UNIVERSITY OF CALGARY

Library Migration: A Retrospective Analysis and Tool

by

Syed Sajjad Hussain Zaidi

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN THE DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

March, 2019

© Syed Sajjad Hussain Zaidi 2019

Abstract

Modern software engineering practices advocate the principle of reuse through third party software libraries. Software libraries offer abstract programming interfaces (APIs) for use by developers, thereby significantly reducing development cost and time. These libraries are however subject to deprecation, vulnerabilities, and instability. Furthermore, as the product matures, developers have to worry about upgrading and migrating to better alternatives in order to attract and retain clients. Refactoring a system to start using a new library can cause a domino effect resulting in serious damage to the software system. Little is currently known about library migration; the few existing studies are either too preliminary or too problematic to tell us much.

We perform an empirical study of 114 open source Java-based software systems in which library migration had occurred. We find that library migration leads to significant compatibility issues and breakage of source code in practice: library migration broke 67% of the software projects in which it occurred, while 22% of the transitively dependent projects broke due to coupling of APIs with external dependencies. Developers do not effectively use available resources to notify their clients about such migrations, preferring to use internal GitHub threads in contrast to the release notes, and even then often failing to discuss or announce the fact that library migration is planned or has been performed. We also found that mitigating library migration requires substantial effort by developers: on average, transformations affected 9.3% of the total classes in the dependent system, while impacting 15% of the total lines of code.

We propose a systematic recommendation tool to assist developers in migrating or upgrading to a new software library. Our prototype tool, named EDW (for External Dependency Watcher), can be utilized to estimate and predict impacts while mitigating migrations among third party libraries. We evaluate EDW by detecting the impacts of migration across three granularities in ten distinct open source projects: our tool has perfect precision for all three granularities; for recall, it obtained over 0.99 for class- and field-granularity while achieving ~0.86 for method-granularity. We conduct a controlled experiment on EDW: human participants were able to perform the tasks in significantly less time and with better precision/recall using EDW as compared to JRipples.

Acknowledgements

I would like to thank my supervisor, Dr. Robert Walker, for giving me the opportunity to work with him. He taught me how to think critically while being pragmatic. Thanks, Rob, for pushing me towards better research when I hit a dead end and for having faith in me. I would also like to thank Dr. Günther Ruhe for his helpful comments and small talks; I was also able to learn a lot from him, while assisting him during his course.

I would like to thank May Sayed for giving me useful tips when I started my program. I would like to thank Hao Men for helping me out in my research. I would also like to thank Hamza Sayed for his helpful feedback and comments on my research. Finally, I would like to acknowledge Jing Zhou's work that guided me to start my research; I would like to meet him in person some day.

Dedication

I would like to dedicate this work to my Dad and Mom; without them I am nothing.I would also like to thank my Taya, Tai, and Chacho for having my back.

Table of Contents

Abstract ii

Acknowledgements iii

Dedication iv

Table of Contents v

List of Figures viii

List of Tables x

1 Introduction 1
- 1.1 Background and Motivation . . . 3
- 1.2 Research Questions and Objectives . . . 7
- 1.3 Contributions . . . 7
- 1.4 Thesis Structure . . . 8

2 Motivation 9
- 2.1 Motivational Scenario . . . 9
- 2.2 Summary . . . 21

3 Related Work 22
- 3.1 Library Upgrading . . . 23
- 3.2 Tool Support for Library Migration . . . 26
- 3.3 Change Impact Analysis . . . 28
  - 3.3.1 Incremental change . . . 30
- 3.4 Summary . . . 31

4 Data Collection and Analysis 32
- 4.1 Selection of Projects . . . 35
  - 4.1.1 GitHub Crawler . . . 35
  - 4.1.2 Code Analyzer . . . 36
  - 4.1.3 Manual Selection . . . 37
- 4.2 Data Gathering . . . 38
  - 4.2.1 Find it EZ . . . 38

4.2.2 JAPICC . . . 39
4.2.3 Code Compare . . . 40
4.3 Analysis and Results . . . 40
4.4 Summary . . . 53

5 Tool **58**
5.1 Overview of EDW . . . 60
5.2 Dependency Viewer Module . . . 62
5.2.1 External JAR submodule . . . 64
5.2.2 Affected classes submodule . . . 65
5.2.3 Imports submodule . . . 65
5.3 Editor Module . . . 65
5.4 Impact Analysis Viewer Module . . . 66
5.5 Graph Builder Module . . . 69
5.6 Metrics Builder Module . . . 69
5.7 Workflow . . . 71
5.8 Summary . . . 73

6 Simulation Study **74**
6.1 Selection of Projects . . . 74
6.2 Procedure . . . 76
6.2.1 Ground truth . . . 76
6.2.2 Estimated impact set for EDW . . . 77
6.2.3 Estimated impact set for JRipples . . . 77
6.2.4 Comparing estimated impact set and ground truth . . . 78
6.3 Results . . . 79
6.3.1 RQ5: How well is EDW able to locate the initial impact set for library migration? . . . 79
6.3.2 RQ6: What factors impact how well EDW performs? . . . 83
6.4 Summary . . . 87

7 Human-Subjects Experiment **88**
7.1 Hypotheses . . . 88
7.2 Subject System . . . 89
7.3 Tasks . . . 90
7.4 Participants . . . 90
7.5 Experimental Procedure . . . 91
7.6 Data Collection . . . 92
7.7 Results . . . 93
7.7.1 Quantitative . . . 93
7.7.2 Qualitative . . . 96
7.8 Limitations . . . 101
7.9 Summary . . . 101

8 Discussion **103**
8.1 Trends in Software Library Usage . . . 103
8.2 Code Breaking of Systems and Their Clients . . . 104
8.2.1 Categories of Libraries that Broke Clients . . . 105
8.3 Notification of Software Library Migration . . . 105
8.4 Evaluation of EDW . . . 106
8.5 Threats to Validity . . . 107
8.5.1 Threats to validity of the library migration study . . . 107
8.5.2 Threats to validity for the evaluation of EDW . . . 108
8.6 Future Work . . . 109
8.6.1 Knowledge base for external libraries . . . 109
8.6.2 Improve effort estimation . . . 110
8.6.3 Improvements to EDW . . . 110
8.7 Summary . . . 111

9 Conclusion **113**
9.1 Future Work . . . 115

Bibliography **117**

A Studied Systems **125**

B Studied Library Migrations **130**

C Systems that Broke Transitively Dependent Clients **139**

D Systems Used for Evaluation **141**

E Individual Project Results for Evaluation **143**

F Experimental Materials **148**
F.1 Task Descriptions and Instructions . . . 149
F.2 Pre-Study Questionnaire . . . 151
F.3 Post-Task Questionnaire . . . 153
F.4 Final Questionnaire . . . 154

G Detailed Experimental Results **156**

List of Figures

1.1 Relationships between a studied project, a system that depends on that project, and the libraries depended upon by both. 2

2.1 Dependencies from classes in the developer's system to Guava that he ultimately discovers. 15
2.2 Transitive dependencies, from utilityFunction1 in the developer's system to Guava, that he ultimately discovers. 16

4.1 Approach for data collection and analysis. 34
4.2 Algorithm to detect elimination of external dependencies. 37
4.3 Project count versus reasons behind successful library migration without code breaking. 41
4.4 Project count versus reasons behind complete removal of dependency from a project. 45
4.5 Project count versus alternative library selection criterion. 48
4.6 Percentage of classes impacted versus Maven project (sorted in descending order). . 54
4.7 Percentage of classes impacted versus Android project (sorted in descending order). 54
4.8 Percentage of classes impacted versus project from either platform (sorted in descending order). 55
4.9 Percentage of LOC impacted versus Maven project (sorted in descending order). . . 55
4.10 Percentage of LOC impacted versus Android project (sorted in descending order). . 56
4.11 Percentage of LOC impacted versus project from either platform (sorted in descending order). 56

5.1 EDW analysis on the project cron-utils while using our standard view arrangement. . 61
5.2 EDW structure and interaction between modules. 62
5.3 Dependency viewer after analysis on the project cron-utils. 63
5.4 Editor after selection of the affected class StringValidations. 67
5.5 Impact analysis view after selecting the affected class StringValidations from the dependency viewer. 68
5.6 Impact analysis view after performing complete CIA on Guava for cron-utils. 70
5.7 Dependency graph for the class StringValidations in the project cron-utils. 70
5.8 Four metrics calculated for the Guava library in the project cron-utils. 71

6.1 The field Logger used in a nested member. 84
6.2 Areas marked "Next" and "Impacted" by the impact analysis view. 85

6.3 Method from the class AlwaysFieldValueGenerator in project cron-utils that is dependent on the class List from the Guava library. 86
6.4 Method from the class AlwaysFieldValueGeneratorTest that depends on the class AlwaysFieldValueGenerator. 86

List of Tables

4.1 Libraries that support smooth replacement with another one while having minimal code impact. . . . 43
4.2 Mutually exclusive classification of rationales behind library migration. . . . 48
4.3 Errors and types that impact client code after library migration. . . . 51
4.4 Percentage of impacted classes. . . . 52
4.5 Percentage of impacted LOC. . . . 52

6.1 Systems for evaluation. . . . 75
6.2 Approach mean accuracy for class-granularity. . . . 80
6.3 Approach mean accuracy for method- and field-granularity. . . . 80
6.4 Recall of projects at method-granularity. . . . 82

7.1 Guava impact set for tasks. . . . 90
7.2 Overview of participants' experience. . . . 91
7.3 Time taken by individual participants for tasks (in minutes). . . . 93
7.4 Average time taken in minutes for tasks. . . . 93
7.5 Results of the Kolmogorov–Smirnov test for elapsed time. . . . 94
7.6 Results of *t* tests on time taken. . . . 94
7.7 Average JRipples task result. . . . 95
7.8 Results for Wilcoxon test on F-score data combined from Task 1 and 2. . . . 96
7.9 Results for the post-task questionnaire. . . . 97
7.10 JRipples task categorization. . . . 100

8.1 Preferred libraries involved in Android migrations. . . . 104
8.2 Domain of libraries that have a higher probability of breaking transitively dependent clients. . . . 105

A.1 Maven projects. . . . 125
A.2 Android projects. . . . 128

B.1 Maven projects. . . . 130
B.2 Android projects. . . . 136

C.1 Breaking APIs. . . . 139

D.1 Systems for evaluation. . . . 141

E.1 Individual project results for class-granularity, JRipples search strategy 1. . . . 144

E.2 Individual project results for class-granularity, JRipples search strategy 2. 144
E.3 Individual project results for class-granularity, JRipples search strategy 3. 145
E.4 Individual project results for class-granularity, EDW. 145
E.5 Individual project results for method-granularity, JRipples search strategy 1. 146
E.6 Individual project results for method-granularity, EDW. 146
E.7 Individual project results for field-granularity, JRipples search strategy 1. 147
E.8 Individual project results for field-granularity, EDW. 147

G.1 Effectiveness: JRipples on Task 1 at class-granularity. 157
G.2 Effectiveness: JRipples on Task 1 at method-granularity. 157
G.3 Effectiveness: JRipples on Task 1 at field-granularity. 157
G.4 Effectiveness: JRipples on Task 2 at class-granularity. 158
G.5 Effectiveness: JRipples on Task 2 at method-granularity. 158
G.6 Effectiveness: JRipples on Task 2 at field-granularity. 158

Chapter 1

Introduction

Research suggests that software reuse positively impacts the quality of software at a reduced cost [Lim, 1994]. The demand for reuse has promoted the now-predominant practice of using third-party software libraries or dependencies [Ebert, 2008]. Software libraries are readily available without payment from various online repositories such as Maven Central Repository,[1] SourceForge,[2] and GitHub.[3] The reuse of tested libraries from these repositories results in improved development time, usability, and maintainability [Mohagheghi and Conradi, 2007, Jezek et al., 2015]. Libraries expose their application programming interface (API) to allow third-party developers to interact with the library functionalities; an API is an implicit agreement between the software library and the client application that uses it. As a result, the clients can use exposed functionalities of the library without knowledge of the underlying implementation, at least in principle.

