (TF-IDF) for prepossessed text of \( \hat{\text{rmFun}}_{\text{Des}} \), and \( \hat{\text{addFun}}_{\text{Des}} \) that generate vector of numeric numbers for \( W_{\text{rmFunDes}} \), and \( W_{\text{addFunDes}} \).

\textit{Third}, apply \textit{Cosine Similarity} between two vectors that is a measurement of how similar are two vectors based on the dot product of their magnitude [71]. If the similarity score is greater than or equal to the average similarity of the top detected \textit{one-to-one} correct mappings, then SA creates a new fragment \( \text{newFragment} \) that contains these two methods and restarts the intersection process.

SA terminates the search and returns a list of fragments, when there are no more intersections to be found, and no more newly created mappings based on methods similarity. Each fragment contains a unique method mapping.

\textbf{Output.} Detected methods mapping, and method’s documentation for every migration rule as shown in Figure 6.6.
CHAPTER 6. MIGRATIONMAPPER TOOL

6.4 Validation

In this section, we evaluate MigrationMapper performance comparing with existing state-of-the-art studies. Also, we highlight the first few months of our tool exposure to the community.

6.4.1 MigrationMapper Correctness

To evaluate the correctness of our detection process, we challenge our tool using an existing dataset provided by Teyton et al. [77]. This dataset contains 4 Migration Rules and their corresponding method mappings, detected across 16 projects, from which only 7 projects were using Maven, and so compatible with our tool. To challenge the ability of MigrationMapper to identify all the Migration Rules and their related mappings, we consider these 7 projects (that contain 3 migrations). Then, we compare our findings with the results of their manual detection to calculate the precision and recall. As shown in Table 6.1, MigrationMapper was able to detect all the Migration Rules, and all their corresponding fragments, achieving precision and recall of 100%. More interestingly, we have identified three additional Migration Rules, namely lucene − core → compass, jmf → gstreamer − java, jersey − client → wink − client, along with their fragments. We manually inspected and validated all the detected fragments in the client code. Thus, MigrationMapper achieved a precision of 100%. Since we did not manually investigate whether there are more unrevealed fragments, calculating the recall is not applicable for this case study.

<table>
<thead>
<tr>
<th>Migration Rule</th>
<th>New?</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons − lang → guava</td>
<td>No</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>commons − io → guava</td>
<td>No</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>commons − lang3 → guava</td>
<td>No</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>lucene − core → compass</td>
<td>Yes</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td>jmf → gstreamer − java</td>
<td>Yes</td>
<td>100%</td>
<td>N/A</td>
</tr>
<tr>
<td>jersey − client → wink − client</td>
<td>Yes</td>
<td>100%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

6.4.2 MigrationMapper vs. MigrationMiner

In this section, we want to show how helpful is the output MigrationMapper and MigrationMiner [7] to a developer in the context of migrating methods from a retired library to methods
of a new library. In particular, we want to show how MigrationMapper extends MigrationMiner output to make it more practical for developers to use:

MigrationMiner [7] detects the migration and distills it in terms of code fragments. Each fragment contains one mapping (one or many removed method(s) and one or many added method(s)). In a single fragment, there could be more than one migration (method mapping) applied, i.e., many removed and added methods within a single code chunk. In such case, it would be hard to distinguish between mappings. With such output format, research would benefit from such raw data, by analyzing the functional correspondence between removed methods, to better understand their usage. However, from a practical point of view, it made the tool’s output harder to understand.
In particular, for fragment with more than one mapping, developers need to manually untangle the mappings and figure out which removed/added methods belong to which mappings. This was the main drawback of MigrationMiner that MigrationMapper is mitigating. To give an illustrative example, in Figure 6.1, MigrationMiner [7] detects a fragment containing two removed methods `JSONObject(String), toString()` and two added methods `Gson(), toJson(Object)`. In such scenario, either this can be seen as a many-to-many method mapping (both removed and added methods belong to one single mapping), or they can be seen as two independent one-to-one
method mapping. Just with two mappings in the same fragment, the distinction between them is difficult, and so, with the existence of several potential mappings per fragment, the tool can become impractical.

Coming back to the example in Figure 6.1, it actually illustrates two separate mappings. The two constructors `JSONObject(String) → Gson()` are mapped, and the two methods `toString() → toJson(Object)` are mapped as well. Using MigrationMiner [7], this distinction is manual.

For the same example, MigrationMapper will be able to break down the tangled mappings. Looking at Figure 6.6, if a developer is only interested in migrating `get(int) → get(int)`, it is easier to find the corresponding method mapping by looking at MigrationMapper output Figure 6.5 rather than looking at MigrationMiner output Figure 6.5. This will drastically facilitate the usage of our mappings. As previously explained, this distinction is performed by running Substitution Algorithm [8] on migrationMiner output (fragments). The Substitution Algorithm breaks down the fragments into method’s mapping.

<table>
<thead>
<tr>
<th>Migration Rule</th>
<th>Average fragment size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MigrationMiner</td>
</tr>
<tr>
<td>easymock → mockito</td>
<td>4.36</td>
</tr>
<tr>
<td>testng → junit</td>
<td>2.28</td>
</tr>
<tr>
<td>commons − logging → slf4j</td>
<td>2.14</td>
</tr>
<tr>
<td>google − collect → guava</td>
<td>2.42</td>
</tr>
<tr>
<td>slf4j − api → log4j</td>
<td>2.26</td>
</tr>
<tr>
<td>gson → jackson</td>
<td>3.61</td>
</tr>
<tr>
<td>commons − lang → slf4j-api</td>
<td>3.75</td>
</tr>
<tr>
<td>json → gson</td>
<td>3.64</td>
</tr>
<tr>
<td>Average Size</td>
<td>≈ 3</td>
</tr>
</tbody>
</table>

To further elaborate on the difference between MigrationMiner [7], and MigrationMapper, we run both tools on a dataset provided by Alrubaye et al. [8]. For a given Migration Rule, we calculate the average of number of methods per fragment. The size of a fragment is calculated by summing number of added/removed methods in a fragment. The Fewer methods are in a fragment the better because it requires less decision making by the developer. As shown in Table 6.2, the MigrationMapper can break even large size of fragments into small fragments. For example, if we consider the following rule `easymock → mockito`, the average methods per fragment for MigrationMiner [7] is more than four methods. Using MigrationMapper, we were able to break down the average method in fragments to approximately of two methods per fragment.
As a summary, the approximate average of methods in a fragment, generated by MigrationMiner, are three methods, which means at least there are two methods removed and one method added or vice versa. While the average methods in a fragment, generated by MigrationMapper, are two, which means, on average, a fragment has one method removed, and one method added.

6.4.3 Practical Impact

In this study, we want to show how useful is our tool to the software engineering practitioners community. The original tool MigrationMiner [7] was published in August of 2019. We upgraded MigrationMiner to MigrationMapper in December 2019. As of April 13th 2020, MigrationMiner attracted 128 stars and 137 forks on the GitHub repository [9]. Moreover, Figure 6.7 shows the usage analytics of our tool since its initial release until April 13th 2020. There are around 2,134 unique users from 97 different countries who visited the website, which contains the link to the tool, its documentation, and the dataset. Most of these users came back to the website through 3,977 sessions. On average, a visitor stays 6.5 minutes in the website. From this data and with the important extension of MigrationMapper, we expect that the tool will attract broader audience and usage in the future from software developers and researchers in both open source and industrial projects.
6.5 Conclusion and Future Work

We presented MigrationMapper, an open source tool to detect method-level mapping between third-party Java libraries. The evaluation of MigrationMapper has shown its effectiveness in detecting manually validated migrations. MigrationMapper has been already used to detect and analyze 9 various migrations, detected in 57,447 projects, and this work has been published as a research paper in the Applied Soft Computing journal [6]. As potential usage of the tool, MigrationMapper is not limited to library migration, it can be used for detecting mappings in library upgrades as well. As future work, we plan on extending MigrationMapper, an providing an online API that can be directly be used by developers, in order for us to increase the usage of the tool.
Chapter 7

The Impact of API Migration on Software Quality

7.1 Abstract

The process of migration between different third-party software libraries is hard, complex and error-prone. Typically, during a library migration process, developers opt to replace methods from the retired library with other methods from a new library without altering the software behavior. However, the extent to which the process of migrating to new libraries will be rewarded with improved software quality is still unknown. In this chapter, our goal is to study the impact of library API migration on software quality. We conduct a large-scale empirical study on 9 popular API migrations, collected from a corpus of 57,447 open-source Java projects. We compute the values of commonly-used software quality metrics before and after a migration occurs. The statistical analysis of the obtained results provides evidence that library migrations are likely to improve different software quality attributes including significantly reduced coupling, increased cohesion, and improved code readability. Furthermore, we release an online portal that helps software developers to understand the pre-impact of a library migration on software quality and recommend migration examples that adopt best design and implementation practices to improve software quality. Finally, we provide the software engineering community with a large scale dataset to foster research in software library migration.
CHAPTER 7. THE IMPACT OF API MIGRATION ON SOFTWARE QUALITY

7.2 Introduction

Prior studies show that software maintenance activities consume up to 70% of the total life-cycle cost of a typical software product [13]. One of the important software maintenance activities in modern software development is third-party library migration [76,77]. In practice, library migration can be seen as the process of replacing a library with a different one, while preserving the same program behavior. The library migration process tends to be a manual, error-prone, and time-consuming process [5,9,20,45,78]. Hence, developers have to explore and understand the new library’s API, its associated documentation, and its usage scenarios in order to find the right API method(s) to replace in the current implementation belonging to the retired library’s API. As a consequence, developers often spend considerable time to verify that the newly adopted features do not introduce any regression. Indeed, previous studies have shown that developers typically spend up to 42 days to migrate between libraries [8].

Unlike library upgrades, library migration typically requires more fine-grained code changes and refactorings, e.g., changing types of variables and parameters, renaming attributes and methods, etc., since developers need to accommodate the syntactic and semantic mismatch between the added and removed methods [76]. These refactoring changes may account for the overhead needed to fulfill the migration and adjust the existing software design to the newly introduced methods. Even if refactoring is perceived to be one of the best software engineering practices for restructuring code to improve its quality [73], the intention behind API-related refactoring operations might be different. Typically, API migrations introduce a set of methods and objects with different lexicality and naming conventions, which have to be integrated into the existing codebase terminology. That is, developers may refactor their code during along with the migration to contextualize the new library methods. These unintended refactorings have an impact on software design metrics (e.g., cohesion, coupling, etc.) [19] as well as the changes in terminology and renaming activities affect code readability [1,15,65].

Various studies have focused on analyzing the impact of API evolution on software quality in terms of change and bug-proneness [41, 50, 64], software usability and rating [12, 47]. Other studies focused on estimating the impact of API documentation on the library adoption and usability has been investigated in the literature [27, 63]. Moreover, recent studies attempted to identify traces of manually performed library migrations. They provide the community of a set of real-world migrations between popular Java libraries, in various open source projects [8,77,78].

Existing studies reveal the importance of taking into account the software design characteristics
when performing the migration to reduce maintenance costs. However, there is little knowledge on the impact of API migration, and its related refactoring changes, on the quality of software’s design as well as code comprehension and readability.

As software systems evolve rapidly, there is a need for appropriate tools, reliable, and efficient techniques to support developers in replacing their deprecated library APIs with up-to-date ones, and maintaining/improving the quality of their software design.

To address the above-mentioned issues, we conduct a large-scale empirical study to assess the impact of library migration on both software design quality and code comprehension. We consider an existing dataset of 9 popular migrations between Java libraries, mined from 57,447 open-source Java projects [8]. Afterward, we shortlist all commits containing traces of method swaps, as part of any of considered migrations. We refine our dataset by untangling each commit to identify the specific code elements involved in the migration using program analysis. Then, for the selected code elements, we calculate the values of their corresponding design and readability metrics, before and after the migration. Finally, we statistically compare the variation of these values, to analyze whether the migration had a significant, positive, or negative impact on design quality and readability. To better understand the variation of these values, we use refactoringMiner [81] to extract the refactoring activities that were associated with the migration process. We finally associate a ranking score, to each migration trace, according to the extent to which it was able to improve the design and readability of the existing code. Furthermore, We survey 10 senior developers to assess the usefulness of the ranking score in providing better migration examples.

Our study is driven by the following research questions:

**RQ1. (Design Improvement) What is the impact of library migration on the quality of software design?**

To answer this research question, we assess the impact of library migration on software design quality, in terms of complexity, coupling and cohesion, widely popular structural metrics [69], and previously used in similar empirical studies [18,62]. For each analyzed source file in the dataset (that we detail later in the next subsection), we measure the value of its coupling and cohesion before and after the migration. As we aggregate all values before and after the migration, we observe the variation in the aggregated values to investigate whether the migration had a positive or negative impact on design quality.

**RQ2. (Code Readability) Does migration improve the code readability?**
Similarly to RQ1, we consider popular state-of-the-art readability tools and metrics [15,66]. For each metric, we measure its pair values in the dataset files, before and after the migration, and then we analyze the values for statistical significance.

RQ3. (Refactoring Operations) What types of refactoring changes do developers perform during library migration?

We explore, in this research question, design-related change patterns, observed across various migrations. We aim at understanding what are the most solicited refactoring operations that facilitate the integration of the target API methods.

RQ4. (Quality Recommendation) Can we leverage design and readability metrics to recommend better code examples for migration?

Since there are multiple code fragments, belonging to various projects and containing the same mappings, we design a recommendation-based ranking method that aggregates various quality metrics. Our method ranks the collected code fragments based on the extent to which they preserve the design coherence and improve the code comprehension. We then perform a qualitative study with 10 senior developers to evaluate the usefulness of our recommendation-based ranking method.

The chapter key findings show a positive variation of structural and readability metrics, i.e., developers do pay attention to design and readability when performing the migration process, through applying a variety of refactoring operations to bridge the syntactic gap between the replacing and replaced methods. Moreover, results show that code fragments with higher ranking score were also voted by the majority of developers, as good examples of migrations. This study makes the following contributions:

1. We release an online portal\(^1\) that showcases real-world migration fragments, with their corresponding positive or negative impact on coupling, cohesion, and readability.

2. We propose a ranking score, that we label Migration Quality Score (MQS), for recommending migration examples that ensure better software quality and comprehension.

3. We survey with senior software engineers at an outstanding company\(^2\) to evaluate MQS’s ability to recommend high-quality migration examples for 9 popular migrations. Findings

\(^1\)http://migrationlab.net/index.php?cf=saner
\(^2\)Hidden for double-blind review
show that MQS effectively recommends high-quality migration examples.

Figure 7.1: Experimental Design Overview.

7.3 Empirical Setup

7.3.1 Data Collection

Figure 7.1 provides an overview of our study workflow. To measure the impact of library migration on software quality attributes, we need to analysis the source code before and after library migration has happened. To do so, we used MigrationMiner [7], which is a tool used to detect library migration at the method level. Given 57,447 GitHub Java projects which provided by Allamanis et al. [2] as input to MigrationMiner [7], The tool detects 8,938 migration commits where a developer migrates the project’s source code from using library A to library B (ex easymock → mockito). To analysis the impact of library migration on code quality, We run Understand$^3$, readability.jar [15,66] and RefactoringMiner [70] on migration commits (Commit N) and a commit before migration (Commit N-1).

$^3$https://scitools.com/
Each migration commit contains at least one or multiple mappings, i.e., fragments of code containing one or multiple removed methods, being replaced with one or multiple added methods, along with other code changes that may or may not be related to the migration. Since any code change, non-related to migration represents a noise for this study, we only consider files containing migrations fragments in each migration commit. We notice that some migrations are instant, i.e., all method replacements are located in the same commit, but in multiple source files, and some migrations are delayed, i.e., method replacements are scattered across multiple commits.