Software reuse through external libraries creates strong dependencies between the client system and the library [Dig and Johnson, 2006]. Old versions of libraries are subject to deprecation and evolution, resulting in newer releases. Due to emergence of better alternatives in the market, developers have to consider either migrating or updating their third-party libraries. However, the difficulties of dependency management are greatly underestimated by developers, and most

[1] https://search.maven.org/ [accessed 2018/10/11]
[2] https://sourceforge.net/ [accessed 2018/10/11]
[3] https://github.com/ [accessed 2018/10/11]

software systems keep their outdated libraries [Kula et al., 2018b]. While it might seem like an obvious decision to shift towards a better library, systems with complex dependencies require careful consideration before doing so; upgrading a system to start using a new library version results in additional rework for the developers [Bavota et al., 2015].

Figure 1.1 explains some key context and terminology that we use in this thesis. We study software projects that depend on external libraries and that are themselves depended on by yet other systems (i.e., transitively dependent systems); the studied projects undergo library migration, as per the following. The solid arrows express dependency relationships in which the project is directly involved, while the dashed arrows express dependency relationships that are indirectly mediated by the project; the "**x**" symbols indicate relationships that are eliminated, and the "**+**" symbols indicate relationships that are added. Thus, the studied project undergoes library migration as its dependency on the external library is replaced by an alternative dependency. Before migration, the project depended on the old library; as a consequence, its dependent system (i.e., another system that depended on the studied project) had a transitive dependency on the old library. After migration, the system's dependency was moved to the new library; as a result, the transitively dependent system was required to change as well.

No systematic study has been conducted on migrations between distinct libraries from different vendors. Thus, we have little evidence as to the frequency and severity of this phenomenon; when

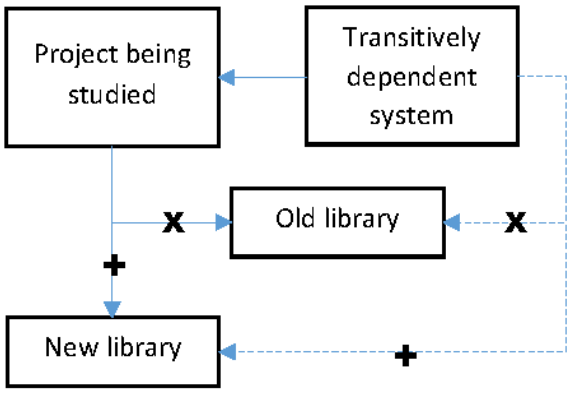

Figure 1.1: Relationships between a studied project, a system that depends on that project, and the libraries depended upon by both.

and if to perform library migration and what the consequences of such transitions may be, all remain unclear. The first goal of this thesis is to empirically study the migration between external software libraries; we examine 114 open source Java[4]-based systems, in which library migration has occurred, to this end. The aim of this study is to empirically explore the impacts of library migrations on Java projects and their transitively dependent systems.

The second goal of this thesis complements the first one. As a corollary to the lack of knowledge about library migration, current tool support is lacking. For the most part, developers are on their own while performing library transitions, which is error-prone and wastes time. On the other hand, developers can reduce their manual effort by utilizing a recommendation system that suggests the impacts of library migration: we introduce a prototype tool (EDW) that can assist developers in performing library migration.

1.1 Background and Motivation

Nowadays, software developers prefer to use external libraries for integrating a broad range of functionalities into their systems [de la Mora and Nadi, 2018]. As a consequence, the availability of third party libraries has significantly increased over the years. Fox [2010] reported Maven Central Repository to contain over 260,000 artifacts, serving over 70 million downloads per week; as of August 2018, the repository had grown to 3,081,887 artifacts. These software libraries generally support smooth integration into existing systems with minimal effort.

Software libraries expose their application programming interfaces (APIs) to client applications, making their usage possible without requiring knowledge of the implementation details, in principle. An API is the core agreement between the dependee (library) and the dependent (software system), the breakage of which can result in issues such as lack of backwards compatibility. Ideally after successful integration, the system should continue to use the same library without any alterations; the system can then evolve without worrying about any change in the used dependency. However,

[4]https://www.java.com/ [accessed 2018/10/11]

the reality contradicts the ideal: the APIs of libraries evolve and deprecate over time for the same reason as any software [Lientz, 1983]. Similar to other software, the outdated libraries that do not undergo continual changes can become progressively less useful [Lehman, 1996]. Therefore, developers have to consider migrating from deprecated and outdated libraries in favour of better alternatives.

Availability of better alternatives or issues with existing ones can drive developers to migrate to new libraries. For example, JUNG (the Java Universal Network & Graph Framework)[5] is an open source Java-based modelling and visualization framework, a common choice for developers who require a means for modelling, analyzing, and visualizing data that can be represented as a graph or network. JUNG had depended on the Apache Commons Collections,[6] as this library provides many advanced utilities and data structures free of cost that were useful for JUNG. JUNG was strongly coupled with the external library, exposing some of its data structures in their APIs; however, in 2016 the JUNG developers decided to replace it with Guava (Google Core Libraries for Java),[7] another commonly used collection library. The decision to migrate was made due to the discovery of a security vulnerability in Apache Commons Collections. In addition, the JUNG developers thought that Guava was a better alternative to rely on, as per their release notes[8]:

> The commons-collections-generics library dependency has been replaced with Guava v19.
>
> This change is not backwards-compatible with existing code. It was necessary because of a security issue, but we also decided that Guava was a much better library to depend on in future for a variety of reasons (some of which will become apparent in future JUNG releases). In terms of interfaces, the replacement of commons-collections with Guava has the following user-visible effects:
>
> - Predicate to Predicate
> - Factory to Supplier

[5] https://github.com/jrtom/jung [accessed 2018/10/11]
[6] https://commons.apache.org/proper/commons-collections/ [accessed 2018/10/11]
[7] https://github.com/google/guava [accessed 2018/10/11]
[8] https://github.com/jrtom/jung/releases/tag/jung-2.1 [accessed 2018/10/11]

- Transformer to Function

There are also (of course) many internal changes related to this replacement, e.g.,

- ChainedTransformer to Functions.compose()
- Buffer, UnboundedFifoBuffer to Queue, LinkedList
- BidiMap to BiMap

Available research [McDonnell et al., 2013, Kula et al., 2015, Bogart et al., 2016] mostly concentrates on exploring library upgrading. These studies have explored the impacts and consequences of modernization between different versions of the same library. Library upgrading is usually less extreme than library migration, as it involves updating between different versions of the same library, and so one would expect there to exist significant commonality. Library migration on the other hand, is performed between two distinct libraries, generally from different vendors. Due to scarcity of research on library migration, we have little evidence as the frequency and severity of this phenomenon, such as the impacts of library migration on a project and its transitively dependent systems. For example, checkstyle[9] is a Java-based development tool that allows developers to write code that adheres to best coding practices. The checkstyle developers decided to eliminate their dependency on Guava in favour of moving to the standard Java software development kit (SDK). As a result of dropping Guava, the new release was not backwards compatible, since Guava was being exposed in their APIs, as follows.[10]

> Since this change is being done in API code, this will break backward compatibility with any existing code that uses it if it is not re-compiled. When upgrading, to mitigate this issue, all custom JARs must be recompiled against new checkstyle code.

Research has explored the rationales that drive developers to perform library upgrading. Their studies have found that library *upgrading* normally happens after the occurrence of problems and issues in the existing dependency version [Bavota et al., 2015]. However, the findings in this domain

[9] https://github.com/checkstyle/checkstyle [accessed 2018/10/11]

[10] https://github.com/checkstyle/checkstyle/issues/3455 [accessed 2018/10/11]

are restricted to evolution of APIs in different versions of a library. Therefore, we are not clear about the rationales that drive developers to perform library *migrations*. Hypothetically, library migrations should be motivated by legitimate rationales such as improvement in stability, performance, size, and quality of the system. For example, the project DCMonitor[11] underwent library migration from the database system named InfluxDB[12] that it depended on, to an alternative called Prometheus.[13] The migration was motivated by improvement in performance and stability[14]:

> I'm really tired of InfluxDB!
>
> From my experience, it is very unstable, highly [with high] resource usage (CPU, disk IO), and the Java client is not developer friendly. Since DCMonitor is mean[t] to be a light weight tool, it should be easy to use, maintain and resource restraint. But relying on InfluxDB makes things difficult. Luckily the history metric storage is only used like a k-v storage with time range and group by, there are lots of storage system[s] [that] can handle this. Maybe the good old MySQL is a nice option.

We note that developers have different resources available to them for notifying third-party developers about migrations and their rationales (e.g., release notes, change logs, etc.). However, which of these means are more effective and which are more often used in the community is also not clear.

The effort required for mitigating library migration is still not apparent and needs further exploration. Xavier et al. [2017] studied the impacts of backward incompatibility due to evolution of library APIs, finding that 14.78% of API changes broke backward compatibility while impacting 2.54% of the clients. While Bavota et al. [2015] found that the impacts of upgrading to a newer version of library are rather limited, as only 5% of the total classes and 6% of kLOC are impacted due to upgrading; the limited impact is to be expected as vendors try to ensure that the newer version of their library is backward compatible. Most library version upgrades involve fewer changes in

[11] https://github.com/shunfei/DCMonitor [accessed 2018/10/11]
[12] https://mvnrepository.com/artifact/org.influxdb/influxdb-java [accessed 2018/10/11]
[13] https://github.com/prometheus/prometheus [accessed 2018/10/11]
[14] https://github.com/shunfei/DCMonitor/issues/28 [accessed 2018/10/11]

the client code for integration. However, this is not the case during migration to another library altogether, as the dependencies are from independent developers with the possibility of having incompatible initial structures.

1.2 Research Questions and Objectives

The first objective of our study is to answer the following research questions, investigating the phenomenon of library migration.

RQ0: How often does library migration occur?

RQ1: How often does library migration in a software system break it?

RQ2: What means are used by developers for notifying about library migrations?

RQ3: What impact does library migration from a system have on its dependent clients?

RQ4: How much effort is required for mitigating library migration?

Our second objective is to assist developers in the practical problem of migrating from one library to another, providing developer support in the form of a tool we call EDW. Our research questions for this objective are as follows.

RQ5: How well is EDW able to locate the initial impact set for library migration?

RQ6: What factors impact how well EDW performs?

1.3 Contributions

The thesis has two main contributions.

- We empirically explore library migration in 114 Java-based projects, selected from both Maven and Android[15] platforms.

[15] https://developer.android.com/ [accessed 2018/10/11]

- We develop a prototype tool, EDW, to assist the developer in performing library migration. EDW utilizes change impact analysis for locating the impacted areas of change for the user, and provides resources through which library migration can be performed systematically. We conduct both a simulation and an experiment with human participants, in order to evaluate EDW. The high precision and recall along with significant time savings of our tool show that it has promise in guiding developers during the process of library migration.

1.4 Thesis Structure

The remainder of the thesis is structured as follows. Chapter 2 describes a motivational scenarios regarding how prior knowledge and support would assist a developer who wants to perform a library migration. Chapter 3 presents related work, elaborating on how it lacks in answering our research questions. Chapter 4 discusses the project selection criteria,data gathering, analysis and its results for our empirical study on library migration. Chapter 5 presents our prototype tool, EDW, and explains the workings of its internal modules. Chapter 6 discusses the evaluation of EDW and JRipples for our simulation study. Chapter 7 presents our controlled experiment using human participants, in order to compare the effectiveness and efficiency of EDW with JRipples. Chapter 8 discusses some of the related topics of our research while specifying its threats to validity and scope of future work. Chapter 9 concludes the thesis.

Chapter 2

Motivation

Nowadays, software developers rely on third-party libraries to provide functionality in performing various tasks (i.e., data parsing, networking, etc.) in their systems [de la Mora and Nadi, 2018]. There are several stable and tested libraries available online that can be used by the developers for these functionalities, access to which is delivered by means of their APIs [Fox, 2010]. Using an available library avoids wasting time on re-inventing the wheel and can spare significant resources for the non-trivial modules of a system. However, the utilized libraries can easily become deprecated or fail to be enhanced as the environment around them changes for the same reason as any software [Lientz, 1983]. As a result, the client developers might waste significant effort, time, and resources in performing library migration.

Section 2.1 presents a motivational scenario to demonstrate how issues in third party libraries can create the need for migration. The scenario also highlights how prior knowledge about migration and a readily available tool would help us in such a situation.

2.1 Motivational Scenario

Imagine a software developer who wants to create a novel universal visualization framework as the basis for his software startup company. His framework would allow the user to model, analyze, and visualize data as a graph or network, while requiring fewer resources than the alternatives but

providing significantly better performance and results. Due to the competition from other available benchmark software (i.e., JUNG, JGraph, etc.), the developer wants to put his product in the market as quickly as possible. Since the time pressure is high and the available resources are limited, he decides to prioritize development of the visualization algorithm. This algorithm is the key element that would make this application stand out amongst its competitors.

The developer decides to adhere to the principle of reuse by adopting external software libraries where possible. Since the emphasis of the project would be on handling data, the algorithm would require various readily available data structures and their respective utilities. In order to search for a library that would fulfill his needs, the developer uses Maven Central Repository search, as is commonly done. Keeping all the resources and requirements in mind, he chooses to use one of the popular libraries from the search: Guava, a collection and extension of the standard Java collections framework that provides advanced utilities to the developer. With time this dependency becomes an integral part of his project, as it becomes coupled with the APIs of the system; with successive releases, his product becomes popular and is downloaded by many clients.

After some time, the developer discovers that the library he had decided to use during the initial development phase has severe security vulnerabilities despite having been deemed safe and stable by the community initially. As a result, support for this library has declined over the years and the vendor of the library started to show intentions of deprecating it after several other security vulnerabilities were found; thus, this library has become a risk for its dependent systems. The developer is now in a dilemma about what to do, as his system has become strongly coupled with the library. As a consequence, migration from Guava to something else will not be backward compatible. The decision regarding migration from this library is delicate, as the impacts can propagate to other working modules of the system as well as to transitively dependent clients.

Upon recognizing the vulnerabilities in the used library, the developer has the following options:

1. Keep using the same library while exposing the clients to a security risk.

2. Replace the existing library with a newer one that does not have this security vulnerability.

3. Replace the existing library with a module developed in-house.

If the developer chooses the first option, his clients will eventually discover the transitive vulnerability and would stop using his system; his reputation would be damaged, as it would be evident that he knew about the problem but exposed clients to it anyways; and besides, he recognizes that such (in)action would be unethical. Thus, the second or third options are more appropriate. The second option would require finding a reasonable alternative for the existing library, while the third option would cost more time and resources but could save the company from future, similar issues. Both options would require migrating from the existing library to an alternative one; however, replacing or removing a strongly coupled dependency is not a simple matter.

Finally after reviewing his available time and resources, the developer decides to go with option 3. To avoid the need for future library migration, the developer decides to replace Guava with functionality from a standard Java software development kit (SDK) instead of introducing a new external library. (An SDK is shipped along with each release of the Java platform, which does not require addition of external libraries for using offered functionalities.) As the available tool support for library migration is lacking, the developer decides to use a manual process for performing the change, as follows.

1. The developer needs to first locate classes that are explicitly dependent on the library to be replaced. These are the initial impact points that would be directly affected in migrating from the library. This step would require a thorough search for all directly affected classes throughout his system.

2. After locating impacts at the class-level granularity, the developer needs to locate the affected members (i.e., methods, fields, etc.) and statements inside these classes.

3. Once the initial impact points are found, the developer needs to figure out the transitively impacted areas. These secondary points would be affected as a result of changing the initial impact points. These secondary points of impact are harder to interpret, as they can arise from detailed semantic dependencies.

4. Finally he has to replace the initial impact points with calls to the Java SDK, changing the secondary impact points accordingly.

The first step in locating the affected classes requires the developer to figure out the list of imported classes from Guava. In order to efficiently locate these imports, the developer localizes his manual search by utilizing his knowledge of the system, first exploring his utility module. The utility module contains general-purpose functions that are used throughout the system; based on the developer's experience, this module has the highest probability of using data structures from Guava. The developer starts to explore the module in an iterative manner. He locates the class UtilityClass1 that relies directly on Guava (see Listing 2.1); the class imports five classes from Guava (lines 2–6). After discovering these imports, the developer decides to locate all classes relying on them, by means of the search functionality provided by his integrated development environment (IDE); he finds references to four more classes, all of which reside in the utility module, reinforcing his assumption that the use of the external library is localized to the utility module.

Listing 2.1: Listing of UtilityClass1.

```
1  //  ... other imports
2  import com.google.common.base.Joiner;
3  import com.google.common.collect.Range;
4  import com.google.common.collect.Lists;
5  import com.google.common.collect.Sets;
6  import com.google.common.base.Strings;
7
8  public class UtilityClass1  {
9    public static final String SEPARATOR = "_";
10   private Lists fields ;
11   // ...
12
13   public static Strings utilityFunction1 (boolean edges, Graph graph) {
```

```
14      Strings graphName;
15      if (edges) {
16          final List<String> edgesByPart = new LinkedList<>();
17          graphName = Joiner.on(SEPARATOR).join(edgesByPart);
18      }
19      else
20          graphName = "EMPTY";
21
22      return graphName;
23  }
24
25  private Lists utilityFunction2 (ZonedDateTime graphReferenceDate, String[] from, int year, int
        month, List dateToBeSelectedFrom) {
26      Range<Integer> rangeForMonth = Range.closedOpen(LocalDate.of(year, month,
            1).getDayOfYear(), month == 12 ? LocalDate.of(year, 12, 31).getDayOfYear() + 1 :
            LocalDate.of(year, month + 1, 1).getDayOfYear());
27      Sets<Integer> selectedDatesBasedOnMonth = Sets.newHashSet();
28
29      for(Integer dayOfYear : dateToBeSelectedFrom) {
30          if (rangeForMonth.contains(dayOfYear))
31              selectedDatesBasedOnMonth.add(dayOfYear);
32      }
33
34      Lists finalSelectedDates = new Lists.newArrayList(Arrays.asList(from));
35      for(Integer dayOfYear : selectedDatesBasedOnMonth)
36          finalSelectedDates.add(LocalDate.ofYearDay(graphReferenceDate.getYear(),
                dayOfYear).getDayOfMonth());
37
38      return finalSelectedDates;
```

```
39    }
40
41    public void utilityFunction3 (String graphName) {
42      if (Strings.isNullOrEmpty(graphName)) {
43        // ...
44      }
45    }
46
47    public void utilityFunction4 (Object graph) {
48      fields = Lists.newArrayList();
49      fields.add(graph);
50    }
51
52    // ...
53  }
```

Realizing that the five initially located classes depend on only two imported classes (com.google.common.base and com.google.common.collect), the developer decides to perform an IDE search using the common path prefix (i.e., com.google.common) with the expectation of immediately locating all affected classes. However, his search yields many false positives, as the system also depends upon a testing framework, called Truth, with imported classes in common with Guava (e.g., com.google.common.Truth). Truth has more extensive usage and coupling with the system in comparison to Guava. Instead of wasting time on going through false positives, the developer decides to perform another IDE search while being more specific about imports by using com.google.common.base and com.google.common.collect. Going through the results he further discovers four more imported classes (com.google.common.collect.Iterables, com.google.common.collect.Table, com.google.common.base.MoreObjects, and com.google.common.base.Function) that he previously missed. The developer then locates all affected classes of these imports before moving

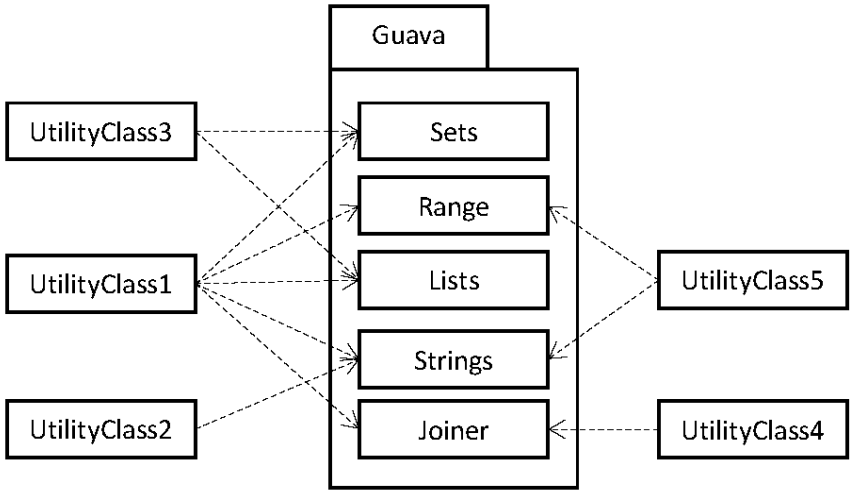

Figure 2.1: Dependencies from classes in the developer's system to Guava that he ultimately discovers.

to the next step. Figure 2.1 illustrates the dependencies between from his system's classes and the classes from Guava, as discovered by the developer.

Having performed his extensive search in the utility module, the developer now thinks that he has located all affected classes relying on Guava within his system, so he moves to Step 2. The developer starts his search for affected members from UtilityClass1: he manually locates the impacted LOC directly using the imports in the class, and in the process, he tries to comprehend the purpose of using the library. His search yields four utility methods and one field that depend directly on the Guava library; Listing 2.1 shows the related logic in the utility function and fields. For example, utilityFunction2() generates the list of dates based on the given year and month: the function relies on the Range class for defining an interval of days falling in the given month range, which is then utilized to populate the list of final selected dates to be returned after performing some logic on it.

After successfully performing Step 2, the developer now thinks he has comprehended the coupling of the system with the library. He can now move to locating transitive impacts of library migration. In order to perform Step 3, the developer utilizes an impact analysis tool (Find it EZ[16]) that recommends the ripple effects of changing the initial impact points. The developer is required to perform impact analysis on each of the located members of affected classes from Step 2. He

[16]https://www.finditez.com/ [accessed 2018/10/11]

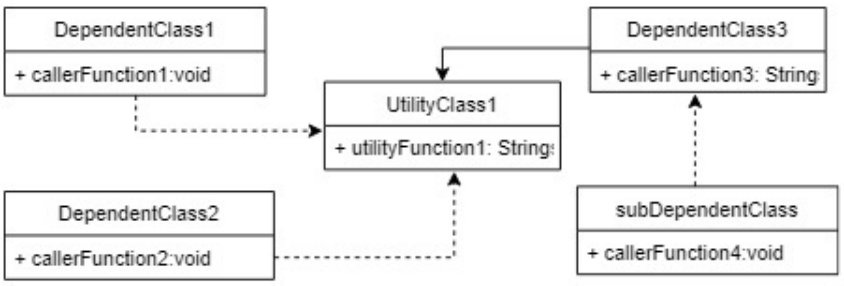

Figure 2.2: Transitive dependencies, from utilityFunction1 in the developer's system to Guava, that he ultimately discovers.

starts Step 3 from UtilityClass1 (line 2.1) by locating the ripple effects of changing utilityFunction1() (line 13). The impact analysis tool would report any member of a class that has a dependency on (i.e., a call to) utilityFunction1() without being specific. Figure 2.2, shows the recommended classes along with their members that depend on utilityFunction1(). The dashed lines show dependencies that will not be impacted after library migration. The developer has to manually review each transitively affected member in order to verify whether it would actually be impacted after migration or not. For example, the transitively dependent classes DependentClass1 and DependentClass2 will not be impacted after library migration, as callerFunction1() and callerFunction2() only utilize the returned value (i.e., graphName) as-is in a conditional statement. On the other hand, DependentClass3 is transitively impacted as it first stores the returned graphName, before performing further logic on it. Since the developer has determined that DependentClass3 would be impacted, he has to further determine the ripple effects of callerFunction3 on the system. In this case subDependentClass is not impacted, as callerFunction4 also uses the returned value as-is. Similarly the developer determines transitive impacts for other affected members from Step 2.