The data collection process has analyzed commits belonging to a diverse set of 57,447 projects. We have identified 36,023 classes, each contains at least one mapping. We also enumerated 9,380 unique mappings, already showcased in the dataset’s website. We identified 3,579 refactoring operations that are associated with these mappings. We provide our collected data for replication and extension online.

### 7.3.2 Metrics Measurement

#### Structure and size metrics

To collect the design metrics, we use, Scitools Understand, a static analysis framework that captures a variety of structural metrics, across languages such as C++ and Java. Based on the computed metrics values, we can calculate the effect of migration-related changes on the system design. In particular, we analyze the following size and structure metrics: Coupling Between Objects (CBO), normalized Lack of Cohesion (LCOM), and Cyclomatic complexity (CycC).

Since each source file may contain multiple migration fragments, and since we only care about
these specific files, we calculate metrics only for these fragments and then we average them to construct one value per file. In other terms, each data point in our analysis is a file with an average metric value.

**Code readability metrics**

Source code readability is one of the important aspects of software engineering. Several studies have been focusing on the automation of its approximation through deep static analysis. In this context, we measure code readability during the migration process using two state-of-the-art metrics, proposed by Buse and Weimer [15], and Scalabrino et al. [66]. We deploy both metrics as they were widely-employed in recent empirical study [62], and because they address different readability aspects. On the one hand, Buse and Weimer’s Readability metric (BWR) combines the source code size characteristics to approximate its readability. On the other hand, Scalabrino et al.’s Readability metric (SR) does not only look at the structural characteristics of code, and adds another lexical dimension, in which it considers more linguistic properties such as comments consistency with the source code and its coherence etc. Both metrics generate a score that, the higher it is, the better is the readability of the code.

Similarly to structural metrics, each data point in our analysis represents an average readability score per source file.

**Refactoring operations collection**

To extract the refactoring history of the selected commits, we use RefactoringMiner [70], an accurate state-of-the-art tool that can detect refactoring operations that are applied in the development history of a Java project. RefactoringMiner parses the source code in each commit, and returns a summary of applied refactoring operations such as a change in parameter type, moves attribute, renames attribute, renames parameter, renames method, renames variable, extracts class, etc. We selected this tool because of its high accuracy [75,81] (precision of 98%, and recall of 87%), and because it is designed to mine refactorings from commit history, which perfectly matches our study context.

After applying these tools on all predefined *mappings commits*, before and after the migration, we generate a dataset that contains, for each commit, its associated code fragments, structural and readability metrics pairs of values, any detected refactoring operation(s). We then use this dataset
as a base of examples that we rank according to how much they improve quality and comprehension. We detail our proposed ranking model in the following Section 7.3.3.

### 7.3.3 Ranking Model

The migration dataset [8] contains, for each migration rule, e.g., easymock to mockito, several commits, extracted from various projects, containing similar mappings. Therefore, for the same mapping, there are various real-world examples of how a deprecated method has been replaced with one or multiple replacing methods. Although these examples exhibit similar sets of removed/added methods, they differ in their overhead in the software design, since the migration process is subjective [9,77,78], and developers may perform different types of code changes to perform the same type of migration. Moreover, as maintaining a good quality of the source code, in terms of design and readability, is critical for code longevity, our aim is to favor the recommendation of source code migration examples that correctly execute the migration while also maintaining, or improving the current client code quality. To do so, we simply leverage the existing metrics, previously explored in research questions, and combine them into an overall *Mapping Quality Score* (MQS). For each given migration in the dataset, we loop through all its mappings, for each mapping, we locate all its instances in the course code (\(\text{inst}\)). Then, for each instance, we calculate its MQS, and finally, we rank them on a descendent order, to favor examples with the highest quality improvement. Formally, we calculate the MQS as follows:

\[
MQS(\text{inst}) = \sum_{i \in m} \varphi_i(\text{inst})
\]  

where MQS represents the weighted sum of the software quality attributes (\(\varphi_i\)). The term \(m\) is a set, \(m = \{\text{CBO, LCOM, CycC, CLOC, LOC, CR}\}\). The term \(\text{inst}\) denotes code instances to be ranked for a given mapping.

Since the combined metrics do not belong to the same scale, we normalize them using *min-max normalizer* that linearly rescales every metric value to the [0,1] interval. Rescaling in the [0,1] interval is done by shifting the values of each feature \(x\) so that the minimal value is 0, and then dividing by the new maximal value (which is the difference between the original maximal \(\max(x)\) and minimal \(\min(x)\) values).

Moreover, since not all metrics are to be maximized, we transform all of them to be minimized using the duality principle. For example, since the lower are the values of coupling, the better they are,
Table 7.2: Statistical significance of samples difference, before and after API migration, for each of the considered metrics.

<table>
<thead>
<tr>
<th>Metric</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCOM</td>
<td>$1.06 \times 10^{-75}$</td>
</tr>
<tr>
<td>CBO</td>
<td>$8.11 \times 10^{-148}$</td>
</tr>
<tr>
<td>CycC</td>
<td>$4.78 \times 10^{-131}$</td>
</tr>
<tr>
<td>BWR</td>
<td>$3.95 \times 10^{-62}$</td>
</tr>
<tr>
<td>SR</td>
<td>$3.40 \times 10^{-12}$</td>
</tr>
<tr>
<td>Refactoring Operations</td>
<td>0.013</td>
</tr>
</tbody>
</table>

we maximize the complement of the normalized value of coupling, i.e., $\phi_{CBO} = (1-z(CBO(src)))$, where $z$ returns the min-max normalized value.

As an illustrative example, we observe in Figure 7.1 that for a given mapping between `createStrictMock`, belonging to the removed library `easymock`, and `mock`, belonging to `mockito`, 4 instances are being shown and recommended as migration examples. Note that each example contains a link to the actual location of the code on GitHub. The examples have been ranked according to their MQS. For instance, the first example has the highest MQS of 2.475, while the second example has an MQS of 2.239.

Note that the normalization was restricted to the MQS calculation, we still use the actual raw values of the metrics for the results, which are detailed in the following sections. Also note that we weights for the actual MQS score are by default equal to 1 i.e., for this study, we consider all metrics to be equally important, and thus, this can be improved, if any metric has been found to be more influential than others in this context of API migration.

### 7.4 Results

This section details the results of our empirical setup to answer the research questions.
7.4.1 RQ1. (Design Improvement) What is the impact of library migration on the quality of software design?

Figure 7.2 outlines the box plots of the values, for each of the structural metrics, calculated before and after the migration. To better understand the statistical significance of the observed results, we setup our statistical analysis as follows: for each metric, we cluster its values according to whether it was measured before or after the migration. We apply this to each code fragment. As a result, we create two groups of equal size, each containing measurements of the same metric before and after the migration. Then, we use the Wilcoxon signed rank test, since these groups are dependent (measurement on the same code fragments), to evaluate the significance of the difference between the values, in terms of their mean.

![Box plots of CBO, LCOM, and average CycC values](image)

Figure 7.2: Box plots of CBO, LCOM, and average CycC values, extracted from migrated code fragments, before and after the migration (lower values are better).

Our Null hypothesis indicates no variation in the metric values of pre- and post-migrated code elements. In contrast, the alternative hypothesis advocates for a variation in the metric values. In this research question, a decrease in the mean values is considered desirable (i.e., an improvement in design quality). Additionally, the variation between values of both sets is considered significant if its corresponding p-value is less than 0.05 (a confidence level of 95%). We deploy the same statistical analysis for RQ2 as well, but with a difference in the interpretation, since for readability metrics, an increase in mean value is considered desirable.

As can be seen in Figure 7.2, for the coupling between objects metric (CBO), we clearly notice a general trend of values being significantly decreased, just after the migration. The mean CBO
value has decreased from 2.047 to 1.884 \((p - value < 0.05)\), and the upper quartile has become significantly lower while decreasing from 2.147 to 1.955. Interestingly, we also observe from the figure a similar trend for the Lack of Cohesion of Methods metric (LCOM), since its mean value has gone from 0.548 to 0.482 \((p - value < 0.05)\). We also notice a drop in the lower quartile, going from 0.460 to 0.370.

As for the average Cyclomatic complexity, there is a slight decrease in the upper quartile, varying from 2.146 to 2.050, but the mean value has decreased from 1.593 to 1.505 \((p - value < 0.05)\).

Figure 7.3: Illustrative example of a code migration from log4j to slf4j, with a positive impact on coupling.

To better understand the observed results, we manually analyze few random instances. Figure 7.3 illustrates a code fragment example of such migrations, extracted from Github \(^6\). In this fragment, the methods addPackage with addClasses, belonging to the library log4j, is being replaced with the method addClasses, from slf4j. We can observe the difference in the used parameters between the replaced and replacing methods. More precisely, addPackage with addClasses have a CBO of 4, while addClasses only have a CBO of 3, which did improve the overall CBO of all methods by adopting this newly deployed method.

Another interesting example \(^7\), shows how the newly introduced object DefaultHttpClient does not rely on any parameter, unlike the retired object HttpClient whose constructor is initialized with connectionManager. Therefore, the new object is more cohesive and it reduces the lack of cohesion of the system.

---

\(^6\)https://github.com/aerogear/aerogear-unifiedpush-server/commit/4861157566723bc3179b69d0755e5bf5460d9729

\(^7\)https://github.com/anthonydahanne/ReGalAndroid/commit/6410cc8a12246745b19a102da5dd2e92d326b9f9
CHAPTER 7. THE IMPACT OF API MIGRATION ON SOFTWARE QUALITY

7.4.2 RQ2. (Code Readability) Does migration improve the code readability?

Figure 7.5 outlines the boxplots of the values, for each of the readability metrics, calculated before and after each API migration.

For the BWR [15] metric, we observe an improvement in its values. In particular, the mean BWR [15] value has increased from 0.474 to 0.482 ($p - value < 0.05$). Similarly, the lower and the upper quartiles have slightly increased respectively from 0.316 to 0.329, and 0.579 to 0.587. As for the second readability metric, namely SR [66], the improvement is more significant since its mean value exhibits an increase from 0.568 to 0.603 ($p - value < 0.05$). The increase is also seen in the lower quartile, going from 0.461 to 0.484, whereas the upper quartile exhibits a slight decrease from 0.709 to 0.706.

If we take deeper look into the code example\(^8\), illustrated in Figure 7.6, we notice that the developer just moved from using the method `put`, from `json` to the method `addProperty`, from `gson`. Note that the developer did not perform any additional activities; however the BWR [15] improved from 0.0013 to 0.0023 since the method name `addProperty` has better readability score than `put`, as shown

---

\(^8\)https://github.com/groupon/Selenium-Grid-Extras/commit/4d9bada8aeab50b9e7a27926fe9ecab8bb51b51
in the console output of BWR [15] in Figure 7.6.

Figure 7.5: Box plots of BWR and SR values, extracted from migrated code fragments, before and after the migration (higher values are better).

```java
public void addKeyValues(String key, int value) {
    checkIfKeyDescriptionExist(key);
    - keyValues.put(key, value);
    + keyValues.addProperty(key, value);

    keyValues.put(key, value);
    ###
    0.0013698068214580417

    keyValues.addProperty(key, value);
    ###
    0.002353090327233076
```
CHAPTER 7. THE IMPACT OF API MIGRATION ON SOFTWARE QUALITY

Summary for RQ2. API migrations do improve code readability, as both BWR [15] and SR [66] readability metrics experience a significant increase when comparing code fragments before and after the migration.

7.4.3 RQ3. (Refactoring Operations) What types of refactoring changes do developers perform during library migration?

Replacing one method from the retired library, with another one from the new library has a syntactic overhead that typically trigger non-functional code reword that varies from renaming variables, parameters, to changing types, and even moving/extracting code elements. As our goal is to assess whether developers change and refactor differently when they migrate, we need to compare also the refactoring activities in other regular commits in which there was no APIs migration performed, to get appropriate statistical analysis. In other terms, we need to evaluate whether the changes and refactoring are related to the migration or any other factor. Indeed, causal inference stems from the social sciences and explores cause and effects as its main concern. In econometrics, Difference-In-Differences (DID) methods are one of the key analytical elements for causal inference [10]. We adopted the DID method in our analysis to statistically visualize actual and counterfactual scenarios, thereby enabling a causality analysis. DID consists of comparing two groups, one with the intervention (i.e., migration) and one without it.

Indeed, DID depends on the common trends assumption [10] based on the selection of an appropriate control group. We selected our control group (i.e., code fragments that did not exhibit API migration) using the propensity score matching since it is a popular matching technique. In particular, we used the well-known nearest neighbor matching algorithm in propensity score matching based on the following characteristics: the subsystem, the source file size, the contributor who applied the refactoring, and the period of time (the same month). A total of 3,579 refactoring operations were identified as a control group, to have an equal group to our current dataset size as described in Table 7.1.

Figure 7.7 shows compassion between refactoring operations that happen during migration activities with the refactoring of our control group in terms of the percentage of applied refactoring. Overall, we found that the distribution of refactoring in migration commits and other commits are statistically different ($p-value = 0.013$), as reported in Table 7.2. This finding indicates that developers do refactor and change their code differently when they perform API migration. In particular, as can be seen in Figure 7.7, we find that developers are likely to change the parameter type, rename variables and rename attributes when they migrate their APIs. The results make sense because the developer may refactor her/his client code around the method of the retired library to
Figure 7.7: Distribution of refactoring operations per type, before and after the migration.

map the code and match the requirements of the method from the new library. Such refactoring
Figure 7.8: Illustrative example of a code migration from `json` to `gson`, being supported by applying a change parameter type refactoring.

may facilitate the migration by adjusting the existing code elements to match the signature of the added method(s). Indeed, this explains the high rate of type change refactoring, being performed along with various rename refactoring to bridge the lexical gap between the existing codebase and the introduced API.

Moreover, as can be seen in the figure, while in regular refactoring commits, developers most likely to apply extract method, rename method, and move class/attribute/method refactoring, the refactoring practices has changed in migration commits. As an illustrative example, Figure 7.8 shows a sample of migration code for developer changed method parameter type from `JSONObject` to `JsonObject`, while refactoring the code to migrating from method `add` that belongs to `json` to the method `addProperty` that belongs `gson`.

Summary for RQ3. Developers change the way they refactor their code during API migrations by focusing more on applying refactoring operations that facilitates the integration of the added methods. We highlight operations such as change parameter type, rename parameter and rename attribute.

---

---

9https://github.com/groupon/Selenium-Grid-Extras/commit/4d9bada8aeab5b09e7a27926fc9ecab8bb5a1b51 in FirstTimeRunConfig.java
7.4.4 RQ4. (Quality Recommendation) Can we leverage design and readability metrics to recommend better code examples of migration?

To evaluate our ranking model based on the structural and readability metrics, we conducted a qualitative analysis with 10 senior developers from an outstanding software development company. All the participants volunteered to participate in the experiment and were familiar with Java programming, Maven ecosystem, and API usage. The experience of these participants with Java development is 10+ years. Prior to the experiment, the participants were provided with a 30-minutes tutorial on the tool usage and the experiment process. Each participant was provided with 10 code fragments to perform 10 migration tasks between libraries including easymock to Mockito, and json to gson. Then, for each of the migration tasks, the developer runs our migration code examples tool that returns a list of examples but exposed to the developers at a random order (at least for our experimental study to avoid biased selection from a ranked list). Then, the developer reviews all the returned examples and picks the top-3 examples that fit her/his preferences and the quality of the examples.

Figure 7.9 reports the survey results, where the x-axis represents the index of example \( k \) in the ranked list, and the y-axis represents the number of times an example \( @k \) has been chosen by a developer as their top choice, divided by all choices. In other terms, the y-axis percentage of developers’ choice of an example whose rank is \( k \). For instance, the value \( @k = 1 \) is the percentage of how many times the example number one in the ranked list was chosen at the best example.

According to Figure 7.9, we could see that 59% of developers agreed that the first recommended example is the best example. If we allow the top-2 ranked examples \( (k <= 2) \), our recommendation already captures 80% of developers’ choices, which also improves further to become 94% for top-3 ranked examples \( (k <= 3) \).