Finally after locating the impacted classes and the directly and transitively affected members, and after comprehending library usage, he moves to Step 4 where he replaces the library with the Java SDK. The developer has to find the appropriate alternative options in the SDK that would retain the functionality after replacement. Listing 2.2 shows the members of class UtilityClass1 after they are replaced with the alternative library; the developer has to keep the same semantics while migrating between the two libraries that have different structure. For example, in utilityFunction2()

he replaces the use of Range with LocalDate (lines 28–29) for generating the boundary months that are then used to populate the list of final selected dates. Similar changes are made for the imports Joiner (line 18), Sets (line 32), Lists (lines 43 and 12), and Strings (line 38).

Listing 2.2: Listing of modified UtilityClass1.

```
1  // ..... other imports
2  import java.lang.String;
3  import java.time.LocalDate;
4  import java.time.Month;
5  import java.util.stream.Collectors;
6  import java.util.Collections;
7  import java.lang.Math;
8  import java.util.List;
9
10 public class UtilityClass1 {
11   public static final String SEPARATOR = "_";
12   private List<Object> fields;
13
14   public static String utilityFunction1 (boolean edges, Graph graph) {
15     String graphName;
16     if (edges) {
17       final List<String> edgesByPart = new LinkedList<>();
18       graphName = String.join(SEPARATOR, edgesByPart);
19     }
20     else
21       graphName = "EMPTY";
22
23     return graphName;
24   }
```

```
25
26    private List utilityFunction2 (ZonedDateTime graphReferenceDate, String[] from, int year, int
          month, List dateToBeSelectedFrom) {
27      // Generate Range
28      int low = LocalDate.of(year, month, 1).getDayOfYear();
29      int high = month == 12 ? LocalDate.of(year, 12, 31).getDayOfYear() + 1 : LocalDate.of(year,
            month + 1, 1).getDayOfYear();
30
31      // Generate Days candidates
32      List<Integer> finalSelectedDates = dateToBeSelectedFrom.stream().filter(dayOfYear ->
            dayOfYear >= low && dayOfYear < high).map(dayOfYear ->
            LocalDate.ofYearDay(graphReferenceDate.getYear(),
            dayOfYear).getDayOfMonth()).collect(Collectors.toList());
33
34      return finalSelectedDates;
35    }
36
37  public void utilityFunction3 (String graphName) {
38    if (graphName!= null && !graphName.isEmpty()) {
39      /* ... */ }
40   }
41
42    public void utilityFunction4 (Object Graph) {
43      fields = Collections. singletonList (Graph);
44    }
45
46    // ...
47  }
```

After performing the change and running unit tests to confirm semantic preservation of the system, the developer thinks that he has successfully migrated from the library. As the imports from Guava are not needed anymore, the developer decides to remove them before dropping the Guava dependency itself. However, in the process of doing this, he finds that he failed to consider the class com.google.common.primitives.Ints imported by the class UtilityClass5, shown in Listing 2.3. The import declaration was buried between other import declarations and resides in a different package (com.google.common.primitives), which resulted in him missing it during the first iteration. Realizing his mistake, the developer again performs an IDE search for other imported classes in this package and then repeats Steps 2–4 on the located classes.

Listing 2.3: Listing of UtilityClass5.

```
1  //  ... other imports
2  import com.google.common.primitives.Ints;
3  //  ... other imports
4
5  public class UtilityClass1  {
6    public void utilityFunction5 ()  {
7      int checked = Ints.checkedCast(Math.round(getGraphSize()));
8      if (checked) {
9        //  ...
10     }
11   }
12 }
13
14 //  ...
```

After making the required changes, the developer confirms that he has successfully migrated from the library by removing imports from the system. As all usage of the library has been removed from the system, the developer finally removes Guava from the system's build path and

re-builds the system. The build fails: his initial assumption about localized usage of Guava was wrong. The file management module written by another developer was using the imported classes com.google.common.io.Files and com.google.common.io.CharStreams directly. These import classes reside in a different package and belonged to another module that resulted in them being missed in previous searches. Again realizing his mistake, this time he decides to perform a thorough search of the whole system based on build errors. He has to perform Steps 1–4 again on any newly discovered imports while removing the imports one at a time.

Finally, after having performed multiple iterations of the steps, the developer successfully replaces Guava with the Java SDK. At this point, the developer has spent significant time and effort on the following:

1. Manual search for the directly affected classes, likely missing some and/or getting false hits.
2. Manual exploration of the directly affected classes in search of impacted members and their ripple effects based on his past experience, via trial and error; in the process, he likely misses some and/or gets false hits, plus he has to remember all the details.
3. Risk of the need for repeated iteration of the same steps, due to his limited knowledge and unsystematic, manual approach.
4. The need to comprehend library usage for replacing it with another alternative.

On the other hand, the developer could have saved both time and effort if he had known beforehand the affected classes of migration, along with their affected members and transitive impacts. Moreover, migration could have been performed more efficiently if he had followed a systematic approach with some insight and information on library migrations. The developer could then have spent more time on comprehending how the library is used and how to replace it.

2.2 Summary

We described a scenario in which a software system was in need of migration from an unstable and unsafe external library.

The motivational scenario demonstrates how mitigating migration of a strongly coupled dependency requires the developer to perform different steps manually. The developer is required to first find the impacted areas on different granularities: this involves locating the affected classes, members, and lines of code, in this specific order. Once the developer has finalized the directly affected classes the next step is to find the transitive impacted segments and their ripple effects. All of these steps have to performed while remembering or recording results of the preceding steps and actions. As a result, errors can easily be introduced into the procedure that require iteration of the steps and repeated effort. Tool support for the mechanical aspects of the procedure would free the developer's time and attention on the more complex aspects of the problem.

Chapter 3

Related Work

Library migration falls under the general category of software evolution; however, our research is more strongly related to the sub-domain of library upgrading, which considers upgrading existing dependencies on libraries to new versions. Efforts have been made to understand library upgrading by exploring evolution of APIs in dependencies, by conducting empirical studies specific to the domains of API evolution, API breaking, and API migration. However, all of these studies are restricted to an "intra"-level; that is, they are centred on studying changes among different versions or variants of the same library. In contrast, library migration between independent libraries has not been explored before, and we have little evidence as to the frequency and severity of this phenomenon. Our research targets the scarcity of information regarding such "inter"-level library migrations and explores the impacts of these changes on dependent systems. Due to the lack of knowledge regarding library migration, we also lack specialized development tools aimed at helping the developer to transition their software between libraries.

Section 3.1 discusses previous research on library upgrading. Section 3.2 highlights existing tool support for library migration. Section 3.3 discusses approaches on change impact analysis. Section 3.3.1 highlights the process of incremental change.

3.1 Library Upgrading

Library upgrading (also known as "library modernization") due to API evolution has been studied in recent years. Researchers have empirically studied how developers deal with library upgrading that arises due to changes in APIs: the evolution occurs due to changes in the APIs of the library, with impacts propagating to dependent systems. Mitigation of the issues of library upgrading has been explored by conducting research on API evolution in software ecosystems, API migration, API compatibility, and API upgrading in historic repository releases.

Bavota et al. [2015] thoroughly investigate the evolution of dependencies and their APIs in the Apache ecosystem. The goal of their study is to understand what factors drive developers toward updating their dependencies. They find that a new release of a library with major changes usually triggers an upgrade in the dependent systems. Their results show that, at the API level, developers tend to upgrade a dependency when substantial changes such as bug-fixing activities are included. However, in most cases that involve API changes, developers are hesitant to perform the final transition. McDonnell et al. [2013] study the Android ecosystem to understand the impacts of API evolution in Android projects. They find that client adoption lags behind the exponentially growing API evolution. They discovered that in their studied projects 28% of library references were outdated. Developers did indeed upgrade to newer API versions, but the adoption time was found to be much slower than the average API release intervals, as the projects had a median lag time of 16 months. The findings of McDonnell et al. are further strengthened by Hora et al. [2018]: they study the impacts of API evolution on the Pharo ecosystem, centring on exploring the reaction of developers to the API evolution in the source code. They find that "53% of the analyzed API changes caused reaction in 5% of the systems and affected 4.7% of developers [...]".

Bogart et al. [2015] explore the stability of dependencies among modules of an ecosystem. Continuing this work, Bogart et al. [2016] compare API breaking changes in libraries belonging to three different ecosystems: Eclipse, R/CRAN, and Node.js/npm. They state the difference in the applied practices, policies, and tools when performing a breaking API change in these distinct ecosystems. They find that the three ecosystems differ significantly in their API cost negotiation

and community values. These studies explore the area surrounding API evolution and their consequences on its respective ecosystems. The focus of these studies is to determine the significance of changes in APIs while moving from an old version of the same library to a newer one.

Some research studies library upgrading by reviewing migration of APIs between similar library versions. For instance, Dig and Johnson [2006] study API migration between software systems, on two major releases of four frameworks and one library written in Java. They define a catalogue of breaking and non-breaking API changes that can be used by future researchers. In their research they find out that 80% of the changes that break client code were due to refactoring operations; Cossette and Walker [2012] however call this result into question. Hora and Valente [2015] propose a tool called apiwave, which keeps track of API popularity and migrations for popular frameworks and libraries; their tool notifies system maintainers regarding the evolution of APIs in the supported frameworks. Kula et al. [2018a] conduct a large scale study on library migration for GitHub projects. They report that 81.5% of the studied systems kept their outdated dependencies even when subject to the risk of security vulnerability; 69% of the interviewed developers were not even aware of the vulnerable dependencies; most developers do not prioritize dependency update as it means additional work for them. They conclude that updating deprecated dependencies is not common practice for most developers and projects. Kapur et al. [2010] study the issue of dangling references during API migration; they present a prototype tool called Trident that can be used to automate refactoring of references during migration. The available research is only restricted to migration of API belonging to different versions of the same library. The area related to migration across domains and distinct libraries is yet to be studied thoroughly.

Another take on library upgrading considers API compatibility issues. Xavier et al. [2017] study the breaking API changes in Java libraries and explored their impacts on client code. They find out that 14.78% of library releases broke compatibility with previous versions while impacting 2.54% of the clients in the process. Moreover, the projects with higher frequency of breaking changes are comparatively more popular, large, and active. They recommend that, to avoid the imminent risk of using obsolete libraries, developers update their system as early as possible. Jezek

et al. [2015] study the compatibility issues of Java APIs as the third party library evolves. The goal of their study is to locate impacts of API evolution on programs using these libraries. They find that API compatibility is a commonly occurring issue and can cause problems for programs depending on the library. They conclude that better tools and methods are in dire need to support this ever growing library evolution. These empirical studies express concerns regarding issues with API incompatibilities while updating a library to a newer version.

Researchers have also studied library upgrading in the history of releases in online repositories. Kula et al. [2015] study the latency of adopting new releases in Maven projects. They explore the adoption of new library versions based on the trust factor of functional and non-functional correctness. They find that 82% of the studied systems were inclined towards adopting the latest versions in favour of existing systems, in contrast to Bavota et al. [2015]. Visser et al. [2012] perform a historical analysis of stability and impact on clients of the Apache Commons library. They present four stability metrics based on method changes and removals. Their matrices can be used to calculate stability of implementation of third-party libraries. Raemaekers et al. [2014] study the means and approaches of notification about update option regarding breaking changes in libraries on Maven Central Repository, finding that the current mechanism to signal interface instability is not often used and requires further improvement. The research in this domain has studied library upgrading by exploring different releases of the same library in a repository.

These studies focus on exploring API evolution of different versions of the same library. These studies explore what changes drive the developers to opt for latest releases and the impacts of these changes. However, this research does not explore movement between distinct libraries. The questions of when and why to update a library to an alternative one still remains unanswered. The reasons forcing a developer to move to another community or alternative option, the impacts due to this decision, and the effort required to perform this transition are still not clear. Thus the library "upgrading" performed on an inter-level (i.e., library migration) requires further exploration.

Our research addresses some of these unanswered questions by studying library migration between independent libraries and explores the rationales behind such transformations.

3.2 Tool Support for Library Migration

Developers have to manually perform upgrading of software libraries, due to a lack of better alternatives. The task of modifying a software system to start using a new library is developer driven, error prone, and time consuming. The available literature shows that the current tool support for dependency management and upgrading is lacking.

Walker and Murphy [2000] pointed out that tightly coupling to external libraries can cause brittle APIs, but they presented only anecdotal evidence in support of it.

Bogart et al. [2015] argue that current collaboration tools work well within a single community but break down when used for inter-dependent projects between distinct communities. Their preliminary results show that in the studied ecosystems, developers struggled with dependency management and evolution and existing tools were rarely used to help the cause. Cossette and Walker [2012] study the incompatibilities of APIs between different versions of Java software libraries, discovering that most of the available recommendation techniques for API migration are incompetent, with an accuracy rate of only 20%. Their results show that most available recommendation techniques are insufficient for library migration.

Researchers have recently started to explore approaches from the field of visualization to help developers while mitigating library evolution. Kula et al. [2014] study the lack of historical knowledge regarding efficient maintenance of outdated libraries, wanting to assist maintainers by visualizing the evolution of relationships between a system and its dependencies. To materialize their idea they propose a time-series visualization plot called library-centric dependents diffusion plots (LDP): LDP incorporates knowledge of how different systems manage their dependencies. Their approach shows potential in understanding the dependency management in real world evaluated examples. Kula et al. [2018a] introduce the software universe graph (SUG) that can be used to model the popularity, adoption, and diffusion trends of dependencies in a repository or ecosystem. The SUG models the usage of a library over time and the latency of introducing new dependencies in system. The visualization provided by SUG can be used for extracting valuable insights from dependency management. Building on the findings of Kula et al., Todorov et al. [2017] propose

a tool named SoL Mantra (Software Library Mantra) that makes use of the knowledge available through wisdom of the crowd to suggest update opportunities to the developers; it checks which of the system dependencies are out of date and recommends which dependencies should be updated together or individually. By making use of the coexistence coefficient provided by Kula et al., SoL Mantra can also compute the complexity of each potential update. The tool extensively makes use of visualization to present the maintainer with a complex orbital layout[17] for the dependencies relation. Their visualization however gets messier in case of massive systems that have overlapping dependencies; such overlapping visualizations are very hard to comprehend for the user without having proper support. SoL Mantra only provides visual cues for update opportunities of different versions of the same library. It is helpful in an environment where a developer wants to update to a newer version of the same dependency; however, the tool is not built for the purpose of moving to another distinct library.

Another approach to upgrade libraries using visualization is proposed by Gerasimou et al. [2018] as an end-to-end methodology. Their approach makes use of a visualization technique called the city metaphor [Wettel et al., 2011], used to highlight the impacted areas of change as three dimensional cities associating the extracted dependency metrics with visual properties of city components. Their approach is a four step process that starts with parsing and analysis of the source code that needs upgrading. The second step visualizes the gathered information regarding dependencies metrics by utilizing the city metaphor. Steps 3 and 4 involve the automatic transformation of old source code to the upgraded one with its verification. The impacted areas are automatically mapped and transformed to certain template patterns which are then verified for accuracy. Their results remain preliminary.

[17] https://github.com/emeeks/d3.layout.orbit [accessed 2018/10/11]

3.3 Change Impact Analysis

Analogously to a stone thrown in a pond, an initial software change in a system can result in "ripple effects" that demand subsequent, secondary changes to the rest of the code base [Bohner and Arnold, 1996]. Therefore performing change impact analysis (CIA) is an essential step of incremental software development Rajlich and Gosavi [2004]. The origin of impact analysis can be traced back to Haney [1972], who described a simple model to estimate stability of a large system as a function of its internal structure. The proposed matrix was able to record the probability of change propagation in a system of significant size. The traditional definition of CIA comes from Bohner and Arnold [1996]: the process of identifying and estimating the areas of a system that need modification in order to accomplish a change.

As most changes in a system propagate due to program dependencies, several impact analysis techniques exploit models that utilize the system dependence graph (SDG). [Horwitz et al., 1990] proposes the SDG approach along with its two-phase graph reachability algorithm to compute inter procedural slices. An SDG is directed acyclic graph that utilizes program dependence graphs [Ferrante et al., 1987] to model the inter-procedural structure of a system under analysis. By building on top of the results from the Horwitz et al. algorithm, Reps et al. [1994] proposes an algorithm that is asymptotically faster then the original one. However, the asymptotic complexity of their algorithm is still bounded by $O(n^3)$ where n is the maximum number of parameters at any call site. Similarly other research [Liang and Harrold, 1998, Forgacs and Gyimothy, 1997, Sridharan et al., 2007, Graf, 2010] tries to improve the original proposed algorithm and model of SDGs by either reducing additional parameter vertices, supporting new functionality, or optimizing the graph traversal algorithm. Even with all these proposed advancements, constructing and maintaining SDGs or similar graphs for the whole system is too expensive for practical purposes [Men, 2018].

The available CIA techniques can be classified into two broad categories: structural analysis and semantic analysis. The structural analysis techniques construct dependence graphs in order to extract structural dependencies between program entities, working on the principle that, if a program entity in a structural dependency network changes, other dependent entities might also

have to change [Petrenko and Rajlich, 2009, Wen et al., 2010]. A call graph based approach is proposed by Arnold [1996], but it suffers from imprecise results as the call graph does not contain the impacts of callees on callers. In order to further improve accuracy, Badri et al. [2005] incorporate control flows into their model. Similarly different variants of structural approaches [Lee et al., 2000, Tonella, 2003, Breech et al., 2004, Jasz et al., 2008] have been proposed over the years. These techniques usually start with building an intermediate representation of the program based on the extracted structural dependencies. Change impact is then calculated based on the transitive closure and traversal of the constructed representation.

The second type of CIA techniques incorporate the semantics of the source code in their analysis. These techniques can distinguish between changes that have local (i.e., refactoring) or substantial (i.e., functionality) impacts. The semantic CIA approaches recommend probable changes by exploiting semantic rules and patterns of the source code. Poshyvanyk et al. [2009] try to overcome the limitations of static coupling measures by capturing the conceptual relation between program entities. Their approach uses latent semantic indexing for detecting the similarities between code entities. Gyori et al. [2017] proposed an inter-procedural semantic CIA that was based on equivalence relations between program versions. Their technique improves precision in the presence of semantic preserving changes. Several other approaches [Robbes et al., 2007, ten Hove et al., 2009, Canfora and Cerulo, 2005, Ajienka and Capiluppi, 2016] have been proposed over the years, that perform fine-grained CIA by utilizing code semantics. These approaches compute semantic similarity for embedded text information in the analyzed artifacts.

The semantic approaches for CIA are more thorough and they can even discover couplings that are harder to find for the traditional techniques. These approaches however consume significant resources in order to achieve their results, being too slow when applied on industrial scale. On the other hand, structural techniques are comparatively faster and have more feasible applications in the industrial domain. Moreover, fine-grained approaches have been proposed to remedy the issue of lower precision and accuracy in structural techniques. However, the structural techniques have the issue of producing overestimated results due to their undecidability [Landi, 1992].

3.3.1 Incremental change

The process of incremental change (IC) is a systematic approach to introduce new functionalities in an existing system. [Rajlich and Gosavi, 2004] propose an incremental change model to support evolution in software systems. Their model is divided into three broad phases namely initiation, design and implementation of incremental change. The initiation phase include interaction with the customer or stakeholders to initiate the change request. The implementation phase that comes after design deals with introducing the actual change in the system by performing several activities. These activities are pre-factoring, actualization, change propagation, post-factoring, and testing.

The design phase involves concept location and impact analysis that are to be performed before the actual implementation. Concept location allows the developers to determine the initial location of change within the source code. If effectively followed, concept location can successfully determine the areas where change is to be made [Marcus et al., 2005]. On the other hand, the activity of impact analysis identifies the full extent of an initial change. Buckner et al. [2005] propose JRipples in order to partly automate the IC cycle of Rajlich and Gosavi [2004]; it is an interactive tool for program comprehension during IC that supports activities of impact analysis and change propagation. JRipples requires developer to discover at least one class of the impact set. The rest of the impact set is discovered by looking at the neighbours of the dependency graph with an iterative approach. In order to calculate impacts, JRipples supports different propagation rules (marks) that determine how change in one node is propagated to other dependent ones. An "Impacted" mark indicates that the class would be changed during the IC. A "Propagating" mark indicates that the class itself will not be changed but the neighbouring nodes might be affected. Finally, a "Next" mark indicates that the marked class might change as consequence of changing the neighbouring node. The JRipples analysis ends when there are no longer any classes that are marked as "Next".

JRipples is built to support IC, after the developer has located the initial impact points in the code base; it does not specifically assist in library migration. As demonstrated in Section 2.1, its use results in error-proneness and repetition that can waste time and resources: in essence, the

developer misses low-level details in migration, due to information overload.

3.4 Summary

We discussed the existing work related to software library upgrading, and pointed out how that phenomenon potentially differs from library migration; thus, the literature contains little knowledge of the issues that arise in library migration.

Similarly, available tool support for library upgrading relies on visualization to assist the developer in transformation. These tools only provide visual cues and feedback to the developers. None of the available tools and approaches is appropriate for efficiently performing library migration.

We also discussed the available literature on change impact analysis. Semantic approaches are quite costly in contrast to their purely structural counterparts. However, as a consequence of their higher cost they are more accurate. The approaches based on SDG and similar graph representations are expensive to construct and maintain; this generally makes them too expensive for practical use. We also highlighted the systematic approach of incremental change to support software systems evolution. JRipples, a tool for incremental change, can support a manual approach for performing library migration, but this approach would require a developer to manually locate classes, members, and impact points relying on the old libraries being removed, before calculating their ripple effects; something more is needed.

Chapter 4

Data Collection and Analysis

In this chapter, we address our research questions on software library migration. Our aim is to empirically explore the impacts of library migrations on Java projects and their transitively dependent systems. We use the terms "dependency" and "library" interchangeably in this chapter; they refer to an external system or JAR file which was added to a project in order to fill the void for certain functionality. For simplicity we refer to both replacement and outright removal of a library from a system under study as "library migration".

Our research questions are as follows.

RQ0: How often does library migration occur?

RQ1: How often does library migration in a software system break it?

RQ2: What means are used by developers for notifying about library migrations?

RQ3: What impact does library migration from a system have on its dependent clients?

RQ4: How much effort is required for mitigating library migration?

To consider certain narrower aspects, we subdivide some of the research questions as follows.

RQ1.1: When do developers remove an external library without adding a new one?

RQ2.1: What are the rationales that drive developers to migrate libraries in a working system?

Our approach is overviewed in Figure 4.1. The process conforms to a pipe-and-filter architectural style, where each step transforms its input to its output that is used for the input to the subsequent step. The rectangular boxes constitute the processing steps performed for our study (the greyed ones are described further); the curved-bottom boxes represent the data passed to/from the processing steps; the projected cylinders represent databases. The solid arrows represent the sequencing of the processing steps with the data passed therein (note that the apparent cycle is only an artifact of the representation); the dashed arrows represent additional dependencies. The solid circle at top represents the start of the process, and the target symbol at bottom represents the end of the process.

A total of seven processing steps are performed for the sake of finding and analyzing data relevant to our research questions. (1) Our custom crawler performs a GitHub search for relevant projects by generating a series of queries; the search accesses the GitHub repository and it determines metadata identifying the projects relevant to the queries. The metadata is then used by the crawler to download the files stored in GitHub for the relevant projects (see Section 4.1.1). (2) We use a custom analyzer to analyze the downloaded project source code for occurrences of library migration, filtering out projects that have not (potentially) undergone library migration (see Section 4.1.2). (3) We determine a random subset of these filtered projects through a manual selection process, and obtain the binary code for the filtered subset through access to the Maven Central Repository (see Section 4.1.3). (4) An industrial tool, Find it EZ, is used to determine initial points of impact involved in the library migrations (see Section 4.2.1). (5) A second industrial tool, JAPICC, is used to determine whether the library migrations have caused any breaking changes to the projects' exposed APIs (see Section 4.2.2). (6) A third industrial tool, Code Compare, is used to determine whether the changes that occurred broke the code or not (see Section 4.2.3). (7) We collected and analyzed the resulting data (see Section 4.3).