We can conclude that our ranking model efficiently recommends what developers consider to be their decision if they are requested to perform the migration.

| Summary for RQ4. | The qualitative analysis of our ranking model shows its efficiency to considerably prune the search space for developers when they are searching for good migration examples. Our ranking score was able to match 59% of the developer’s chosen examples when recommending top-1 example. |
7.5 Threats to validity

We report, in this section, potential factors that can threaten the validity of our empirical study.

7.5.1 Internal Validity.

Our empirical analysis is mainly threatened by the accuracy of the migration dataset. Since our assumption that all studied commits carried at least one migration, any intruding files would be considered as noise to our analysis. We did not perform any rigorous verification concerning the correctness of the dataset, but we did perform various manual checks when gathering the files for statistical analysis and for qualitatively analyze our findings, and we did not notice any single case where the file we were investigating did not contain at least one migration trace.

The second main threat to the validity of our work is the choice of the metrics used in this study. We have chosen coupling, cohesion, and complexity, as being representative to design quality and
popular metrics, being used in similar empirical studies [16,18].

The non diverse set of developers, along with the randomness in assigning them the examples, has a direct impact on the results. The choice of experienced and volunteers was to reduce the effect of non interest to the problem resolution. Developers were genuinely interested to support the work, and they were aware of it being potentially published for the community.

7.5.2 Construct validity

Threats to construct validity describe concerns about the relationship between theory and observation and, generally, this type of threat is mainly constituted by any errors related to measurements. More precisely, any error in the used tools directly impacts the correctness of our findings. For calculating metrics, we have used popular frameworks and libraries such as RefactoringMiner [70] and Understand. For RefactoringMiner, previous studies [70,81] report that RefactoringMiner has high precision and recall scores, compared to other state-of-the-art refactoring detection tools. Similarly, readability tools have been used in previous similar study [62], and based on our own humble experience, we did not notice any anomaly while using them.

Moreover, in this study, we did not differentiate between instant and delayed migrations, by combining their results. This may not have allowed to fully understand the difference between both, especially that the instant migration is performed faster than the delayed migration, which may hypothesize that developers may have focused on the correctness of their migrated code, rather than optimizing the design of their system. This remains one of our main future experiments.

7.5.3 External validity

Threats to external validity are connected to the generalization of the obtained results. Our empirical study was limited to only open source Java projects. However, we constrained by the tools we use to collect the metrics, and besides Understand, others can only process Java source code. Thus, only the first research question can be extended across languages, if there is such a dataset because the one we have used is also limited to Java libraries.
Conclusion and Future Work

In this chapter, we conducted a large scale empirical study to investigate the impact of software migration between third-party libraries on code quality and comprehension. Our qualitative and empirical analysis indicate that library migrations have a positive impact on software’s design, in terms of coupling and cohesion. We also experiment their effect on two state-of-the-art code readability metrics, and we observe an improvement in both metrics. We observed multiple factors that explain the improvement, including the typical better naming conventions and more cohesive API methods. We also noticed a particular refactoring activity that aims to facilitate the migration by adjusting the existing code elements to match the signature of the added method(s). This explains the high rate of type change migrations, being performed simultaneously with the addition of new methods, along with various rename refactorings to bridge the lexical gap between the existing codebase and the introduced API. Finally, we leverage structural and readability metrics to define a ranking score for migration examples. To evaluate the effectiveness of our ranking, we surveyed developers to see whether our top recommended examples would match what developers consider to be the best choice. Results show that our top-1 recommended example achieves an agreement of 59%.

These factors drive our future work. We plan on further leveraging API contextual information to recommend better APIs for usage, with respect to a given code fragment. We also plan on extending the structural metrics used to characterize software design quality, such as including the weighted method per class, response for a class, class stability, and depth of inheritance tree.
Part II

API Recommendation
Chapter 8

Learning to Recommend Third-Party Library Migration

8.1 Abstract

The manual migration between different third-party libraries represents a challenge for software developers. Developers typically need to explore both libraries Application Programming Interfaces, along with reading their documentation, in order to locate the suitable mappings between replacing and replaced methods. In this chapter, we introduce RAPIM, a machine learning model that recommends mappings between methods from two different libraries. Our model learns from previous migrations, manually performed in mined software systems, and extracts a set of features related to the similarity between method signatures and method textual documentations. We evaluate our model using 8 popular migrations, collected from 57,447 open-source Java projects. Results show that RAPIM is able to recommend relevant library API mappings with an average accuracy score of 87%. Finally, we provide the community with an API recommendation web service that could be used to support the migration process.

8.2 Introduction

Modern software systems rely heavily on third-party libraries as a means to save time, reduce implementation costs, and increase software quality while offering rich, robust, and up-to-date fea-
However, as software systems evolve rapidly, there is a need for appropriate tools, reliable, and efficient techniques to provide developers with support for decision making when replacing their old and obsolete libraries with up-to-date ones. This process of replacing a library with a different one, while preserving the same code behavior, is known as library migration [76,77].

The migration process between libraries is widely acknowledged to be a hard, error-prone, and time-consuming process [5,9,20,45]. Hence, developers have to explore the new library’s API and its associated documentation in order to locate the right API method(s) to replace in the current implementation that belongs to the retired library’s API. Developers need often to spend significant time to verify that the newly adopted features do not introduce any regression. For instance, previous works have shown that developers typically spend up to 42 days to migrate between libraries [8].

A number of migration approaches and techniques have been proposed recently with the aim of identifying what the replacements of a deprecated API are with a newer version of the same API [21,42,68,85]. Other studies recommend which library to adopt, when, retiring another one [33,39,52,59,93]. However, such approaches do not provide guidance to software developers on how to concretely perform a fine-grained migration at the method-level. Indeed, method-level recommendations have been the focus of many studies, but, only for recommending the same library, across different programming languages or operating systems [32,60,61]. Obviously, there is a need for a more comprehensive recommendation technique that is both library and language independent i.e., it takes as input two different libraries and generates mappings on how to replace one with another at the method level.

In this chapter, we design a learning model, labeled as RAPIM (Recommending API Migrations), that leverages previously performed migration changes by developers and recommends API-level migrations for similar migration contexts. RAPIM takes as input two different libraries and identifies as output potential mappings between their API methods. The basic idea behind RAPIM is to reuse and take advantage of the valuable migration knowledge available in previous manually performed migrations by developers in a different open-source project, i.e., learn from the “wisdom of the crowd”. RAPIM uses predefined features related to the similarity of method signatures and their corresponding API documentation to build its model. The model treats the matching game between two API methods as a classification problem: for each method from the retired API, RAPIM recommends the most relevant method from the new API, based on how close they are from a lexical and descriptive standpoint.

The key findings of our experiments show that RAPIM performs significantly better than state-
of-the-art techniques. On average, RAPIM’s accuracy was 86.97%. We also challenge the stability of RAPIM with respect to the training size, i.e., we illustrate that the used dataset is sufficient to generalize the model and deploy it. Finally, we supply the community with an open source API recommendation tool that is deployed as a web API.

To summarize, this study makes the following contributions:

1. We propose, RAPIM, an automated approach for library APIs migration that takes as input, two different third-party libraries along with their APIs and documentation and recommends existing mappings between their API methods. RAPIM learns from existing library migration changes manually performed by developers in different open-source projects, then builds a model using various features related to method signatures and method documentation in order to recommend mappings between methods in similar contexts.

2. We conduct an empirical study to evaluate RAPIM’s performance in detecting mappings for 8 popular migrations, along with comparing it to adapted state-of-the-art migration techniques. Findings show that RAPIM effectively generates correct mappings while improving the state-of-the-art results by 39.51% in terms of accuracy.

3. We implement RAPIM and deploy it as a lightweight Web service that is publicly available for software engineers and practitioners to support them in any migration process. We also publicly issue RAPIM’s dataset online for replication and extension purposes.\(^1\)

\(^1\)http://migrationlab.net/index.php?cf=asc2019
CHAPTER 8. LEARNING TO RECOMMEND THIRD-PARTY LIBRARY MIGRATION

Figure 8.1: The proposed RAPIM approach for method APIs mapping recommendation.
8.3 Methodology

In this section, we initially give an overview of our approach. Then, we detail the different steps and features needed to design our model.

A migration rule is denoted by a pair of a source (removed) library $L_s$ and a target (added) library $L_t$, and represented by $L_s \rightarrow L_t$. For example, easymock $\rightarrow$ mockito represents a Migration Rule where the library easymock\(^2\) is migrated to the new library mockito\(^3\). For a given migration rule $L_s \rightarrow L_t$, let $L_s = m_s^{(i)}$ denote a set of methods that belong to $L_s$, where $m_s^{(i)} = \{m_1, m_2, ..., m_{L_s}\}$, and $L_t = m_t^{(i)}$ denotes a set of methods that belong to $L_t$, where $m_t^{(i)} = \{m_1, m_2, ..., m_{L_t}\}$. Our goal is to find an alignment between both $L_s$ and $L_t$.

$$f : L_s \rightarrow L_t \quad (8.1)$$

in such a way that each source method $m_s^{(i)} \in L_s$ is mapped to an equivalent target method $m_t^{(i)} \in L_t$, this process is called Method Mapping.

Figure 8.1 outlines an overview of RAPIM approach which consists of two main phases: the first phase, called (1) Collection Phase, collects the necessary information, e.g., library documentation, for all the mappings contained in the data set \cite{5}, to generate the features. This phase starts with (A) the collection of APIs and their corresponding documentation; (B) text preprocessing, and (C) the feature engineering that used to extract the feature from method signature and API documentation. The second phase, called (2) Recommendation Phase, starts with (D) the selection of relevant features, before (E) passing them to the learner. The learner generates RAPIM model that used to recommend relevant library API mappings between two libraries. In the following, we detail RAPIM’s five main processes.

8.3.1 Data Collection

This phase takes two inputs. The first input, method mappings, consists of a manually inspected dataset of valid and invalid method mappings for different migration rules from a study by Alrubaye et al. \cite{5}. For example, in Figure 8.1, for a given migration rule easymock $\rightarrow$ mockito, we identify one of the valid method mappings between the two following methods $\text{createMock}(\text{String name}, \text{MockType type}) \rightarrow \text{mock}(T \text{ classToMock})$.

\(^2\)http://easymock.org
\(^3\)https://site.mockito.org
The second input of this phase is the API documentation, which is represented by the *Documentation Collector*\(^4\) in Figure 8.1. For a given method mapping, the *Documentation Collector* collects the API documentation for both the source method and the target method. Based on a migration rule, it automatically downloads the library documentation as a *jar* file for all library releases involved in migrations. Our approach relies on the libraries documentation on Maven Central Repository\(^5\). The *Documentation Collector* then converts the API documentation from a *jar* file to multiple *HTML* source files using the *doclet API*\(^6\). It parses all of the *HTML* files and collects the documentation related to class descriptions, method descriptions, parameter descriptions, return descriptions, package names, and class names. The *Documentation Collector* identifies the documentation associated with every method mapping. The collection process ends when all the information associated with every method involved in all method mapping in the dataset are collected.

### 8.3.2 Text Engineering

Our approach aims at automatically recommending API method mappings to support developers in their library migration tasks. The migration task involves typically the analysis of structured and unstructured data sources, including method signatures, textual API descriptions, code snippet examples from open-source repositories, etc. To automatically explore such data, we deploy information retrievals (IR) techniques such as text preprocessing, vector space model, and cosine similarity to preprocess our sources.

**Information Extraction (IE)**

Let \(d\) be method signature (name, class name, or package name). In this step, we extract \(d^*\) using the function named Information Extraction *IE* as follows:

\[
d^* = IE(d)
\]  

(8.2)

For example, in Figure 8.1, if \(d\) is the target API package name, then \(d^*\) is generated using *IE* which is described as follows:

\(^4\)http://migrationlab.net/tools.php?cf=asc2019&tool=DoC  
\(^5\)central.maven.org  
\(^6\)https://goo.gl/S3xRwk
### Information Extraction (IE)

**Input**: 'com.IMockBuilder'.

1- **Special Characters Cleanup**: In this step, we replace all special characters such as dots with a space. For the given input, the output for this step is 'com < space > IMockBuilder'.

2- **Camel Case Splitter**: In this step, we split all identifiers with using camel case. The output for this step is 'com < space > I < space > Mock < space > Builder'.

**Output**: 'com I Mock Builder'

### Text Preprocessing (TPP)

$d$ may have a mix of words and special characters, such as a dot, a colon, etc. In text processing, **TPP**, we clean the documents of special characters and common English words, such as "the" and "is". We then apply a stemming transformation to all extracted words to put them in their root format using Natural Language Processing\(^7\) (NLP). This process helps to reduce the noise when calculating the similarity between two documents.

\[
\hat{d} = TPP(d) \quad (8.3)
\]

For example, in Figure 8.1, if $d$ is a source method description, then $\hat{d}$ is generated, using **TPP**. The **TPP** process is described as follows:

\(^{7}\)nlp.stanford.edu/software/corenlp.shtm
Text Preprocessing (TPP)

**input** \(d\): ‘Create a named mock of the request type from this builder. The same builder can be called to create multiple mocks.’

1- **Tokenization:** In this step, we convert text words into a list of tokens so that we can process each token alone.

\[
\text{['Create', 'a', 'named', 'mock', 'of', 'the', 'request', 'type', 'from', 'this', 'builder', 'can', 'be', 'called', 'to', 'create', 'multiple', 'mocks']}\]

2- **Unnecessary punctuation removal:** This is the process of removing unnecessary punctuation, tags such as ‘.’ from a list of tokens.

\[
\text{['Create', 'a', 'named', 'mock', 'of', 'the', 'request', 'type', 'from', 'this', 'builder', 'The', 'same', 'builder', 'can', 'be', 'called', 'to', 'create', 'multiple', 'mocks']}
\]

3- **Stop and reserved words removal:** In this step, we remove all English words and reserved words\(^a\) such as "a", "of", "the", "from", "this", "can", "be", "to".

\[
\text{['Create', 'named', 'mock', 'request', 'type', 'builder', 'builder', 'called', 'create', 'multiple', 'mocks']}
\]

4- **Lemmatization:** is the process of reducing words to the root. This helps to remove inflection and reduce inflectional forms. For example called, calling, call’s, ⇒ call, mocks, ⇒ mock.

\[
\text{['Create', 'name', 'mock', 'request', 'type', 'builder', 'builder', 'create', 'multiple', 'mock']}
\]

5- **Output** \(\hat{d}\): This last step is to convert all characters to lowercase and combine all tokens into one string.

\['create mock request type builder builder create multiple mock'\]

---

\(^a\)http://www.textfixer.com/resources/common-english-words.tx

Vector Space Representation

As part of generating the features, we calculate the similarity between the source method documentation \(s\) and the target method documentation \(t\). This includes the similarity between each method description, method name, or method return type description of \(s\) and \(t\). To calculate the similarity between two textual documents, we first need to convert the text to a numeric vector and then calculate their closeness using cosine similarity. To convert the text into a numeric vector, we use the Term Frequency-Inverse Document Frequency (TF-IDF) technique. For a given document, the weight vector \(W_d\) represents an array of frequency weights for each term in the document. The weight for each term \(w_{t,d}\) is based on the classic \(tf \times idf\) weighting, as shown in equation 8.4 where \(tf_{t,d}\) is the number of times a term \(t\) appears in a document and \(t_n\) is the number of terms in the document. While \(N\) is the number of documents. In our case, \(N = 2\) since we are performing binary comparisons(source and target method). \(df_t\) is the number of documents in which the term
$t$ has appeared. In our case, it has the value of 1 if it appears in one document or 2 if it appears in both documents.

$$w_{d,t} = \begin{bmatrix} w_{t,1} \\ w_{t,2} \\ \vdots \\ w_{t,n} \end{bmatrix}, \quad w_{t,d} = \frac{tf_{t,d}}{tn} * \log \left( \frac{N}{df_t} \right)$$ \hspace{1cm} (8.4)

Then, we use cosine similarity $sim(s, t)$ to measure how similar two vectors are, based on the dot product of their magnitude [71]. For a given source weight vector $W_s$, and a target weight vector $W_t$, we calculate $sim(s, t)$ between the two vectors using the equation 8.5 which outputs a value between [0-1], where 0 means the two documents are completely distinct, and 1 means both documents are identical. The higher the $sim(s, t)$ is, the closer the two documents are.

$$sim(s, t) = cos(s, t) = \frac{W_s \cdot W_t}{||W_s|| \cdot ||W_t||}$$ \hspace{1cm} (8.5)

### 8.3.3 Feature Engineering

We extract numeric features from the source and target method information that we think may help the machine learning model to recommend more accurate results. Initially we extract nine different features $\phi_1(s, t)$ to $\phi_9(s, t)$ from $s$ and $t$ method information, and one binary class $Output$ which is either valid or invalid and predefined in the dataset. Every feature is calculated between every method from the source library $L_s$, with every method from the target library $L_t$.