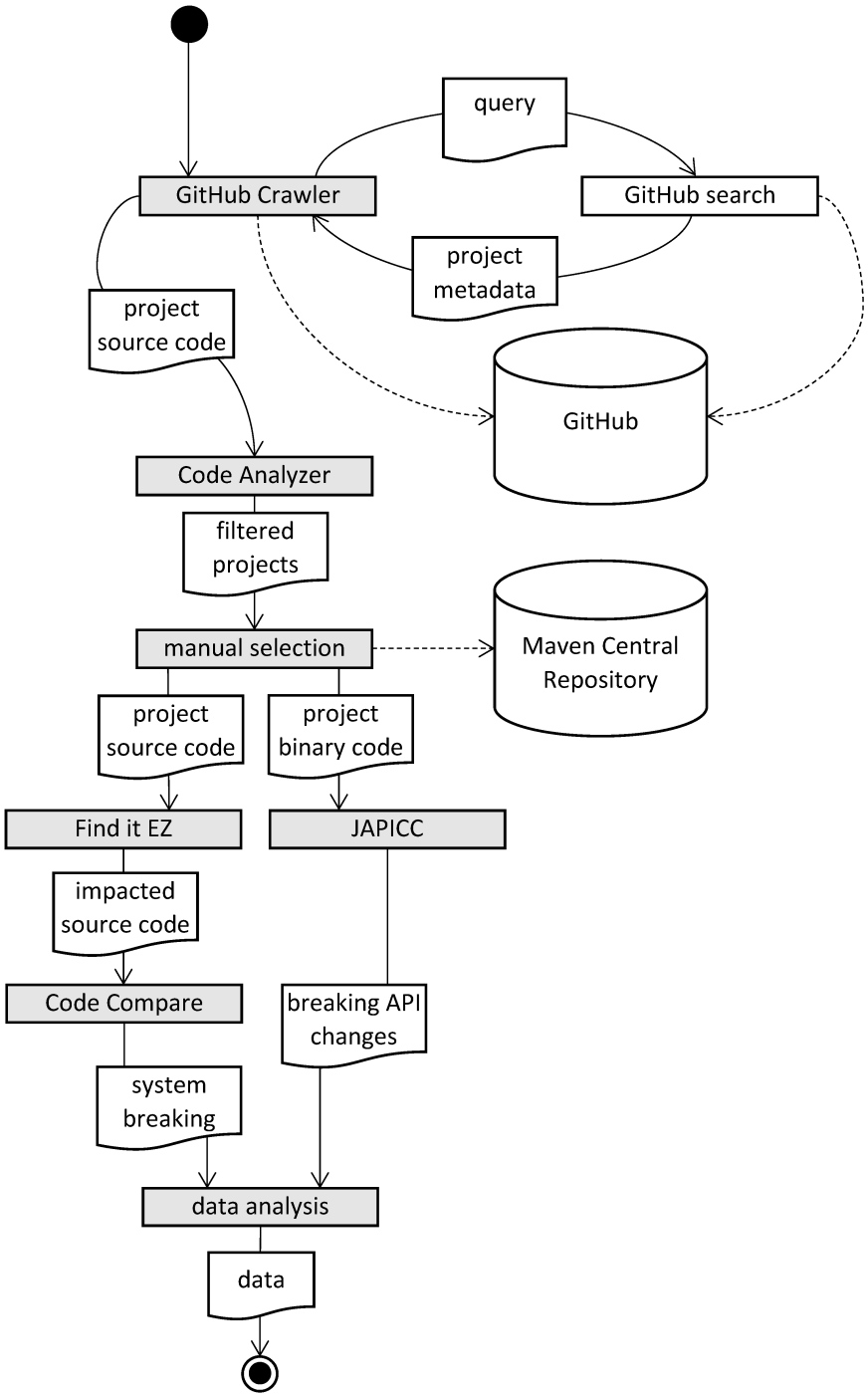

Figure 4.1: Approach for data collection and analysis.

4.1 Selection of Projects

We used GitHub, a web-based hosting service that supports the version control system Git, to retrieve the systems used in this study. GitHub allows its users to access a large number of public repositories and is considered to be the largest host of open source code in the world [Gousios et al., 2014]. We selected GitHub due to its unrestricted access to commits, developer conversations, and source code. GitHub is home to open source projects that are currently written in a total of 337 unique programming languages[18]; however, we narrow our study to consider only Java, the third most popular language on GitHub, for the sake of avoiding the engineering complexities in handling additional languages. Furthermore, in order to automate extraction of dependencies, we restrict our studied systems to Java-based Maven and Android projects. Both of these platforms contain XML-based, central code management files (named POM.xml and Build.xml for Maven and Android projects, respectively) containing general information about the project with their configuration details that are used during compilation. More specifically, these files contain information about any external library that a project depends on.

4.1.1 GitHub Crawler

We wrote a Java crawler that utilizes GitHub search to identify and download relevant projects. GitHub offers representational state transfer (REST) APIs[19] for helping researchers to automate the tedious process of fetching large code bases. GitHub APIs allow the user to select various parameters (e.g., star interval and language) to limit the search. The number of stars corresponds to the popularity of a project; we decided to only download projects that had 100 stars or more.

In order to protect themselves against denial-of-service (DOS) attacks, GitHub limits the API hit rates per user: 5,000 hits per hour when using authenticated access and 60 otherwise. We adapted to this limitation by making use of periodic sleeping, reduced star interval, and access tokens; the access tokens provide the authentication needed to permit 5,000 hits per hour, while

[18] https://octoverse.github.com/ [accessed 2018/10/11]
[19] https://developer.github.com/v3/? [accessed 2018/10/11]

the periodic sleeping resets the current rate limit window. The crawler generates queries with increasing star intervals, in order to avoid hit overload and for feasibly managing the retrieval and download of relevant projects. The crawler generates a query for each interval difference of 100, starting from 100 until it reaches 27,000 (i.e., it starts from the interval $[100, 200]$, then $[201, 300]$, and so on). We stopped at 27,000 as we manually verified that the most popular relevant project at that time had 26,220 stars. The metadata of projects returned from the API is in paginated form; therefore, the crawler periodically increases the page number starting from 1 until it reaches a page with 0 projects. The metadata of retrieved projects contains a set of downloadable links for each of their released versions; the crawler uses these links to download the source code of the projects for a star interval. The first query generated by our crawler is: "https//api.github.com/search/repositories?access_token=token&page=1&q=language:Java%20language:Maven%20language:Android%20stars:100..200". Other queries generated by our crawler are similar; only the star intervals and page numbers are adjusted for retrieving the relevant projects.

The crawler was used to download a total of 9,561 projects in May 2017, the complete set of projects that satisfied our queries at that time.

4.1.2 Code Analyzer

We wrote a Java-based code analyzer to filter the downloaded projects. Selected projects were required to have migrated between libraries in any two successive released versions. We tried different approaches for our analyzer including the use of change logs, release notes, version numbers, and different granularities, but such approaches resulted in excessively high false positive rates; therefore, we reverted to the simple approach of checking the absence of prior dependencies in newer versions. Our simple algorithm for the code analyzer is shown in Figure 4.2.

The algorithm iterates through the located projects. Assuming that the versions of the project are ordered chronologically, it then iterates through them also locating each version's successor (the last version is not selected as it does not have a successor). It locates the relevant XML configuration

for all $p \in Projects$ **do**
  for all $v_i \in \text{Versions}(p) \setminus v_n$ **do**
    $xml_i \leftarrow \text{locateXMLFile}(v_i); xml_{i+1} \leftarrow \text{locateXMLFile}(v_{i+1})$
    $DEP_i \leftarrow \text{Parse}(xml_i); DEP_{i+1} \leftarrow \text{Parse}(xml_{i+1})$
    return $DEP_i \setminus DEP_{i+1}$
  end for
end for

Figure 4.2: Algorithm to detect elimination of external dependencies.

file in both versions, which are parsed to find the absence of an external dependency in the newer version that was present in the older one. The projects demonstrating this phenomenon are collected for further analysis. (Note: The algorithm will locate the XML configuration file and parse it two times in total for versions 2 through $n - 1$ of each project. As the necessary processing has low runtime cost, we chose this naive approach over the alternative of caching the information for the successor version, which would have increased implementation complexity and hence the potential of introducing bugs. Since the runtime cost of the algorithm is negligible relative to the high cost of locating and downloading the projects, optimization of the algorithm was not warranted.)

We ran our analyzer on the downloaded 9,561 projects, returning a total of 1,135 projects which consisted of 680 Android and 455 Maven projects.

4.1.3 Manual Selection

We manually selected a subset of the downloaded projects that fulfilled the following set of criteria.

- Each selected project must be actively using a library, in its older version, that was found missing in the newer one.
- Each selected project must migrate between libraries not developed by the same developers. (Maven projects follow a modular structure; therefore, there are many intra-modular dependencies in which the internal modules act as dependencies for each other in a Maven system. The developer has control over these modules; therefore, we decided to ignore such dependencies.)

- Each selected project must have been actively updated in the last 2 years. (We wished to avoid stale or abandoned projects.)
- Transitive dependencies which are either used by the IDE or are required transitively by other external libraries cannot satisfy the other criteria.

We found 218 projects that fulfilled the criteria. We randomized the list of these filtered projects by using the RAND function provided by Microsoft Excel. Our use of RAND assigns a random number between 0 and 1 to each project; the projects are then sorted on these numbers to obtain their randomized list. We began by selecting the first 120 projects from the randomized list that fulfilled our criteria; when we discovered that six of these projects involved only trivial migration changes, we eliminated them as well, resulting in a final project count of 114. There were 74 Maven projects and 40 Android ones. We obtained the binary code for these projects by manually accessing the Maven Central Repository. The studied projects are listed in Appendix A. Appendix B lists the involved third party libraries for the studied projects.

4.2 Data Gathering

To study migration of external dependencies, we collected information on: (1) impacts on projects undergoing library migration; and (2) impacts on transitively dependent client systems after library migration in a project they were depending on. We decided to use tools for data extraction where possible, as a manual approach would have been error-prone and time-consuming for the scale of our study. Our process of gathering data involved the following steps.

4.2.1 Find it EZ

In order to answer our research questions, we had to locate the directly impacted classes of the library being migrated. Find it EZ is a software productivity tool that is language-independent and can be used for change impact analysis on an industrial scale[20]; thus, it could be applied for our

[20] https://www.finditez.com/ [accessed 2018/10/11]

needs and a community version is available free of charge.

Find it EZ requires a search query to locate the dependent classes in a project. We decided to use the unique common parent package of a library for this search query. The package name is available as an import reference in any class that depends on the library; hence there is no possibility of missing a dependent class during the search, meaning that we avoid false negatives assuming we made no errors. We manually compiled this package name for each library and then used it for searching the impacted classes through Find it EZ. We then manually examined each of the reported impacted class during our analysis; hence, there is no possibility of having a false positive in our final set of impacted classes, assuming we made no errors.

4.2.2 JAPICC

In order to determine the consequences of library migration on a project's dependent client, we decided to explore compatibility issues between binary code of each project's newer and older versions. Any client relying on a project's older version will not run nor compile with the projects' newer version, if there are backward compatibility issues between successive versions of a project. For our analysis, we only examined the compatibility issues resulting from library migration in the project.

In order to achieve our goal we used Java API Compliance Checker (JAPICC),[21] a tool for checking backward compatibility of a Java library API. The tool checks class declarations of the older and newer versions, to analyze changes that may break compatibility for a client depending on the older version of a system. The tool reports compatibility issues that can arise for a client that tries to use the newer version of the project it is depending upon. JAPICC reports the error messages of the Java Virtual Machine and compiler that would be generated for each breaking change found in the API. During our analysis, we only considered those breaking changes in the report that resulted from library migration.

[21] https://github.com/lvc/japi-compliance-checker [accessed 2018/10/11]

4.2.3 Code Compare

We had to determine the severity and type of impact resulting from library migration in a project. In order to achieve this, we had to flag the discrepancies resulting from migration between the impacted classes of a project. We decided to use a diff comparison tool Hunt and McIlroy [1976] to highlight these discrepancies. We used Code Compare, an industrial-scale diff comparison tool,[22] on the impacted classes for the versions involved in migrations; Code Compare allows the user to view a line-by-line comparison between two different projects or files. The diff comparison flagged the discrepancies between successive releases. We examined the discrepancies that resulted from the migration of library, which was then used to determine the breaking of system's code.

We only considered changes in functionality that break system code. By "break system code", we mean that the code had compilation errors after migrating the system from a library. We decided to ignore cosmetic changes (i.e., package and method name changes) that had trivial impact on source code while answering our questions. If such changes are considered, all studied migrations might break the source code, e.g., changing the name of a package being used might result in compilation errors but correcting such errors requires trivial effort.

4.3 Analysis and Results

In this section, we address the research questions based on the collected data.

RQ0: How often does library migration occur?

We found that 1,135 out of the total 9,561 studied projects underwent library migration (11.87% of the total studied projects). These 9,561 projects were retrieved through our GitHub crawler in May 2017; recall from our earlier description that these were all the Java-based projects (in Maven and Android) accessible via GitHub, with a popularity rating of at least 100 stars, which represents the minimum number of developers that are interested in using the system. Thus, many projects

[22] https://www.devart.com/codecompare/ [accessed 2018/10/11]

undergo library migration, and it is not a rare phenomenon. We cannot speak to the frequency of library migration in nascent projects, or in projects that are unpopular for other reasons.

We found exactly one case in which a project underwent library migration *twice*: JStorm migrated from Google JSON Simple to Fast Json and then reverted back to Google JSON Simple when the developers were not content with the results of the initial migration.[23] Our speculative explanations for the lack of a second migration include: none of the projects has been in existence long enough to experience the need for a second library migration; the developers' choices about how to perform the initial library migration avoided the need for a second library migration; the first library migration was sufficiently painful for the developers that they avoided doing it again.

RQ1: How often does library migration from a dependent system break it?

We found that 67.3% of library migrations broke the dependent system. This shows that dependencies are often coupled with the system and moving them out can have a significant impact. However, we also found that 32.7% of migrations did not break the dependent system completely, having minimal impact. The reasons behind successful library migration without code breaking are listed in Figure 4.3 and described in detail below.

[23] https://github.com/alibaba/jstorm

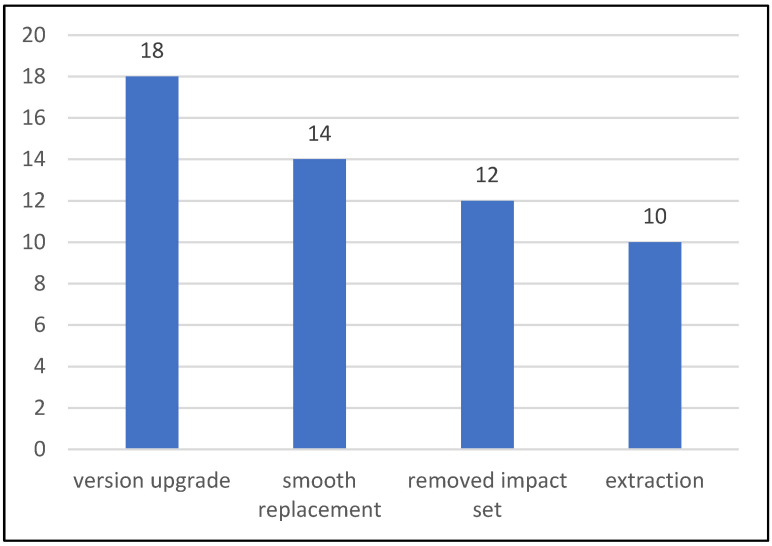

Figure 4.3: Project count versus reasons behind successful library migration without code breaking.

- **Version upgrade**

  We found that 18 projects depended on libraries that changed their product name (i.e., package name) after a major release. This modification was due to alteration in ownership, developers, or hosting platform. However, apart from these cosmetic changes, the new releases for such libraries were mostly backward compatible. As a result, the dependent system was easily repaired.

  For example, Jackson is a popular high performance JSON processor for Java. Before 2011 it was being maintained at Codehaus, a prominent open source Java project community and hosting platform a decade ago; however, after the increasing popularity of GitHub, its influence started waning. Due to this the Jackson developers decided to shift their project to GitHub. They released Jackson version 2.x which changed the name of their parent package from codehaus to fasterxml. Moving from Jackson 1.x to 2.x in most cases (depending on API usage) only requires a change in package names.

- **Smooth replacement**

  We found 10 libraries (shown in Table 4.1) that support smooth replacement (i.e., with minimal code impact). These libraries were involved in 14 smooth migrations in our study, as three of them (Asynchronous Http Client, NineOldAndroids, and Picasso) were involved in more than one library migration. The pairs of dependencies (Google Guice Core Library and Javax Inject, Guava and Guava Mini, Retrofit and OkHttp) have the same developers and allow easy replacement with each other.

  For example, Javax and Guice are both dependency-injection frameworks. Javax is provided by standard Java with standardized dependency injection functionality while GUICE is an external dependency. Javax annotations can function as direct replacements for Guice with little effort. The author of Guice (Bob Lee) is a spec-lead for the Javax team; thus, the basic infrastructure for both libraries is similar. Moving from Javax to Guice is simple since Javax is a subset of Guice, while moving in the opposite direction might require deletion of certain

Table 4.1: Libraries that support smooth replacement with another one while having minimal code impact.

From	To
Jackson Dataformat YAML[24]	SnakeYAML[25]
Asynchronous Http Client by com.ning[26]	Asynchronous Http Client by org.asynchttpclient[27]
Google Guice Core Library[28]	Javax Inject[29]
Apache Log4j API[30]	SLF4J API[31]
Guava[32]	Guava Mini[33]
Plexus Archiver Component[34]	Google Guice Core Library[35]
Apache HttpComponents Client for Android[36]	HttpClient Android Library[37]
NineOldAndroids[38]	standard Android SDK[39]
Retrofit[40]	OkHttp[41]
Picasso[42]	Glide[43]

functionalities.

In cases where library migration is the last resort, developers should first consider an alternative that can accommodate smooth replacement.

- **Removed impact set**

For 12 systems, the module dependent on the library to be removed, itself was discarded.

[24] https://mvnrepository.com/artifact/com.fasterxml.jackson.dataformat/jackson-dataformat-yaml [accessed 2018/10/11]
[25] https://mvnrepository.com/artifact/org.yaml/snakeyaml [accessed 2018/10/11]
[26] https://mvnrepository.com/artifact/com.ning/async-http-client [accessed 2018/10/11]
[27] https://mvnrepository.com/artifact/org.asynchttpclient/async-http-client [accessed 2018/10/11]
[28] https://mvnrepository.com/artifact/com.google.inject/guice [accessed 2018/10/11]
[29] https://mvnrepository.com/artifact/javax.inject/javax.inject [accessed 2018/10/11]
[30] https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-api [accessed 2018/10/11]
[31] https://mvnrepository.com/artifact/org.slf4j/slf4j-api [accessed 2018/10/11]
[32] https://mvnrepository.com/artifact/com.google.guava/guava [accessed 2018/10/11]
[33] https://mvnrepository.com/artifact/com.github.davidmoten/guava-mini [accessed 2018/10/11]
[34] https://mvnrepository.com/artifact/org.codehaus.plexus/plexus-archiver [accessed 2018/10/11]
[35] See footnote 28.
[36] https://mvnrepository.com/artifact/org.apache.httpcomponents/httpclient-android [accessed 2018/10/11]
[37] https://mvnrepository.com/artifact/cz.msebera.android/httpclient [accessed 2018/10/11]
[38] https://mvnrepository.com/artifact/com.nineoldandroids/library [accessed 2018/10/11]
[39] See footnote 15.
[40] https://mvnrepository.com/artifact/com.squareup.retrofit/retrofit [accessed 2018/10/11]
[41] https://mvnrepository.com/artifact/com.squareup.okhttp3/okhttp [accessed 2018/10/11]
[42] https://mvnrepository.com/artifact/com.squareup.picasso/picasso [accessed 2018/10/11]
[43] https://mvnrepository.com/artifact/com.github.bumptech.glide/glide [accessed 2018/10/11]

These modules were so tightly coupled with the dependency that developers preferred to completely rewrite the module instead of modifying the existing code. Since the complete impact set (including transitive impacts) using the dependency was removed, there is no question of breaking the code. For most of these migrations the module was already segregated and had minimal coupling with other parts of the project.

For example, the project OpenZipkin dropped their support for elastic search native transport. The release stated that the user base had no objection to removing the native support and doing this would allow reduction in size[44]:

> Most tools in the Elasticsearch world use HTTP as means to insert or query data. Before, we supported storage commands via the HTTP transport (port 9200) or the native transport (port 9300). Polling users, we found no resistance to removing the native transport. By removing this, we deleted 3.6K lines of code and simplified the dependencies of the project, allowing for easier maintenance moving forward.

- **Extraction of code**

  We found that 10 projects depended on only a small part of a library. Their developers decided to drop the library in favour of extracting the required classes from it as-is. They integrated this extracted part into their system. This extraction made the project more approachable for clients by reducing the total external dependencies of the system. This extraction was only performed for libraries whose license was lenient enough to allow it, e.g., certain licenses only allow the use of the software as a complete JAR and restricts the user from performing reverse engineering of any kind. Since the required part was extracted as-is including the packages, this change did not break the code.

  For example, the developer of checkstyle decided to extract from Apache Commons Lang since it only depended on a small part of it[45]:

[44] https://github.com/openzipkin/zipkin/releases?after=1.23.1 [accessed 2018/10/11]
[45] https://github.com/checkstyle/checkstyle/issues/2428 [accessed 2018/10/11]

> We need to remove the dependency, as right now we depend only on util fields of that library, we could copy them to our CommonUtils.java class. Dependency should be removed.

The results show that, in many projects, library migration can break them. Key exceptions involve situations in which the migration is between closely related libraries, where the dependencies are completely re-written, or where the limited functionality needed is extracted from the library.

RQ1.1: When do developers remove an external library without adding a new one?

Developers usually remove a dependency in favour of an alternative one; however, for 43.6% of migrations, libraries were removed from a project without adding a newer one. The results show that a considerable number of developers decided not to add a dependency after migration. Figure 4.4 shows the reasons behind complete removal of a dependency from a project.

For 37 migrations, developers dropped a dependency outright due to the availability of a better alternative offered by a standard Java software development kit (SDK). An SDK is typically a set of software that is shipped along with the platform and does not require any extra addition for functionality usage. The benefit of using an SDK includes reduction in size and external

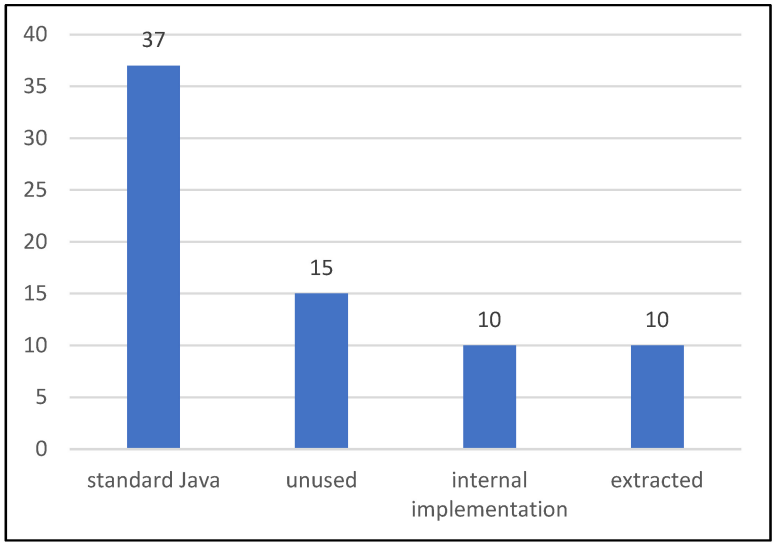

Figure 4.4: Project count versus reasons behind complete removal of dependency from a project.

dependencies, better support, and stability. For example, checkstyle decided to drop dependency on Guava since JDK 8 started providing better support[46]:

> Since now we use Java 1.8, it[']s worth cutting down on checkstyle's dependencies on Guava library: Guava's Optional replaced with Java's native; Guava's Predicate and Iterables should be replaced with Java's Predicate and streams.