**Method Description $\phi_1$**

we extract $\phi_1(s, t)$, by calculating the cosine similarity between the source method description $md_s$, and the target method description $md_t$. We have decided not to apply text preprocessing $TPP$ on the methods’ description because it could have code examples that will be cleaned if we apply $TPP$ on text. We have found that keeping these code examples increases the accuracy by 3% as opposed to removing them using the $TPP$ process.
\[ \varphi_1(s, t) = \text{sim}(md_s, md_t) \] (8.6)

For instance, to calculate \( \varphi_1(s, t) \) from the example in Figure 8.1, we calculate the cosine similarity between \( md_s \) ("Create a named mock of the request type from this builder. The same builder can be called to create multiple mocks."), and \( md_t \) ("Creates mock object of given class or interface. See examples in Javadoc for Mockito class"). In this case, the similarity score is (0.59).

**Return Type Description \( \varphi_2 \)**

This feature is extracted by applying \( \text{TPP} \) on the source method return type description \( rtd_s \), and the target method return type description \( rtd_t \), to generate \( \hat{rtd}_s \), and \( \hat{rtd}_t \). The cosine similarity is then applied between \( \hat{rtd}_s \) and \( \hat{rtd}_t \).

\[ \varphi_2(s, t) = \text{sim}(\hat{rtd}_s, \hat{rtd}_t) \] (8.7)

For instance, to calculate \( \varphi_2(s, t) \) from the example in Figure 8.1, we apply \( \text{TPP} \) on both \( rtd_s \) ("the newly created mock") and \( rtd_t \) ("mock object ") to get \( \hat{rtd}_s \) and \( \hat{rtd}_t \). We then calculate the cosine similarity between \( \hat{rtd}_s \) and \( \hat{rtd}_t \). In this case the similarity score is (0.83).

**Input Parameters Description \( \varphi_3 \)**

This feature is extracted by applying \( \text{TPP} \) on the source method input parameters description \( ipd_s \) and the target method input parameters description \( ipd_t \) to generate \( \hat{ipd}_s \) and \( \hat{ipd}_t \). We then apply the cosine similarity between \( \hat{ipd}_s \) and \( \hat{ipd}_t \).

\[ \varphi_3(s, t) = \text{sim}(\hat{ipd}_s, \hat{ipd}_t) \] (8.8)

For instance, to calculate \( \varphi_3(s, t) \) from the example in Figure 8.1, we apply \( \text{TPP} \) on both \( ipd_s \) ("name -the mock name | type - the mock type"), and \( ipd_t \) ("classToMock - class or interface to mock") to get \( \hat{ipd}_s \), and \( \hat{ipd}_t \), then We calculate the cosine similarity between \( \hat{ipd}_s \) and \( \hat{ipd}_t \). In this case the similarity score is (0.79).
CHAPTER 8. LEARNING TO RECOMMEND THIRD-PARTY LIBRARY MIGRATION

Input Parameters Signature $\varphi_4$

This feature is extracted by applying $IE$ on source method input parameters signature $ips_s$, and target method input parameters signature $ips_t$ that generate $ips^*_s$, and $ips^*_t$. Then apply the cosine similarity between $ips^*_s$, and $ips^*_t$.

$$\varphi_4(s, t) = \text{sim}(ips^*_s, ips^*_t)$$ (8.9)

For instance, to calculate $\varphi_4(s, t)$ from the example in Figure 8.1, we apply $IE$ on both $ips_s$ ("String name, MockType type"), and $ips_t$ ("T classToMock") to get $ips^*_s$, and $ips^*_t$, then we calculate the cosine similarity between $ips^*_s$, and $ips^*_t$. In this case the similarity score is (0.73).

Return Type Signature $\varphi_5$

This feature is extracted by comparing source method return type signature $rts_s$, and target method return type signature $rts_t$, if they have same return type, we return one otherwise we return zero.

$$\varphi_5(s, t) = \begin{cases} 
1 & \text{if } rts_s \text{ is equal to } rts_t \\
0 & \text{if } rts_s \text{ is not equal to } rts_t 
\end{cases}$$ (8.10)

For instance, to calculate $\varphi_5(s, t)$ for example in Figure 8.1, both $rts_s$, and $rts_t$ return generic which is $T$, in this case the result for this matrix will be one (1).

Method Name $\varphi_6$

This feature is extracted by applying $IE$ on source method name $methodName_s$, and target method name $methodName_t$ that generate $methodName^*_s$, and $methodName^*_t$. Then apply the cosine similarity between $methodName^*_s$, and $methodName^*_t$.

$$\varphi_6(s, t) = \text{sim}(methodName^*_s, methodName^*_t)$$ (8.11)

For instance, to calculate $\varphi_6(s, t)$ from the example in Figure 8.1, we apply $IE$ on both $methodName_s$ ("createMock"), and $methodName_t$ ("mock") to get $methodName^*_s$, and $methodName^*_t$, then we calculate the cosine similarity between $methodName^*_s$, and $methodName^*_t$. In this case the similarity score is (0.79).
CHAPTER 8. LEARNING TO RECOMMEND THIRD-PARTY LIBRARY MIGRATION

Number of Input Parameters $\varphi_7$

This feature is extracted by calculating the ratio between number of input parameters in source method $\text{inputParamCount}_s$ and number of input parameters in target method $\text{inputParamCount}_t$ as shown in equation 9.8.

$$\varphi_7(s, t) = 1 - \frac{|\text{inputParamCount}_s - \text{inputParamCount}_t|}{\text{inputParamCount}_s + \text{inputParamCount}_t}$$ (8.12)

For instance, to calculate $\varphi_7(s, t)$ from the example in Figure 8.1, we find different between $\text{inputParamCount}_s$ which has two parameters which are name and type, and $\text{inputParamCount}_t$ that has one input parameters which is (classToMock), so the different is (0.6).

Package Name $\varphi_8$

This feature is extracted by applying IE on source method package name $\text{packageName}_s$, and target method package name $\text{packageName}_t$ that generate $\text{packageName}_s^*$, and $\text{packageName}_t^*$. Then apply the cosine similarity between $\text{packageName}_s^*$, and $\text{packageName}_t^*$.

$$\varphi_8(s, t) = \text{sim}(\text{packageName}_s^*, \text{packageName}_t^*)$$ (8.13)

For instance, to calculate $\varphi_8(s, t)$ from the example in Figure 8.1, we apply IE on both $\text{packageName}_s$ ("org.easymock"), and $\text{packageName}_t$ ("org.mockito") to get $\text{packageName}_s^*$, and $\text{packageName}_t^*$, then We calculate the cosine similarity between $\text{packageName}_s^*$, and $\text{packageName}_t^*$. In this case the similarity score is (0.96).

Class Name $\varphi_9$

This feature is extracted by applying IE on class name where source method lives $\text{className}_s$, and class name where target method lives $\text{className}_t$ that generate $\text{className}_s^*$, and $\text{className}_t^*$. Then apply the cosine similarity between $\text{className}_s^*$, and $\text{className}_t^*$.

$$\varphi_9(s, t) = \text{sim}(\text{className}_s^*, \text{className}_t^*)$$ (8.14)
For instance, to calculate $\varphi_9(s, t)$ from the example in Figure 8.1, we apply IE on both $className_s$ ("IMockBuilder"), and $className_t$ ("Mockito") to get $className_s^*$, and $className_t^*$, then we calculate the cosine similarity between $className_s^*$, and $className_t^*$. In this case the similarity score is (0.94).

8.3.4 Feature selection

We predefined nine features from $\varphi_1$ to $\varphi_9$, however, not all of these features might be helpful for the learner in achieving better results. We applied Filter Based Feature Selection [91] which shows how much each feature contributes to recommending the output. The filter shows us that $\varphi_9$ does not have any contribution in recommending the output class. In this case, there are two methods from two different libraries written by two different developers that could have the same class name. So, we drop this feature.

8.3.5 Classifier Model

There are a number of machine learning algorithms designed precisely for this situation. Such an algorithm takes the form of classifier which operates on instances [54]. For our purposes, an instance is a feature vector extracted between a source and a target method ( $\varphi_1$ to $\varphi_8$). In the training phase, we feed the classifier a set of instances along with labeled “output”. The label
output is binary judgment by the previous study [5] that classify the method mapping as “valid” or “invalid”. We normalize all the instances using $z$-score, to avoid over-fitting problem.

When the training is complete, the classifier model is generated and is ready to use. the classifier generates a model. We give a model an instance that has not been seen before. The model assigns the probability that it belongs to the valid or invalid method mapping class. To ensure that our model is exportable, we generate our models using MatLab R2018a with “Neural Pattern Recognition” toolkit, and we export it as a webservice using Microsoft Azure Machine learning studio.

To compare various potential classifiers that might reach our goal, we conducted an empirical study:

As shown in Figure 8.2, we compared between various state-of-the-art learners, including, neural networks, Support Vector Machines (SVM), Random Forest and Boosted Decision Trees (BDT). Our empirical study revealed that the logistic regression and neural network models achieved the worst results among tested classifiers, which can be explained by relatively small training dataset. For the neural network model classifier, we employed multi-layer perceptron architecture with one hidden layer. We tested models with various numbers of neurons (from five neurons up to 50) in the hidden layer. Our study renders that with the increase of neurons in the hidden layer, the testing accuracy decreases. This decrease in accuracy indicates model over-fitting and is explained by the lack of data. The best result that we were able to achieve using the neural network model is 85% accuracy rate. This result was produced by the model with ten neurons in the hidden layer. The neural network model with one hidden layer and 50 neurons in it achieved only 69% of the accuracy rate. The increasing complexity of the model (adding hidden layers and/or neurons) led to even bigger over-fitting. A J48 decision tree had similar results to the neural network model, with an 86% accuracy rate.

In order to improve results, we tried SVM based classifier with various kernel functions, such as the linear kernel, polynomial kernel, Gaussian kernel, and Radial Basis Function (RBF) kernel. SVM got its best 89% accuracy rate with RBF kernel. The random forest classifier gave an accuracy rate of 90%. A Two-Class Boosted Decision Tree (BDT) was the best learner for our dataset with an accuracy rate of 93%. BDTs are known for their good performance on relatively small datasets due to it use of an ensemble of Decision Trees and weighted voting. In a nutshell, BDT randomly selects samples from the dataset, and for every sample, it applies a Decision Tree (DT) to build and test the

\[8\text{https://www.mathworks.com/products/matlab.html}\]
\[9\text{migrationlab.net}\]
\[10\text{https://studio.azureml.net/}\]
learner using the remaining rest of the dataset. Then, it uses the misclassified samples as part of the training dataset used by the next learner. Afterwards, it finds the probability of the output for all learners. This improves the learner’s accuracy and reduces the over-fitting problem. After tuning, we found that 233 data-set samples are giving the minimum error as shown in Figure 8.3. The choice of Two-Class Boosted Decision Tree (BDT) was based on its high accuracy in recommending valid mappings.

8.4 Experimental Design

We design our methodology to answer the following two research questions.

- **RQ1. (Accuracy)** To what extent is RAPIM able to generate the correct method mappings? How does it perform in comparison with the state-of-the-art techniques?
  
  To answer RQ1, we perform two experiments using two different datasets: 1) we evaluate the accuracy of RAPIM in recommending correct method mappings for eight popular migrations. To ensure a fair comparison, we perform our comparative study using the same dataset [5] (i.e., input migration rules that run under the same execution environment). RAPIM and Learning-To-Rank use one binary output class and the same exact set of eight features that we have discussed previously. Since LTR and RAPIM require a supervised learning, we split our data-set into training and testing as follows: We perform a 9 cross-fold validation, where one migration rule is considered for testing, and the remaining migration rules are used for training. For instance, if slf4j→log4j is the selected rule for testing, we use all the remaining rules for training by assigning all the method mappings as input to the model. RAPIM uses the training set to learn recommendation patterns, while Learning-to-rank uses the training set to compute the weights for the features. Once the two models are trained, we switch to the testing phase by providing all the possible combinations of method mappings between slf4j→log4j, to the model in order to decide whether each combination is valid, or invalid. This process is repeated across the 9 folds. As for TMAP and MS, they only consider the input migration rule because they are deterministic algorithms (no training needed).

2) We use an existing dataset of manually curated mappings, extracted from Teyton et al. [77]. We challenge the ability of RAPIM to recommend the same mappings that have been previously performed by developers and manually verified by the authors of the dataset [77]. For both datasets, we evaluate the algorithms under comparison, in terms of precision and recall, as follows: We perform the experiment for each dataset separately. We select the map-
pings from each dataset and we unmap them to create two sets of groups, the first group contains replaced methods (belonging to the retired library), and the second group contains replacing methods (belonging to the new library). Then we use the Cartesian Product, between both groups, to generate all the possible combinations of method pairs. If \( n \) is the size of the first group, and \( m \) is the size of the second group, then there are \( n \times m \) generated pairs, among which, only a subset belongs to the dataset, and so, it represents the correct set, the remaining pairs are just then labeled not correct. We use this set of correct and incorrect mappings to challenge the ability of each algorithm in distinguishing the correct mappings from the incorrect mappings.

**Accuracy.** is the ratio of all correctly recommended method mappings divided by all of the correct and incorrect recommended mappings.

\[
\text{Accuracy} = \frac{T_p + T_n}{T_p + T_n + F_p + F_n}
\]

where \( T_p \) is the total number of valid mappings that were recommended as a valid mapping. \( F_p \) is the total number of invalid mappings that were recommended as a valid mapping. \( T_n \) is the total number of invalid mappings that were recommended as an invalid mapping. \( F_n \) is the total number of valid mappings that were recommended as an invalid mapping. The higher the **Accuracy** value, the better the recommendation.

**Error.** We use the following equation to measure the tuning error, where a lesser **Error** value will mean the results are better.

\[
\text{Error} = 1 - \text{Accuracy}
\]

- **RQ2. (Training Size)** What is the minimum training data that RAPIM needs to recommend an optimal mapping?

To answer RQ2, we combine all the mappings from all the rules and then randomly split them into 10 equal folds to mitigate the danger of over-fitting. This allows for the creation of a more diverse set of mappings in each fold. We then run the algorithm nine times. For every run, we increase the training size, decrease the testing size, and measure the **Accuracy.** We start with one fold for training and nine folds for testing. We then increase the folding size for training by one and decrease the folding size for testing by one, and so on, until we have nine folds for training and one fold for testing. The goal of answering this research question is to evaluate the impact of the training data sizes on RAPIM’s accuracy. In order to export the solution as a web-service, we need to make sure that our model has been trained on sufficient data. Therefore, we perform this experiment to verify whether our approach is a stable one when using the existing set of migrations as training.
8.4.1 State of the art approaches

In this section, we describe the implementation of three state-of-the-arts approaches that we have compared with our approach. We adopted these three state-of-the-art approaches to recommend method mapping between $L_s$, and $L_t$. For every method in $L_s$, each approach calculates the similarity score with every method in $L_t$ and returns the method that has the highest matching score at $k = 1$. We selected $k=1$ because we only recommend one target method for every source method, for all approaches.

Learning to Rank (LTR)

We adopt library recommendation as ranking problem. We use the same features that we extracted in Section 8.1, along with the dataset as training to calibrate the weights of the features. A score is given for each pair of methods, belonging to the source and target API. The scoring function is a linear combination of features, whose weights are automatically trained on based on the previous mappings. The ranking function is defined as follows:

$$LTR_{score(s,t)} = \sum_{i=1}^{8} W_{i}^{LTR} \times \phi_i(s, t)$$

(8.15)

Where each feature $\phi_i$ measures the specific relationship between the source method $s$ and the target method $t$ of first eight features that discussed in the previous section. The weight parameters $W_i^{LTR}$ are the results of training on the previously solved method mappings. So, for each source method, learning-to-rank ranks the candidate target methods that are most likely to replace it. To ensure the fairness between learning-to-rank and other algorithms under comparison, we only consider the highest ranked method (TOP1).

TMAP

The Pandita1 [60] approach ranks each method mapping based on the similarity of five features.

$$TMAP_{score(s,t)} = \sum \phi_1(\hat{s}, \hat{t}) + \phi_6(s, t) + \phi_8(s, t) + \phi_9(s, t) + \phi_{x}(s, t)$$

(8.16)
Where \( \varphi_x(s, t) \) is calculated by applying \( TPP \) on the source method class description \( cd_s \) and the target method class description \( cd_t \) that generates \( \hat{cd}_s \) and \( \hat{cd}_t \). We then apply the cosine similarity between \( \hat{cd}_s \) and \( \hat{cd}_t \). While \( \varphi_1(s, t) \) is calculated by applying \( TPP \) on the source method description \( md_s \) and the target method description \( md_t \) that generates \( \hat{md}_s \) and \( \hat{md}_t \). We then apply the cosine similarity between \( \hat{md}_s \) and \( \hat{md}_t \). Other features are generated in the same manner as the previous section.

**Method Signature (MS)**

This approach calculates the method signature similarity for each combination of methods as follows [57]:

\[
MS_{\text{score}(s, t)} = 0.25 \times sm(rts_s, rts_t) + 0.25 \times lcs(ips_s, ips_t) + 0.5 \times lcs(methodName_s, methodName_t) \tag{8.17}
\]

where \( sm() \) calculates the token-level similarity [42] between the two return types and \( lcs() \) computes the longest common sub-sequence between the two given input method names [37].
8.4.2 Parameter Tuning

Parameter tuning significantly impacts the performance of the learner for a particular problem [11]. For this reason, we tune the learner in order to improve the accuracy. Since our learner is a Two-Class Boosted Decision Tree (BDT), we start our tuning using the following default inputs: Maximum Number of leaves = 20, Minimum leaf instances = 10, Learning rate = 0.2, and Number of trees = 100. We then iteratively tune the learner until we get a minimum error that cannot be improved upon. Figure 8.3 shows how the error decreased from 15% to 0.5% after we tuned the Decision Tree inputs. We can see that having the number of trees to 233 has stabilized the error rate at 0.5%. We have concluded that the best values for the learner input parameters are: Number of leaves = 6, Minimum leaf instances = 47, Learning rate = 0.14, and Number of trees = 233.

Figure 8.4 illustrates the comparison of learner recommendations with and without tuning. We see that, with turning the learner is farther from the curve and the accuracy is improved by 3%.

The features weights of LTR also needs to be calculated. The LTR parameters’ weight is trained on all of the training set except the given migration rule data. The average parameters’ weights $W_{LTR}^i$ are the following: $W_{1}^{LTR} = 0.41$, $W_{2}^{LTR} = 0.10$, $W_{3}^{LTR} = 0.17$, $W_{4}^{LTR} = 0.39$, $W_{5}^{LTR} = 0.49$, $W_{6}^{LTR} = -0.11$, $W_{7}^{LTR} = 0.37$, and $W_{8}^{LTR} = -0.00058$. 
8.5 Results

Figure 8.5: Performance of approaches under test, in terms of accuracy, across 8 migrations.

8.5.1 Results for RQ1.

We calculated the accuracy of the mappings that are generated by RAPIM, in addition to other state-of-the-art approaches: LTR, TMAP [60], and MS [57].

Figure 8.5 illustrates the accuracy of the four approaches for eight migration rules. RAPIM has the highest accuracy across all of the rules and it only varies from 80% to 98% on average. We observed that the accuracy score achieved by RAPIM is significantly higher than the three other approaches by 39.51%.

To illustrate how different approaches result in a different levels of accuracy, we qualitatively ana-
lyze the results, and we have extracted the following example in Figure 8.6, which was performed during the migration between \textit{json} and \textit{gson}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure8_6.png}
\caption{Samples of method mappings between \textit{json} and \textit{gson}.}
\end{figure}

In Figure 8.6 (A), for a given source method \textit{"String toJSONString()"}, all four approaches were able to recommend the correct target method \textit{"String toString()"}. \textit{MS} recommends the correct target method because the return type and the input parameters for both methods are the same. Also, the method names are very similar. \textit{TMAP} recommends the correct target method because

\footnote{\url{http://migrationlab.net/redirect.php?cf=asc2019&p=1}}
both methods have a similar description $\varphi_1$, and name $\varphi_6$. LTR recommends the correct method because both methods have a similar description $\varphi_1$, input parameter signature $\varphi_4$, and return type $\varphi_5$. These three features also have high weights when compared to other features, which increases the accuracy of the ranking algorithm.

In Figure 8.6 (B), for a given source method "JSONObject put(String key, int value)" only RAPIM was able to recommend the correct target method "void addProperty(String property, Number value)". LTR recommends "void addProperty(String property, String value)" as the target method instead of "void addProperty(String property, Number value)". The reason that LTR recommends the wrong method is because the input parameter for the recommended method "String value" has a higher similarity to the source method for $\varphi_3$, and $\varphi_4$ than the similarity of "Number value" to the correct target method, while other features have the same values for both target methods. So, this is due to the polymorphic nature of the method. So LTR did recommend the right method name, but not the one with the right types of input parameters. TMAP recommends "JsonElement parse(JsonReader json)" as the target method because $\varphi_1$, and $\varphi_9$ have a higher similarity to the recommended target method and source method than the correct target method. MS recommends "JsonElement get(String memberName)" because it has a higher signature similarity score to the source method "put" than the correct target method has to the source method. In both cases RAPIM recommends the correct mapping because it has learned to detect these types of patterns through its various generated decisions tress.

Through our manual analysis of the results, we notice that, all approaches are generally challenged in the following contexts: Method Overloading. It refers to two methods with same name, but with different number of parameters, type of parameters, or the order of the parameters. Polymorphic Methods. They are overridden in a class hierarchy where the subclass method has the same name and number of parameters of the base class method, but with different types. Generic methods Methods including type parameters for both the returned data and the data that is passed to the method. This allows for the method to operate on objects of various types. There are also harder cases when source and target methods differ in name, return types and even input parameters. Finally, there are also few methods without proper documentation, which also can be a challenge mainly for TMAP, LTR and RAPIM.

To further evaluate the correctness of our recommendation, we challenge it using an existing dataset provided by Teyton et al. [77]. This dataset contains 4 Migration Rules and their corresponding method mappings, detected across 16 projects, from which only 7 projects were using Maven, and so compatible with our tool. To challenge the ability of RAPIM to recommend all the mappings,
we consider these 7 projects (currently containing 3 migrations). Then, we compare our findings with the results of their manual detection to calculate the precision and recall.

Table 8.1: Performance of RAPIM, in terms of Precision and Recall, across 3 migrations.

<table>
<thead>
<tr>
<th>Migration Rule</th>
<th># mappings</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons − lang → guava</td>
<td>40</td>
<td>100%</td>
<td>77%</td>
<td>87%</td>
</tr>
<tr>
<td>org.json → gson</td>
<td>28</td>
<td>100%</td>
<td>85%</td>
<td>92%</td>
</tr>
<tr>
<td>jmock → mockito</td>
<td>18</td>
<td>100%</td>
<td>89%</td>
<td>94%</td>
</tr>
<tr>
<td>Average</td>
<td>28</td>
<td>100%</td>
<td>83%</td>
<td>91%</td>
</tr>
</tbody>
</table>

As shown in Table 8.1, RAPIM’s recommended mappings did belong to the dataset of manually identified mappings, achieving precision of 100%, respectively for (commons − lang → guava), (org.json → gson), and (jmock → mockito). As for the recall, RAPIM has missed few mappings, by annotating them to 0 instead. Missing 9 mappings made a recall of 77% for (commons − io → guava), for (org.json → gson) RAPIM missed 4 mappings, and so the recall was 85%, as for (jmock → mockito) missing 2 mappings made a recall of 89%. As we look closer to the reasons behind such performance, we notice that our approach was unable to download the documentation associated with the commons-io’s version, used in existing dataset. Therefore, our approach was deprived from a subset of its documentation features. Also, some of the missed mappings were challenging to verify, even manually. For instance, (org.mockito.Mockito.verify(T) : T → org.jmock.Mockery.assertIsSatisfied() : void) was a mapping that belong to the dataset, and was missed by our approach. As we analyze this mapping, we notice that both functions are different in their return types, their method names, their parameter numbers and types. Since RAPIM is trained in detecting methods exhibiting similarities in their function signatures, this mapping will not be recommended by RAPIM, regardless of whether it is correct or not. For instance, RAPIM has recommended the following mapping for the first method (org.mockito.Mockito.verify(T) : T → org.jmock.Mock.verify() : void), and the recommendation was based on the method name similarity. Overall, RAPIM’s recommendations are considered safe since they did not introduce any errors (wrong mappings), and only recommending mappings between methods that exhibit a minimum number of similar characteristics.
Summary for RQ1. The qualitative analysis of 8 migration rules has demonstrated that RAPIM’s accuracy has an average of 86.97%, while, the maximum accuracy scored by the other approaches is 47.46%. Thus, RAPIM has increased the accuracy of the state-of-art approaches by 39.51%. Moreover, RAPIM has given satisfactory results when tested using an existing mappings dataset, achieving an average F-score of 91%.

8.5.2 Results for RQ2.

Figure 8.7 shows the performance of RAPIM, in terms of Accuracy, as a function of the number of folds used for training. We observe that by increasing the training size, the accuracy has slightly increased from 83.3% (when trained using one fold) all the way to 92% (when trained with nine folds). We statistically tested the significance of the difference in values by applying the Mann-Whitney U Test and we found no significant difference between the result of training on fold2 and the result of training on all remaining folds.

We confirm that 10% from the training set is sufficient to recommend a method mapping with an accuracy of 83.3% as shown in Figure 8.7. Also, 30% from the data-set, used as training, was enough to recommend a method mapping with an average accuracy of 86.90%. This argues that the extracted features are independent and that using only a subset of the training set is enough for RAPIM to achieve acceptable accuracy.

Summary for RQ2. RAPIM achieves near optimal accuracy using only a subset of the training set. Thus, using the whole training set raises our confidence that our model, being exported as web-service for practitioners to use, will achieve satisfactory results.

8.5.3 Discussion and limitations

In our study assumes that libraries typically contain sufficient documentation to describe the behavior of each method offered by their APIs. However, this is not necessarily the case for all libraries. For instance, we investigated the existence of documentation in the libraries used in our study. We were able to collect library documentation for 78.4% of the methods, while we could not locate any textual descriptions for 21.6% of the remaining methods. Without the acquisition of method’s corresponding documentation, our model cannot calculate features (Q1, Q2, Q3). To test how our model performs without library documentation, we rebuilt our model without requiring library documentation features (we exclude Q1, Q2, Q3), and we label it RAPIM–. We rerun the
same experiments of RAPIM, for RAPIM−. Our key findings show that RAPIM−’s average accuracy per-migration rule is 77.53%, while RAPIM’s accuracy was 84.24% on average. This shows the importance of using documentation as part of the recommendation process, however, RAPIM− performance do still outperform state of the art techniques, and it is only needed when libraries do not contain sufficient documentation, which is not often as libraries typically support their users if they want to be competitive.

Since the detection of libraries dependencies is based on Maven, our recommendation is exclusive for Java, which represents one of many languages that also heavily rely on libraries and web services. However, our RAPIM extracts its features from method signatures and their documentation, so it can be applied to any other language, if we update the data collector module.

Our approach is entirely static, it does not include any behavioral features. Testing can be used to support the recommendation, if we can leverage existing test cases to verify if any module impacted by the migration, does not exhibit any unexpected behavior. However, such setting requires that all mined software projects must contain test suits with sufficient coverage for the migrated methods. Furthermore, running such programs may require human intervention, which hinders
the automation of our process. Also, the selection of the appropriate test cases to verify the behavior of impacted modules is not straightforward. Change Impact Analysis (CIA) algorithms can be useful to shortlist all code elements, impacted by the migration changes, and so in need of verification. RAPIM does not take into account the body of the method as part of the recommendation. Although comparing method bodies, in terms of their similarity can be employed using clone detection techniques for type 4 clones (behavior), not all method bodies are publicly available via their APIs. Several Java libraries are not open source and only their JAR files are available. So, exploring the similarity of method bodies will be only useful for open source libraries.

Another interesting direction to extend RAPIM, is to consider applying it into the service ecosystem. Service operators provide various service matching strategies to support users with integrating newer services into their existing frameworks. However, inaccurate service matching will not only decrease service utilization rate, but also will drastically reduce the satisfaction of users. In this context, we plan on challenging RAPIM in providing good service matching, which is able to meet the users’ needs, and competitively perform in comparison with state of the art matching techniques.

8.6 Threats to validity

We report, in this section, any potential factors that threaten the validity of our analysis.

8.6.1 Internal validity

Threats to construct validity describe concerns about the relationship between theory and observation and, generally, this type of threat is mainly constituted by any errors related to measurements. For calculating the features, running the experiments, we have used popular frameworks and libraries such as Microsoft AI [25], and NLTK [48].

For the comparative study, we have implemented LTR and TAMP, and this is another threat to validity. We mitigated this threat by verifying that our findings match the results of the previous papers. For instance, LTR’s accuracy@K=1 varies between 10% to 45%, while in our study, LTR’s accuracy@K=1 is 47%.
8.6.2 Construct validity

Another threat to internal validity of our work, is the absence of API documentation for some APIs. Since we do extract features from the library documentation, its absence does impact the performance of RAPIM. Thus, We have initially performed our experiments with well-known libraries, and therefore, their documentations is typically available online for all methods. This may not be applicable to all libraries.

8.6.3 External validity

Threats to external validity are connected to the generalization of the obtained results. All our tested libraries were Java libraries, belonging to Maven, and so they follow the Object-Oriented principles and Maven naming and documentation conventions, and this may represent a threat to our classification since it heavily depends on textual similarities. Also, we should report that not all methods were documented, and this may further impact the performance of some of our features, but these instances were very limited. Since our findings show that our approach did achieve good results across various libraries, written by different developers, even with a few sample of training.

8.7 Conclusion

This study addressed the challenge of recommending method mapping when migrating between third-party libraries. We have described a novel approach that recommends method mappings between two unknown libraries using features extracted from the lexical similarity between method names and from the textual similarity from method documentations. We evaluated RAPIM by conducting a comparative study with three states of art algorithms for method similarities, namely TMAP [60], Nguyen [57], and LTR [90]. We find that our approach outperforms all existing state of the art approaches, across 8 popular migrations. The qualitative and comparative analysis of our experiments indicate that our approach significantly increases the accuracy of the recommended mappings by an average accuracy of 39.51%, in comparison with existing state-of-the-art studies.

As part of our future investigations, we plan extending the number of migrations used, along with comparing against a larger set of binary classifiers. We plan on increasing the feature space by including the usage context for methods, in the code. Moreover, We plan on applying RAPIM in the context of Micro-services, since they also rely on the use of public APIs, and the tremendous
growth of their numbers have been raising a challenge for practitioners with regard to switching to better services, at the expense of not altering the behavior of their existing systems.

If we want to execute our recommended migrations, we need to understand the type of changes developers typically perform, to make their code compliant with the new library’s methods. Since there are identified projects with previously performed migrations, we can analyse code fragments, containing mapped methods, before and after the migration, to extract code changes surrounding the migration. Such change patterns need to be studied and replicated in similar scenarios, so that our recommendation does not only recommend the mappings, but also its corresponding code changes to accommodate the newly introduced methods.

Finally, RAPIM operates only with Java based libraries, while many developers are also facing challenges migrating between libraries across languages, and this represents one of the important future directions of our work. Many studies have previously explored migrating across languages, for instance, Dehkharghani and Shamsfard [23,24] used ontologies to support migrating between two natural languages. Pandita et al. [61] also studied the migration of same libraries between Java and C#Sharp. It would be interesting to complement this existing effort, and explore the use of ontologies and domain knowledge to extend our approach and to make it language-aware.
Chapter 9

Recommendation Using Domain And Source Code Knowledge

9.1 Abstract

The manual migration between different third-party libraries represents a challenge for software developers. Developers typically need to explore both libraries Application Programming Interfaces, along with reading their documentation, in order to locate the suitable mappings between replacing and replaced methods. In this chapter, we introduce DSKR, a machine learning model that recommends mappings between methods from two different libraries using domain and source code knowledge. Our model learns from previous migrations, manually performed in mined software systems, and extracts a set of features from domain knowledge (method signatures, method textual documentations) and source code knowledge (source code implementation). We evaluate our model using 8 popular migrations, collected from 57,447 open-source Java projects. Results show that DSKR is able to recommend relevant library API mappings on real-world known migration with an average accuracy score of 84.4%. Finally, we provide the community with an API recommendation web service that could be used to support the migration process.
9.2 Introduction and Motivation

Modern software systems rely heavily on third-party libraries as a means to save time, reduce implementation costs, and increase software quality while offering rich, robust, and up-to-date features [9,20]. However, as software systems evolve rapidly, there is a need for appropriate tools, reliable, and efficient techniques to provide developers with support for decision making when replacing their old and obsolete libraries with up-to-date ones. This process of replacing a library with a different one, while preserving the same code behavior, is known as library migration [76,77].

The migration process between libraries is widely acknowledged to be a hard, error-prone, and time-consuming process [5,9,20,45]. Hence, developers have to explore the new library’s API and its associated documentation in order to locate the right API method(s) to replace in the current implementation that belongs to the retired library’s API. Developers need often to spend significant time to verify that the newly adopted features do not introduce any regression.

Software maintenance activities consume up to 70% of the total life-cycle cost of a typical software product [13]. Previous works have shown that developers typically spend up to 42 days to migrate between libraries [8]. In the same context, another study shows how the task of library migration is typically given to developers with relatively higher years of coding experience, to reduce the possibility of introducing any regression [6].

A number of migration approaches and techniques have been proposed recently with the aim of identifying what the replacements of a deprecated API are with a newer version of the same API [21,42,68,85]. Other studies recommend which library to adopt, when, retiring another one [33,39,52,59,93]. However, such approaches do not provide guidance to software developers on how to concretely perform a fine-grained migration at the method-level. Indeed, method-level recommendations have been the focus of many studies, but, only for recommending the same library, across different programming languages or operating systems [32,60,61].

Furthermore, RAPIM [6] is recommendation model that recommend method-level mapping between different third-party libraries using domain knowledge that learns from similarity between method signatures, method textual documentations. There is a need for a more comprehensive recommendation technique learns from both domain knowledge (similarity between method signatures, method textual documentation) and source code knowledge (similarity between method source code implementation). Our hypothesis that If two different methods have same behaviours with different signatures, they may have similar implementation. By combining domain and source code knowledge, the new model may able to capture more migration and improve accuracy and lever-
age the issue of lack of documentation that RAPIM [6] faces.

In this chapter, we design a learning model, labeled as DSKR (Domain and Source Knowledge Recommendation), that leverages previously performed migration changes by developers and recommends API-level migrations for similar migration contexts. DSKR takes as input two different libraries and identifies as output potential mappings between their API methods. The basic idea behind DSKR is to reuse and take advantage of the valuable migration knowledge available in previous manually performed migrations by developers in a different open-source project, i.e., learn from the “wisdom of the crowd”. DSKR uses predefined features related to the similarity of method signatures, and their corresponding API documentation, and method source code implementation to build its model. The model treats the matching game between two API methods as a classification problem: for each method from the retired API, DSKR recommends the most relevant method from the new API, based on how close they are from a lexical, descriptive, and method implementation similarity.

The key findings of our experiments show that DSKR performs significantly better than state-of-the-art techniques. On average, DSKR’s accuracy was 86.97%. We also challenge the stability of DSKR with respect to the training size, i.e., we illustrate that the used dataset is sufficient to generalize the model and deploy it. Finally, we supply the community with an open source API recommendation tool that is deployed as a web API.

To summarize, this study makes the following contributions:

1. We propose, DSKR, an automated approach for library APIs migration that takes as input, two different third-party libraries along with their APIs with source code implementing and documentation and recommends existing mappings between their API methods. DSKR learns from existing library migration changes manually performed by developers in different open-source projects, then builds a model using various features related to method signatures, method documentation, and source code implementing in order to recommend mappings between methods in similar contexts.

2. We conduct an empirical study to evaluate DSKR’s performance in detecting mappings for 8 popular migrations, along with comparing it to adapted state-of-the-art migration techniques. Findings show that DSKR effectively generates correct mappings while improving the state-of-the-art by 3.84% results in terms of accuracy was were able to improve average accuracy 80.56% of RAPIM that using domain knowledge to 84.4% when we combine domain with source code knowledge.

3. We implement DSKR and deploy it as a lightweight Web service that is publicly available for
software engineers and practitioners to support them in any migration process. We also publicly issue DSKR’s dataset online for replication and extension purposes\(^1\).

The remainder of this chapter is as follows,

The chapter is structured as follows: Section 9.3 shows our experimental methodology in collecting the necessary data for the experiments that are discussed in Section 9.4. Finally, the conclusion and future work are highlighted in Section 10.5.

\(^1\)http://migrationlab.net/index.php?cf=asc2019
9.3 Methodology

In this section, we initially give an overview of our approach. Then, we detail the different steps and features needed to design our model.

A migration rule is denoted by a pair of a source (removed) library $L_s$ and a target (added) library $L_t$, and represented by $L_s \rightarrow L_t$. For example, easymock $\rightarrow$ mockito represents a Migration Rule where the library easymock\(^2\) is migrated to the new library mockito\(^3\). For a given migration rule $L_s \rightarrow L_t$, let $L_s = m_s^{(i)}$ denote a set of methods that belong to $L_s$, where $m_s^{(i)} = \{m_1, m_2, ..., m_{L_s}\}$, and $L_t = m_t^{(i)}$ denotes a set of methods that belong to $L_t$, where $m_t^{(i)} = \{m_1, m_2, ..., m_{L_t}\}$. Our goal is to find an alignment between both $L_s$ and $L_t$.

\[
f : L_s \rightarrow L_t
\]

(9.1)
in such a way that each source method $m_s^{(i)} \in L_s$ is mapped to an equivalent target method $m_t^{(i)} \in L_t$, this process is called Method Mapping.

Figure 9.1 outlines an overview of DSKR approach which consists of two main feature collection phases: the first phase, called (1) Domain knowledge, collects the necessary information from method signatures, and library documentation, for all the mappings contained in the data set [5], to generate the features. This phase starts with (A) the collection of APIs and their corresponding documentation; (B) text preprocessing, and (C) the feature engineering that used to extract the feature from method signature and API documentation. This phase generate eight features $\phi_1...\phi_8$. The second phase, called (2) Source code knowledge, This phase starts with (A) the collection of APIs and their corresponding method source code implementation for all the mappings contained in the data set [5]; (B) CNN learner where we build model that learn from J2EE interfaces implementations to recommend similarity score (feature $\phi_9$) between two methods source code, and (C) vector space representation of library APIs method source code then we apply Principal Component Analysis(PCA) to reduce number of features from 384*2 to five features only $\phi_{10}...\phi_{14}$. Then all above features besides output class passed to learned to build recommendation model.

We run our study on existing dataset [5] that consists of a manually inspected dataset of valid and invalid method mappings for different migration rules from a study by Alrubaye et al. [5]. For example, in Figure 9.1, for a given migration rule easymock $\rightarrow$ mockito, we identify one of the valid

\(^2\)http://easymock.org

\(^3\)https://site.mockito.org
method mappings between the two following methods
\( \text{createMock}(String \ name, \ MockType \ type) \), from \textit{easymock}, to the method \( \text{mock}(T \ classToMock) \), offered by \textit{mockito}.

As RAPIM [6] reported, We were able to collect library documentation for 78.4% of the methods, while we could not locate any textual descriptions for 21.6% of the remaining methods from dataset [5]. For fair comparison with RAPIM [6] that relies heavily on documentation, we used only 78.4% of the data that has documentation in this study.

9.3.1 Domain knowledge

In this section we describe how we collect and extract features \( (\phi_1...\phi_8) \) from domain knowledge (library documentation, and method signatures).

Extract Method Mapping Documentation

In this phase, we collect library documentation for given method mapping in alrubaye dataset [5]. As shown in Figure 9.1 where we collection documentation for following methods
\( \text{createMock}(String \ name, \ MockType \ type) \), from \textit{easymock}, to the method \( \text{mock}(T \ classToMock) \), offered by \textit{mockito}. Thanks for Alrubaye [6] study that offer \textit{Documentation Collector} that we used to collects the API documentation for both the source method and the target method. Based on a migration rule. The Documentation Collector identifies the documentation associated with every method mapping. The collection process ends when all the information associated with every method involved in all method mapping in the dataset are collected.

From domain knowledge, We extract numeric features from the source and target method information that we think may help the machine learning model to recommend more accurate results. Initially we extract eight different features \( \phi_1(s, t) \) to \( \phi_8(s, t) \) from \( s \) and \( t \) method information, and one binary class \textit{Output} which is either valid or invalid and predefined in the dataset. Every feature is calculated between every method from the source library \( L_s \), with every method from the target library \( L_t \).
Feature Engineering

In this section, we describe how we extract every feature from domain knowledge.

**Method Description $\varphi_1$:** we extract $\varphi_1(s, t)$, by calculating the cosine similarity between the source method description $md_s$, and the target method description $md_t$. We have decided not to apply text preprocessing $TPP$ on the methods’ description because it could have code examples that will be cleaned if we apply $TPP$ on text. We have found that keeping these code examples increases the accuracy by 3% as opposed to removing them using the $TPP$ process.

$$\varphi_1(s, t) = \text{sim}(md_s, md_t) \quad (9.2)$$

For instance, to calculate $\varphi_1(s, t)$ from the example in Figure 9.1, we calculate the cosine similarity between $md_s$ ("Create a named mock of the request type from this builder. The same builder can be called to create multiple mocks.") and $md_t$ ("Creates mock object of given class or interface. See examples in Javadoc for Mockito class"). In this case, the similarity score is (0.59).

**Return Type Description $\varphi_2$:** This feature is extracted by applying $TPP$ on the source method return type description $rtd_s$, and the target method return type description $rtd_t$ to generate $\hat{rtd}_s$ and $\hat{rtd}_t$. The cosine similarity is then applied between $rtd_s$ and $rtd_t$.

$$\varphi_2(s, t) = \text{sim}(\hat{rtd}_s, \hat{rtd}_t) \quad (9.3)$$

For instance, to calculate $\varphi_2(s, t)$ from the example in Figure 9.1, we apply $TPP$ on both $rtd_s$ ("the newly created mock") and $rtd_t$ ("mock object") to get $\hat{rtd}_s$ and $\hat{rtd}_t$. We then calculate the cosine similarity between $\hat{rtd}_s$ and $\hat{rtd}_t$. In this case the similarity score is (0.83).

**Input Parameters Description $\varphi_3$:** This feature is extracted by applying $TPP$ on the source method input parameters description $ipd_s$ and the target method input parameters description $ipd_t$ to generate $\hat{ipd}_s$ and $\hat{ipd}_t$. We then apply the cosine similarity between $ipd_s$ and $ipd_t$.

$$\varphi_3(s, t) = \text{sim}(\hat{ipd}_s, \hat{ipd}_t) \quad (9.4)$$

For instance, to calculate $\varphi_3(s, t)$ from the example in Figure 9.1, we apply $TPP$ on both $ipd_s$ ("name - the mock name | type - the mock type") and $ipd_t$ ("classToMock - class or interface to mock") to get $\hat{ipd}_s$, $\hat{ipd}_t$. 

---

<table>
<thead>
<tr>
<th>Method Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varphi_1$</td>
<td>Extracts $\varphi_1(s, t)$ by calculating the cosine similarity between $md_s$ and $md_t$.</td>
</tr>
<tr>
<td>$\varphi_2$</td>
<td>Extracts $\varphi_2(s, t)$ by applying $TPP$ on $rtd_s$ and $rtd_t$ to generate $\hat{rtd}_s$ and $\hat{rtd}_t$.</td>
</tr>
<tr>
<td>$\varphi_3$</td>
<td>Extracts $\varphi_3(s, t)$ by applying $TPP$ on $ipd_s$ and $ipd_t$ to generate $\hat{ipd}_s$ and $\hat{ipd}_t$.</td>
</tr>
</tbody>
</table>
and \( \text{id}_{t} \), then we calculate the cosine similarity between \( \text{id}_{s} \), and \( \text{id}_{t} \). In this case the similarity score is (0.79).

**Input Parameters Signature \( \varphi_4 \):** This feature is extracted by applying IE on source method input parameters signature \( \text{ips}_s \), and target method input parameters signature \( \text{ips}_t \) that generate \( \text{ips}_s^* \), and \( \text{ips}_t^* \). Then apply the cosine similarity between \( \text{ips}_s^* \), and \( \text{ips}_t^* \).

\[
\varphi_4(s, t) = \text{sim}(\text{ips}_s^*, \text{ips}_t^*) \tag{9.5}
\]

For instance, to calculate \( \varphi_4(s, t) \) from the example in Figure 9.1, we apply IE on both \( \text{ips}_s \) ("String name, MockType type"), and \( \text{ips}_t \) ("T classToMock") to get \( \text{ips}_s^* \), and \( \text{ips}_t^* \), then we calculate the cosine similarity between \( \text{ips}_s^* \), and \( \text{ips}_t^* \). In this case the similarity score is (0.73).

**Return Type Signature \( \varphi_5 \):** This feature is extracted by comparing source method return type signature \( \text{rts}_s \), and target method return type signature \( \text{rts}_t \), if they have same return type, we return one otherwise we return zero.

\[
\varphi_5(s, t) = \begin{cases} 
1 & \text{if } \text{rts}_s \text{ is equal to } \text{rts}_t \\
0 & \text{if } \text{rts}_s \text{ is not equal to } \text{rts}_t 
\end{cases} \tag{9.6}
\]

For instance, to calculate \( \varphi_5(s, t) \) for example in Figure 9.1, both \( \text{rts}_s \), and \( \text{rts}_t \) return generic which is \( T \), in this case the result for this matrix will be one (1).

**Method Name \( \varphi_6 \):** This feature is extracted by applying IE on source method name \( \text{methodName}_s \), and target method name \( \text{methodName}_t \) that generate \( \text{methodName}_s^* \), and \( \text{methodName}_t^* \). Then apply the cosine similarity between \( \text{methodName}_s^* \), and \( \text{methodName}_t^* \).

\[
\varphi_6(s, t) = \text{sim}(\text{methodName}_s^*, \text{methodName}_t^*) \tag{9.7}
\]

For instance, to calculate \( \varphi_6(s, t) \) from the example in Figure 9.1, we apply IE on both \( \text{methodName}_s \) ("createMock"), and \( \text{methodName}_t \) ("mock") to get \( \text{methodName}_s^* \), and \( \text{methodName}_t^* \), then we calculate the cosine similarity between \( \text{methodName}_s^* \), and \( \text{methodName}_t^* \). In this case the similarity score is (0.79).

**Number of Input Parameters \( \varphi_7 \):** This feature is extracted by calculating the ratio between number of input parameters in source method \( \text{inputParamCount}_s \) and number of input parameters in target
For instance, to calculate $\varphi_7(s, t)$ from the example in Figure 9.1, we find different between $inputParamCount_s$ which has two parameters which are name and type, and $inputParamCount_t$ that has one input parameter which is (classToMock), so the different is (0.6).

**Package Name $\varphi_8$:** This feature is extracted by applying IE on source method package name $packageName_s$, and target method package name $packageName_t$ that generate $packageName^*_s$, and $packageName^*_t$. Then apply the cosine similarity between $packageName^*_s$, and $packageName^*_t$.

$$
\varphi_8(s, t) = \text{sim}(packageName^*_s, packageName^*_t)
$$

For instance, to calculate $\varphi_8(s, t)$ from the example in Figure 9.1, we apply IE on both $packageName_s$ ("org.easymock"), and $packageName_t$ ("org.mockito") to get $packageName^*_s$, and $packageName^*_t$, then We calculate the cosine similarity between $packageName^*_s$, and $packageName^*_t$. In this case the similarity score is (0.96).

### 9.3.2 Source code knowledge

In this section, we describe how we collect and extract features ($\varphi_9 \ldots \varphi_{14}$) from source code knowledge (methods source code implementation).

**Extract Method Mapping Source Code**

In this phase, we collect method source code implementation for given method mapping in al-rubaye dataset [5]. As show in Figure 9.1 where we collection source code implementation for following methods

`createMock(String name, MockType type)` from easymock, to the method `mock(T classToMock)`, offered by mockito. To do so, We first download the source/target third-party libraries jar files. Then we reverse engineer jar files to source code using `cfr_0_114.jar`. Then we wrote parser to parse all classes and their associated methods. Then for give method mapping we search for source code

---

4 [https://github.com/hussien89aa/AdsVulnerability/blob/master/cfr_0_114.jar](https://github.com/hussien89aa/AdsVulnerability/blob/master/cfr_0_114.jar)
for both methods in source/target libraries. We notice that there a method is just a wrapper for other methods. These types of wrapping developers avoid breaking changes for projects that upgrade to a new version of API, and their calls for old API methods continue working. For our work, having a wrapper is not be helpful to extract similarity information from source code that is just one line call to another method. Our parser can handle these issues by un-wrapping methods that just wrapper. For example, if method $A()$ implementing has one call $B()$, we just consider code implementing for $B()$ is code implementing for method $A()$ source code. So instead return $B()$ as source code body for method $A()$, we return the code source code body implementation for $B()$ as $A()$ source code body.

**Mapping Code2Vector**

In this step we apply code2Vec [3] on method source code implementation for given method mapping in alrubaye dataset [5]. As show in Figure 9.1 where we generate code vector form following methods source code implementation $createMock(String$ $name,$ $MockType$ $type)$, from easymock, to the method $mock(T$ $classToMock)$, offered by mockito. By end this step, we have 384 features generate for ever method source code body. Since method mapping is pair of methods so we will have 384*2 features generated by this step for given method mapping.

**Principal Component Analysis**

The code2Vec [3] generates 384 features from a method source code. Since a migration rule has two methods, so we have 384*2 features generated from source/target methods’ source code implementations per method mapping. Having such a huge number of features as input to the machine learning model will make the model slow and lead to much noise that may lead to overfilling issues where the model learns from source code knowledge more than what it learns from domain knowledge. We need to represent these 768 features with a fewer number of features while preserving the same feature contribution to the model. Principal Component Analysis (PCA) is a concept in machine learning that can help us here. It is used when we have a large number of features, and we want to represent them with N Component(features) to reduce the noise and make the model run faster. We feed 768 features to PCA and set up the number of a component to five. PCA can generate five components (features) from 768 features. The new features are $\phi_{10}...\phi_{14}$.
CHAPTER 9. RECOMMENDATION USING DOMAIN AND SOURCE CODE KNOWLEDGE

Figure 9.2: CNN Learner

CNN Learner

Another feature, we are interested in is feature $\varphi_9$ which is the similarity score between code vectors of source/target methods. The goal of this step is building a model that able to predicate similarity score between two given code vectors. In this section we discuss details how we build the CNN Learner:

**Data collection:** We need to build a model that able to predicate a similarity score between two methods from source code implementation. We need to train a model on label methods that have similar implementations, and label methods have a different implementation. In Java, we have the interface concept, where different classes could implement same interface method differently, but these methods still do the same functionality. For example, Figure 9.2 shows two different implementation for `getDomain()` method in J2EE project⁵, however both methods get the domain in different way, ideally these two methods are same. To build training data set, We parse J2EE project⁶ searching for interfaces and implementation, we find 360 interfaces methods have at least

---

⁵]https://www.oracle.com/java/technologies/appmodel.html
two different implementations. We end up by 360*2 methods. To build a binary model, we also need to have a pair of methods that have a different implementation, so the model able to predicate if two methods are similar or not. We generate 360 negative pairs, as well where two methods have different implementation. Then for every pair of methods, we run code2Vec [3] to generates 384*2 code vector features. By the end of this step, we have data set that has 768 features and one label binary class mark rows as similar or not similar vectors.

**Train model:** We train our data set, which has 768 features and one label binary class on the Convolutional neural network (CNN). We feed the dataset to the CNN network. The model has an accuracy of over 90% to predicate if two methods have similar implementation or not.

**Similarity Score \( \phi_9 \):** To generate a similarity score \( \phi_9 \), we feed method mapping code vectors to the CNN model, and the model gives a score of how similar these two methods and we consider this score as feature \( \phi_9 \).

**Acknowledgement:** The work of CNN Learner have been done by Dr. Na Meng and her student Alon from Virginia Tech, We collaborate with them in this work.

### 9.3.3 Classifier Model

There are a number of machine learning algorithms designed precisely for this situation. Such an algorithm takes the form of classifier which operates on instances [54]. For our purposes, an instance is a feature vector extracted between a source and a target method ( \( \phi_1 \) to \( \phi_{14} \)). In the training phase, we feed the classifier a set of instances along with labeled “output”. The label output is binary judgment by the previous studies [5] that classify the method mapping as “valid” or “invalid”. We normalize all the instances using \( z \)-score, to avoid over-fitting problem.

When the training is complete, the classifier model is generated and is ready to use. the classifier generates a model. We give a model an instance that has not been seen before. The model assigns the probability that it belongs to the valid or invalid method mapping class. To ensure that our model is exportable, we generate our models using Azure we export webservice 7 using Microsoft Azure Machine learning studio 8.

To compare various potential classifiers that might reach our goal, we conducted an empirical study:

---

7migrationlab.net
8https://studio.azureml.net/
As shown in Figure 9.3, we compared between various state-of-the-art learners, including, neural networks, Support Vector Machines (SVM), Random Forest and Boosted Decision Trees (BDT). Our empirical study revealed that the logistic regression and neural network models achieved the worst results among tested classifiers, which can be explained by relatively small training dataset.

For the neural network model classifier, we employed multi-layer perceptron architecture with one hidden layer. We tested models with various numbers of neurons (from five neurons up to 50) in the hidden layer. Our study renders that with the increase of neurons in the hidden layer, the testing accuracy decreases. This decrease in accuracy indicates model over-fitting and is explained by the lack of data. The best result that we were able to achieve using the neural network model is 87% accuracy rate.

In order to improve results, we tried SVM based classifier with various kernel functions, such as the linear kernel, polynomial kernel, Gaussian kernel, and Radial Basis Function (RBF) kernel. SVM got its best 85.7% accuracy rate with RBF kernel.

A Two-Class Boosted Decision Tree (BDT) was the best learner for our dataset with an accuracy rate of 91.9%. BDTs are known for their good performance on relatively small datasets due to it use of an ensemble of Decision Trees and weighted voting. In a nutshell, BDT randomly selects samples from the dataset, and for every sample, it applies a Decision Tree (DT) to build and test the learner using the remaining rest of the dataset. Then, it uses the misclassified samples as part
of the training dataset used by the next learner.

### 9.4 Experimental Design

We design our methodology to answer the following two research questions.

- **RQ1. (Accuracy)** To what extent is DSKR able to generate the correct method mappings? How does it perform in comparison with the state-of-the-art RAPIM [6]?

  To answer RQ1, we evaluate the accuracy of DSKR in recommending correct method mappings for eight popular migrations. To ensure a fair comparison, we perform our comparative study using the same dataset [5] (i.e., input migration rules that run under the same execution environment). DSKR and RAPIM [6] use one binary output class and the same exact set of fourteen features that we have discussed previously. Since RAPIM [6] and DSKR require a supervised learning, we split our data-set into training and testing as follows: We perform a 9 cross-fold validation, where one migration rule is considered for testing, and the remaining migration rules are used for training. For instance, if slf4j→log4j is the selected rule for testing, we use all the remaining rules for training by assigning all the method mappings as input to the model. DSKR and RAPIM [6] uses the training set to learn recommendation patterns. Once the two models are trained, we switch to the testing phase by providing all the possible combinations of method mappings between slf4j→log4j, to the model in order to decide whether each combination is valid, or invalid. This process is repeated across the 9 folds.

  2) We use an existing dataset of manually curated mappings, extracted from Teyton et al. [77]. We challenge the ability of DSKR to recommend the same mappings that have been previously performed by developers and manually verified by the authors of the dataset [77]. We use this set of correct and incorrect mappings to challenge the ability of each algorithm in distinguishing the correct mappings from the incorrect mappings.

  **Accuracy.** is the ratio of all correctly recommended method mappings divided by all of the correct and incorrect recommended mappings.

  \[
  \text{Accuracy} = \frac{Tp + Tn}{Tp + Tn + Fp + Fn}
  \]

  where **Tp** (True positive) is the total number of valid mappings that were recommended as a valid mapping. **Fp** (False positive) is the total number of invalid mappings that were recommended as a valid mapping. **Tn** (True negative) is the total number of invalid mappings
that were recommended as an invalid mapping. \( Fn \) (False positive) is the total number of valid mappings that were recommended as an invalid mapping. The higher the \textit{Accuracy} value, the better the recommendation.

- **RQ2. (Training Size)** What is the minimum training data that DSKR needs to recommend an optimal mapping?

To answer RQ2, we combine all the mappings from all the rules and then randomly split them into 10 equal folds to mitigate the danger of over-fitting. This allows for the creation of a more diverse set of mappings in each fold. We then run the algorithm nine times. For every run, we increase the training size, decrease the testing size, and measure the \textit{Accuracy}. We start with one fold for training and nine folds for testing. We then increase the folding size for training by one and decrease the folding size for testing by one, and so on, until we have nine folds for training and one fold for testing. The goal of answering this research question is to evaluate the impact of the training data sizes on DSKR’s accuracy comparing with RAPIM [6]. In order to export the solution as a web-service, we need to make sure that our model has been trained on sufficient data, Therefore, we perform this experiment to verify whether our approach is a stable one when using the existing set of migrations as training.

### 9.4.1 Parameter Tuning

Parameter tuning significantly impacts the performance of the learner for a particular problem [11]. For this reason, we tune the learner in order to improve the accuracy. Since our learner is a \textit{Two-Class Boosted Decision Tree (BDT)}, we start our tuning using the following default inputs: \textit{Maximum Number of leaves}=2, \textit{Minimum leaf instances}=4, \textit{Learning rate}=0.06, and \textit{Number of trees}=436. We then iteratively tune the learner until we get a minimum error that cannot be improved upon. Figure 9.4 shows how the accuracy increase from 87.5% to 91.1% after we tuned the Decision Tree inputs. We can see that having the number of trees to 182 has stabilized the accuracy. We have concluded that the best values for the learner input parameters are: \textit{Number of leaves}=36, \textit{Minimum leaf instances}=7, \textit{Learning rate}=0.33, and \textit{Number of trees}=182.

Figure 9.4 illustrates the comparison of learner recommendations with and without tuning. We see that, with turning the learner is farther from the curve and the accuracy is improved by 3.6%.
CHAPTER 9. RECOMMENDATION USING DOMAIN AND SOURCE CODE KNOWLEDGE

9.5 Results

9.5.1 Results for RQ1.

We calculated the accuracy of the mappings that are generated by DSKR, in addition to other state-of-the-art approach RAPIM [6].

Figure 9.5 illustrates the accuracy of the four approaches for eight migration rules. DSKR has the highest accuracy across all of the rules and it only varies from 72% to 97.9% on average. We observed that the accuracy score achieved by DSKR is significantly higher than RAPIM [6] approach by 3.84%.

To illustrate how different approaches result in a different levels of accuracy, we qualitatively analyze the results, and we have extracted the following example\(^9\) in Figure 9.6, which was performed during the migration between `json` and `gson`.

To understand why combining domain and source code knowledge could improve the recommendations let study Figure 9.6 and how our model better recommend replacing the method `put(String key, Map value)` with `addProperty(String key, String value)` when we migrate from the `json`

\(^9\)http://migrationlab.net/redirect.php?cf=asc2019&p=1
library to gson library. Figure 9.6 shows source code implementation for `put(String key, Map value)` and `addProperty(String key, String value)`. As we can see `addProperty(String key, String value)` is just wrapper so our parser will take content of method `add(String property, JsonElement value)` as source code implementation for `addProperty(String key, String value)`. And same thing for `put(String key, Map value)` which is just wrapper for `put(String key, Object value)`. We apply code2vec on both method source code implementation that that generate predicate topics for both methods. As we can see even methods from domain knowledge has different signatures `addProperty` and `put`, while from source code implementation they both has similar implementation and model predicate the first four topics shared between two methods as `put, add, set`. So that clarify source code knowledge could give more insights of the similarities between two methods.

Through our manual analysis of the results, we notice that, all approaches are generally challenged in the following contexts: *Method Overloading*. It refers to two methods with same name, but with different number of parameters, type of parameters, or the order of the parameters. *Polymorphic Methods*. They are overridden in a class hierarchy where the subclass method has the same name and number of parameters of the base class method, but with different types. *Generic methods* Methods including type parameters for both the returned data and the data that is passed to the method. This allows for the method to operate on objects of various types. There are also harder cases when source and target methods differ in name, return types and even input parameters.
Finally, there are also few methods without proper documentation, which also can be a challenge mainly for TMAP, LTR and DSKR.

**Summary for RQ1.** The qualitative analysis of 8 migration rules has demonstrated that that DSKR’s accuracy has an average of 84.4%, while, the maximum accuracy scored by the other approaches is 80.56%. Thus, DSKR has increased the accuracy of the state-of-art approaches by 3.48%.

### 9.5.2 Results for RQ2.

Figure 9.7 shows the performance of DSKR and RAPIM [6], in terms of Accuracy, as a function of the number of folds used for training. We observe that by increasing the training size, the accuracy has slightly increased from 84.2% (when trained using one fold) all the way to 86.99% (when trained with nine folds). We statistically tested the significance of the difference in values by applying the Mann-Whitney U Test and we found no significant difference between the result of training on fold2 and the result of training on all remaining folds.

We confirm that 10% from the training set is sufficient to recommend a method mapping with an accuracy of 84.2% as shown in Figure 9.7. Also, 30% from the data-set, used as training, was
enough to recommend a method mapping with an average accuracy of 90%. This argues that the extracted features are independent and that using only a subset of the training set is enough for DSKR to achieve acceptable accuracy.

### Summary for RQ2
DSKR achieves near optimal accuracy using only a subset of the training set. Thus, using the whole training set raises our confidence that our model, being exported as web-service for practitioners to use, will achieve satisfactory results.

#### 9.5.3 Discussion and limitations

In our study assumes that libraries typically contain sufficient documentation to describe the behavior of each method offered by their APIs. However, this is not necessarily the case for all libraries. For instance, we investigated the existence of documentation in the libraries used in our study. We were able to collect library documentation for 78.4% of the methods, while we could not locate any textual descriptions for 21.6% of the remaining methods, for method source code we were able to extract almost all source codes for all methods. Without the acquisition of method’s corresponding documentation, our model cannot calculate features (Q1, Q2, Q3). To test how our model performs without library documentation, we rebuilt our model without requiring library documentation features (we exclude Q1, Q2, Q3), and we label it DSKR–. We rerun the same exper-
iments of DSKR, for DSKR-. Our key findings show that DSKR-’s average accuracy per-migration rule is 85.7%, while DSKR’s accuracy was 86.99% on average. While RAPIM [6]-’s average accuracy per-migration rule is 77.53%, while the accuracy was 87.03% on average. This shows the importance of using documentation as part of the RAPIM [6] recommendation process, is much higher than importance of documentation on DSKR recommendation process.

Since the detection of libraries dependencies is based on Maven, our recommendation is exclusive for Java, which represents one of many languages that also heavily rely on libraries and web services. However, our DSKR extracts its features from method signatures and their documentation, so it can be applied to any other language, if we update the data collector module.

Our approach is entirely static, it does not include any behavioral features. Testing can be used to support the recommendation, if we can leverage existing test cases to verify if any module impacted by the migration, does not exhibit any unexpected behavior. However, such setting requires that all mined software projects must contain test suits with sufficient coverage for the migrated methods. Furthermore, running such programs may require human intervention, which hinders the automation of our process. Also, the selection of the appropriate test cases to verify the behavior of impacted modules is not straightforward. Change Impact Analysis (CIA) algorithms can be useful to shortlist all code elements, impacted by the migration changes, and so in need of verification. DSKR does not take into account the body of the method as part of the recommendation. Although comparing method bodies, in terms of their similarity can be employed using clone detection techniques for type 4 clones (behavior), not all method bodies are publicly available via their APIs. Several Java libraries are not open source and only their JAR files are available. So, exploring the similarity of method bodies will be only useful for open source libraries.

Another interesting direction to extend DSKR, is to consider applying it into the service ecosystem. Service operators provide various service matching strategies to support users with integrating newer services into their existing frameworks. However, inaccurate service matching will not only decrease service utilization rate, but also will drastically reduce the satisfaction of users. In this context, we plan on challenging DSKR in providing good service matching, which is able to meet the users’ needs, and competitively perform in comparison with state of the art matching techniques.
9.6 Conclusion

This study addressed the challenge of recommending method mapping when migrating between third-party libraries. We have described a novel approach that recommends method mappings between two unknown libraries using features extracted from the lexical similarity between method names and from the textual similarity from method documentations. We evaluated DSKR by conducting a comparative study with three state-of-art algorithms for method similarities RAPIM [6]. We find that our approach outperforms all existing state-of-the-art approaches, across 8 popular migrations. The qualitative and comparative analysis of our experiments indicate that our approach significantly increases the accuracy of the recommended mappings by an average accuracy of 3.84%, in comparison with existing state-of-the-art studies.
Part III

System Behavior Safety
Chapter 10

System Behavior Safety of Migration

10.1 Introduction

In order for our recommended mappings to be actually applied in practice, we need to verify whether our suggested method mapping changes do not introduce any regression in the existing software’s behavior. Behavior preservation can be accomplished either by the formal verification of the code before and after the migration or by testing the code blocks that were affected by the migration. In this context, we build a tool that takes the list of method mapping for a given migration rule (e.g., testing to JUnit) as input. The tool migrates source code from old third-party library APIs to a new third party library. Then we run the tests and validate how many tests pass and fail after migration. If migration was correct and behaviors have not changed, all test cases should pass. In this chapter, we describe in detail how we develop our migration safety protocol, as a tool. We show examples of applying safety tool on real-world projects, in which, we execute a migration from one library to another one. The verification of behavior preservation will also allow us to verify if the manual validation, for method mapping, that we have built in the previous chapters Dataset [8]. Such verification, will challenge the reliability of our previous findings.

Problem statement. If we want to execute the mappings we recommend, we notice the existence of two cases: first, mapped methods have different signatures, so replacing one with another will trigger compiler errors. Second, mapped methods may have the same signatures but this does not guarantee that they perform the exact same functionality. In order for us to overcome these challenges, we design a schema, in this chapter that helps in migrating between two methods, in case they have different signatures, and we add another layer of testing to test whether two methods
are functionally similar.

## 10.2 Methodology

The goal is to build a tool that is able to auto migrate code from using methods of a retired library and replace them with equivalent methods from the new library. To do so, the tool should scan source code to locate methods that belong to the retired API and replace them with methods from the new API.

In this section, we describe the implementation of the safety tool that is shown in Figure 10.1, which consists of three main components: First, *Migration Schema*, which is a schema that defines method mapping constraints and how the migration should be applied on method-level APIs calls. Second, *Parser* that parses code files and keeps only files that have dependency calls to the old library’s methods. Third, *Auto Migration* that applies method mapping constraints on files with calls from the old library, in order to replace them with the appropriate methods of the new library.

Next, we discuss every component in detail.
10.2.1 Migration Schema

The migration schema is a set of constraints that define how the migration between two methods from two different third-part libraries should be performed. Figure 10.1 shows an example, where we define two migration constraints for migrating from testng to jUnit: The first migration constraint asserts that Source method named `assertNotEquals(double, double, double)` belonging to the package `org.testng.Assert` should be replaced with the method `assertNotSame(String, Object, Object)` belonging to the package `org.junit.Assert`. Parameters order of source method \(P1, P2, P3\) all represented in same order in target method with different data-types \(P1\) was `double`, and becomes `String`, \(P2\) was `double`, and becomes `Object`, and \(P3\) was `double`, and becomes `String`. That means, when we migrate an input project from using testng to jUnit, the method `assertNotEquals` with three parameters signature, will be replaced by `assertNotSame` when signatures matches the schema. Migration constraints also define how the parameters of the retired method get called as parameters in the new method. For example, a retired method may have two parameters, while the new method may need only the first parameter from the retired method. In such case, the scheme specifies which of the parameters, of the old method, is mapped to the new method. The schema also supports tagging the retired package, and the newly added packages, so that when we migrate from one API to another, we can import the right package. The second migration constraint indicates that the Source method `assertTrue(Boolean)` that belongs to the package `org.testng.Assert` should be replaced with the method `assertTrue(Boolean)` that belongs to the package `org.junit.Assert`. The source method has one parameter \(P1\) that needs to be mapped to the target method with the same data-type \(P1\) was `Boolean` and should be `Boolean`. Furthermore, various developers may migrate the same method differently [8] when they migrate code from one library to another. Some of these method mappings are valid in all contexts, and some are not. For example, looking at dataset [5] which contains migration rules between testng to jUnit, two different method mappings for `assertTrue(Boolean, String)` where detected, and the target method was different. For instance, in one mapping, `assertTrue(Boolean, String)` was replaced `assertTrue(Boolean, String)` in three code fragments. In another exaple, `assertTrue(Boolean, String)` was replaced with `fail(String)`, in one code fragment, as shown in Figure 10.1. The question is, which method mapping should we consider when we generate migration schema? One way to solve this issue, is to consider the majority, i.e., the more popular a mapping is, the most likely it can be generalized. So, we rank method mapping based on frequency, a method mapping with the highest number migration examples from different developers becomes recommended. So, for the above-mentioned example, we recommend `assertTrue(Boolean, String)` to be replaced with
**assertTrue(Boolean,String)** since it is more frequent than the other mapping, as shown in portal\(^1\).

### 10.2.2 Parser

The parser is a tool that we wrote to parse code files and filter only the files that have imports from packages of a retired library. So we consider these files for a potential migration. The main job of this file is to reduce the processing time, so the Auto Migration tool does not have to scan the file line by line searching for potential method-level migration. If there is no package imported in a file from a retired library package, that means there is no method call from the retired library.

### 10.2.3 Auto Migration

In this part of the process, we scan all candidate code files that were filtered by the Parser, as potential migration, searching for potential method-level migration. When we find a method in a code file that belongs to retired libraries and the method signature matches a method mapping constraint in the Migration Schema, we consider this case as Match case. It means that API method-level call is eligible for migration. We apply the constraint to migrate the method from a retired library with mapped a method from a new library; also we make sure we import the right package and remove the deprecated package. Besides that, we make sure we replace the parameters of the retired library with the right order of parameters of a method of the new library that is defined in the schema. As we can see in Figure 10.1 method **assertTrue** and **AssertFalse**, get replaced with the same method name in the new library with the same number of a parameter only thing change here is the package import change from `import org.testng.Assert` to `import org.junit.Assert`.

### 10.2.4 Limitations

This tool currently does not support type change migration \([40]\). If there is type change migration, the tool migrates the code and notify the developer there is a type change that needs attention so the developer can manually go and fix the issue by performing the necessary type migration or casting between primitives. Supporting type change is challenging, especially in the case of method overloading and polymorphism. It is hard to know which parameters should be replaced with which, because some parameters will be knows only in run-time. Besides, for the **Generic**

\(^1\)http://migrationlab.net/FunctionMapping.php?MigrateFrom=testng&MigrateTo=junit
data type, it is hard to know what is the data type. This issue is a language-specific problem; it can happen with program languages that require defining data types such as Java, C#, etc., while it would be less of a problem with languages such as Python, where the compiler takes care of type change. Figure 10.2 shows potential type change problem that needs developer attention. In this example, the method `assertEquals(float, float)` has to be replaced with `assertEquals(object, object)`. Our tool will not be able to perform any casting, as shown in Figure 10.4, and the developer needs add the necessary adjustments, such as casting to the type `Object`. Also, the tool currently supports the migration of only static methods, which are typically popular in some libraries, such as testing frameworks. Static methods are easier to handle and replace. Object based methods require more work, since tool needs to support `instance` tracking to be able to track if an instance has been created in a file from a source library classes and track all methods calls for that instance to migrate them.

```java
@Test
public void testGetOpacity() {
    Assert.assertEquals(draggableBehavior.getOpacity(), 0F);
    Assert.assertEquals((Object)draggableBehavior.getOpacity(), 0F);
    draggableBehavior.setOpacity(5F);
    Assert.assertEquals(draggableBehavior.getOpacity(), 5F);
    Assert.assertEquals((Object)draggableBehavior.getOpacity(), 5F);
}
```

Figure 10.2: Type Change Issue

### 10.3 Experiment settings

In this section, we describe the experiment settings that we have done to make a safety tool migrate a testing library at WiQuery. WiQuery is a Java project that has 645 unit tests. We set up our tool to migrate the project from using testng Java testing library to JUnit Java testing library. We manually write the Migration Schema based on method mappings that we have previously detected, for the migration rule testng → JUnit in dataset [5]. Also, we follow our portal to locate the appropriate mapping to choose, in case one source method may have more than one target candidate method.

---

2https://github.com/WiQuery/wiquery/commit/1dd479033f09f3e40a4ee206b0d292021973290
3https://github.com/WiQuery/wiquery/commit/1dd479033f09f3e40a4ee206b0d292021973290
4http://migrationlab.net/FunctionMapping.php?MigrateFrom=testng&MigrateTo=junit
To validate the tool performance, we need to verify if the behavior changes after migration or not. The way we do that is by checking the number of unit tests that fail and pass after we execute the migration. There are 645 unit tests in the project, and all these tests are passing before migration. We assume that, in case our tool performs a wrong migration that changes any method behavior, a test case at least will fail.

10.4 Results Discussion

Figure 10.3 and Figure 10.4 show number of test cases that pass and fail after migrating WiQuery project from using testng test framework to jUnit test framework. We can see only six tests were failing while 639 tests cases pass after the migration was executed. We did the needed manual validation to investigate the reasons of six tests failures. We find that all these tests fail because of type change, which we cannot control, as we have previously explained in the limitations of our approach. We did not find test case that failing because of a wrong replacement of method with another. So, all our mappings were either correct, or in need or developer's manual handling of type migrations.

Figure 10.4 shows details of DraggableBehaviorTestCase test cases. We can see there are two test cases failing: testGetOpacity(), and testGetCancel(). The test case method testGetOpacity() fails because of the type change when we migrate assertEquals(float, float) to assertEquals(object, object). The data-type of the first parameter of method (line 234 to 236) named assertEquals(float, float) changed from float to Object in target method assertEquals(object, object). For the second parameter, We can see that even method signature data-type changes from float to Object will not be an issue because we are passing the direct float value of 0F and 5F that can be casted to object by the compiler to Object data-type. As we discuss earlier our tool currently does not support type change migration, which means that our tool will not automatically perform the casting and will notify the developer about it instead. We notice that when we have performed ourselves the necessary castings and type changes, the failing test case becomes passing.

Furthermore, Figure 10.3 shows test cases OptionsTestCase file. We can see all test cases passing, even, the test case named testGetFloat() that has a similar method mapping migration to previous example, but different signatures where here we migrate from assertEquals(Object, Object) to assertEquals(Object, Object). We do not have any type change issue in test case, when we migrate from assertEquals(Object, Object) to target method assertEquals(object, object) since both data types are same so we can see that the test cases passing.
The current results have limitations; since we run one experiment in one migration rule testng $\rightarrow$ JUnit for one project WiQuery, our experiment results may not be applicable to all method mappings, which we can find in all migration rules, in our Dataset [8]. In the future, we plan to improve the tool and address its limitations, so we can run the tool on all migration rules.

**Summary.** The validation of our safety tool, on the WiQuery project, when migrating from testng $\rightarrow$ JUnit, shows that 639 test case pass out of 645 test cases. Only six test cases fail. They fail because of type changes, which require the developer’s input. None of the test cases fail due to a wrong mapping between source and target methods. This ensures the correctness of our recommendation. This approves that the manual validations that we have done on Dataset [8] for method mapping is reliable for our learning models. However, we cannot generalize our findings, since we have tested on only one migration, since it has all its methods static.
Chapter 10. System Behavior Safety of Migration

Figure 10.3: Sample of Passed Test Case

Figure 10.4: Sample of Failed Test Case
10.5 Conclusion and Future Works

To conclude, this thesis, we have explored the challenges developers face when migrating from one library to another. We have exposed the problems that are encountered when deprecating existing methods and the search to replace them. We have performed a comparative study between techniques related to mining mappings between removed and added libraries. Then we proposed our own mining algorithms that outperformed existing studies. Then, we have proposed a novel machine learning mode that was able to recommend mappings between two libraries. We improve our recommendation by combining domain and source code knowledge. Lastly, gaining a developer’s confidence in using our recommendation relies on how our code changes do not introduce any breaking or behavioral changes. Finally, we validate the safety of the API recommendation on the code behavior.

We discussed, in previous chapters, how a method from the source library could be replaced by different target methods (see Figure 1.3) from the target library. All previous recommendation/detection models that we discussed so far recommend one-to-one mappings that are independent. However, when we have a sequence of methods from the source library that needs to be replaced with another sequence of methods from the target library, existing recommendation models will not take into account their dependence. We plan in the future to also investigate sequence calls as part of the recommendation as well. In this context, we plan to develop sequence to sequence model [74] that considers a sequence of method calls from source library and recommends a sequence of method calls from the target library.
Bibliography


[8] HusseinAlrubaye, MohamedWiemMkaouer, and AliOuni. On the use of information retrieval to automate the detection of third-party java library migration at the method level. In