For 15 migrations developers discarded a library when there was no further need of the functionality provided by it. This involves a change in approach or a drop of support for certain functionality that was previously provided by the system. For example, the Hangout project provided various common utilities including input/output/filter to the user for Logstash, an open source tool for collecting, parsing, and storing logs for future use. They provided support filter for the Jinjava template engine; however, later they decided to drop support for this filter and removed the complete package along with it.

For 10 projects, the developers preferred to write their own in-house module instead of using an external library. The in-house development was motivated by customization and control of the module in future releases, which benefits are not obtained from an external library, since the purpose of using one is to use the library as-is in order to save resources. For example, FXGL, a game development framework, decided to drop using Ehcache in favour of developing an internal cache, as Ehache was creating random memory leaks which were very difficult to detect.

For 10 projects, the developers removed the external library after extracting the required part from it. They integrated the extracted part into their system. For example, Zipkin, a distributed tracing system, removed their dependency on Okio as they were only using one class from it. They extracted the Base64 class from Okio and integrated it into their system.[47]

The results show that around half of the projects preferred to completely drop an external library when they had an alternative solution that did not require using a dependency.

[46] https://github.com/checkstyle/checkstyle/issues/3433 [accessed 2018/10/11]
[47] https://github.com/openzipkin/zipkin/pull/1245 [accessed 2018/10/11]

RQ2: What means are used by developers for notifying about library migrations?

41.2% of migrations were mentioned and discussed in release notes: only 19% of the Android projects mentioned their migrations in contrast to 43% of the Maven projects. These numbers are surprisingly low, since migrating a library can directly impact the client using the dependent system.

In order to dig further into this question, we manually analyzed the pull requests (PRs) of the studied projects. GitHub PRs allow developers to discuss and review potential changes within the community before pushing them to the repository. We found that 59.39% of migrations were discussed by the developer in PRs. The individual numbers also increase to 62.38% and 55% for Maven and Android, respectively. This relatively low ratio shows that developers do not follow a thorough process before performing a migration. The number (98) of migrations discussed in PRs is greater than the number (68) found in release notes, suggesting that developers tend to preferentially use internal threads as a means of notification while moving away from a dependency, in contrast to release notes. However, these discussions are mostly not reviewed by general clients who are only concerned with the release notes and the final product release; there are still a large number of migrations that are not discussed at all. As in the studied projects, 40.61% of migrations were performed by the developers without being mentioned or discussed in any of the explored sources.

We conclude that release notes are under-used for notification of dependency migrations. In contrast developers prefer to use pull requests or other such internal threads, but this apparently remains insufficient considering the large numbers of migrations without any discussion or notification whatsoever.

RQ2.1: What are the rationales that drive developers to migrate libraries in a working system?

We manually classified the rationales behind library migration into 4 mutually-exclusive categories listed in Table 4.2. These categories were selected based on the motivation behind the change as explicitly stated in the sources of information available to us. These rationales were motivated by post-migration improvements (i.e., optimized size, performance, scalability, etc.) in the system.

Table 4.2: Mutually exclusive classification of rationales behind library migration.

Rationale	Maven	Android	Total
Moving to a better alternative	61	26	87
Intention to remove external dependency	39	0	39
Improving API	0	23	23
Unknown	9	7	16

We found that 52.72% of migrations were motivated by the availability of a "better" alternative. In order to explore the criteria of what constitutes a better alternative according too developers, we further classified the implicit advantages of introducing a new library. Figure 4.5 lists these subcategories in descending order of the number of projects.

41.3% of the alternative libraries were selected based on better performance, functionality, resources consumption, and future scalability. For example, in the Metrics project, the developer replaced Google Caliper with Oracle JMH as JMH provided better functionality and performance than its counterpart. The developer of the PR for this migration posted the following[48]:

[48]https://github.com/dropwizard/metrics/pull/394 [accessed 2018/10/11]

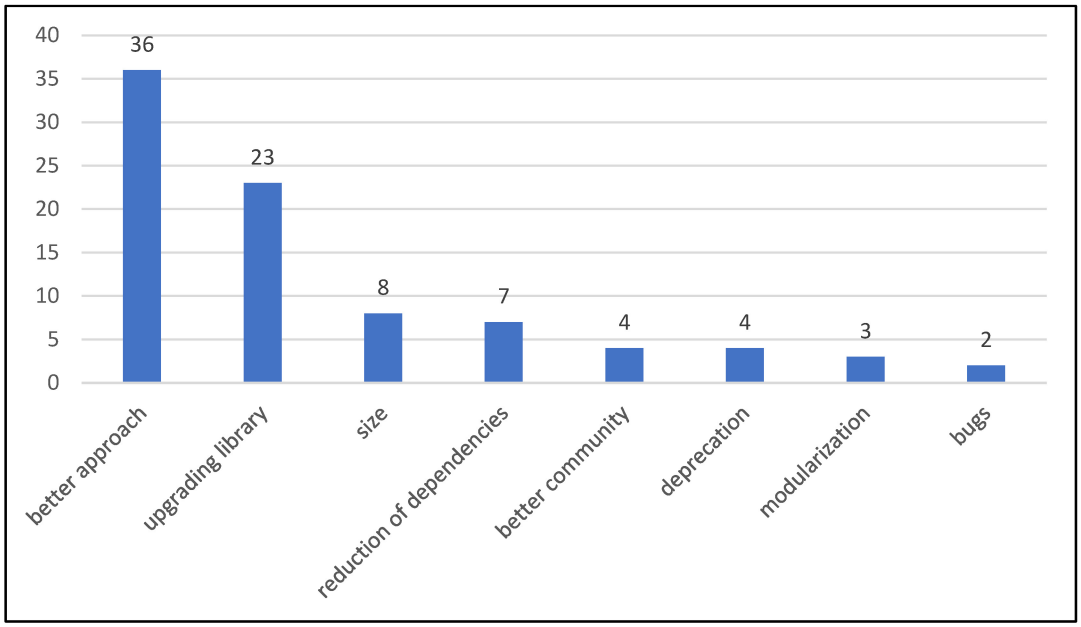

Figure 4.5: Project count versus alternative library selection criterion.

> There is a recent tool from [the] Oracle Java Virtual Machine engineers Oracle JMH, which is used for JDK benchmarking and verification. It has the advantage over Caliper by options for organizing multi-threaded benchmarking.

The owner of the repository replied[49]:

> This is awesome. I hadn't heard of JMH, and it resolves my long-standing complaint about how hard it is to produce a contended environment in Caliper. That said, I'd rather wait until JMH is available in Maven Central [Repository] to drop Caliper.

The rest of the alternatives were selected based on evolution of dependencies, reduction in total dependencies by unification or dropping extra libraries, size, better community, deprecation, avoiding bugs, and module segregation.

Secondly, we found that 23.3% of migrations were motivated by an intention to remove external dependencies from the system. These library removals were motivated secondarily by internal implementation, reduction in size, elimination of external dependencies, improvement in the Java SDK, and removal of unnecessary functionality. For example cron-utils removed various dependencies as their usage was affecting the clients. One of the developers wrote[50]:

> We would prefer to remove dependencies that are not required to provide the functionality we support up to that moment. The goal is that anyone can use cron-utils without having to download many transitive dependencies, which is most important for mobile applications. Another goal is to avoid downloading the same library twice due to version conflicts.

Thirdly, for the Android projects, 41.1% of libraries were migrated to improve the minimum API level of the application software (i.e., app). API level is an integer value that uniquely identifies the framework API revision supported by a version of the Android platform. Each year a new Android OS and API level is announced by Google along with deprecation and dropping of older ones.

[49] https://github.com/dropwizard/metrics/pull/394 [accessed 2018/10/11]
[50] https://github.com/jmrozanec/cron-utils/issues/259 [accessed 2018/10/11]

For example, for the project Circular Reveal, the developers removed deprecated dependencies to improve the minimum API level. The developer of the PR wrote[51]:

> IMO, anything supporting below API 14 is a waste of time and resources - NineOldsAndroid dependency could be removed and save us thousands of method calls and kilobytes.

The NineOldAndroids library description states that new applications should instead rely on API level 14 directly. They have stated this in their description as:

> NineOldAndroids is deprecated. No new development will be taking place. Existing versions will (of course) continue to function. New applications should use minSdkVersion="14" or higher which has access to the platform animation APIs.

These findings show that the Android community is strongly concerned about software evolution. This is due to the fact that the use of external libraries with deprecated API levels usually results in an application that will not work with the latest OS.

Finally, for 16 projects, we were unable to locate a rationale behind their library migrations from the considered sources. These changes went against the norms that were followed in other projects. Due to this we classified them as changes performed due to unknown rationales.

For example in JStorm the developers moved from GSON to Fastjson, despite the fact that GSON is more widely used in the community and has much better performance in contrast to other available alternatives. We note that documentation and support for Fastjson was available in the Chinese language and the developers of JStorm were native Chinese speakers.

The explored sources and our results depict that migrations of software libraries are usually driven by valid rationales, rarely attributable to whimsy.

RQ3: What impact does library migration from a system have on its dependent clients?

22.42% of projects broke dependent client code as a result of library migration. JAPICC distinguishes binary code incompatibilities into three severity levels: high, medium, and low. Low-severity problems can be considered warnings and do not affect the final compatibility verdict of

[51] https://github.com/ozodrukh/CircularReveal/issues/63 [accessed 2018/10/11]

Table 4.3: Errors and types that impact client code after library migration.

Type	Error
Signature	This method has been removed because the return type is part of the method signature. A client program may be interrupted by a NoSuchMethodError exception.
Superclass method or field	1) Access of a client program to the fields or methods of the old superclass may be interrupted by a NoSuchFieldError or NoSuchMethodError exception. 2) A static field from a superinterface of a client class may hide a field (with the same name) inherited from new superclass and cause an IncompatibleClassChangeError exception.
Class removal	A client program may be interrupted by a NoClassDefFoundError exception.
Exception	A client program may be interrupted by an added/removed exception.

the tool. The 37 breaking changes were given a verdict of incompatibility, and thus had at least one medium- or high-severity problem. These projects are listed in Appendix C.

Table 4.3 summarizes errors and types that impacted client code after library migration. The breaking of client code occurred due to discrepancies in either data types or methods of a system. 81% of the problems with data types were due to removed classes and methods. Due to strong coupling, developers preferred to revamp the dependent segments from scratch while using the new alternative library. JAPICC also detected discrepancies in the exceptions that were being thrown from methods or classes. The updated method either removed an exception or introduced an alternative one, as the old exception class was being used from the original library.

None of the non-Maven Android projects broke client code. This indicates that the studied Android projects were loosely coupled with their external libraries. We speculate that this behaviour of Android projects can be explained by the limited domain of external libraries being used currently. The result can also be attributed to the expected usage of the studied Android projects: in general, most Android apps are used as an Android Package Kit (APK) on a mobile phone; they are not developed to act as an external dependency for other Android systems.

The results show that library migration can propagate impacts to transitively dependent clients.

Table 4.4: Percentage of impacted classes.

Platform	Mean	Median
Maven	8.6	3.8
Android	15.5	8.8
combined	9.3	6.9

Table 4.5: Percentage of impacted LOC.

Platform	Mean	Median
Maven	12.1	5.58
Android	21	10.1
combined	15	7.5

This occurs only if the dependee project is using the migrated library in their API or exposed code. Migration forces the developers to break their API contracts and causes backward incompatibility issues for the clients.

RQ4: How much effort is required for mitigating library migration?

To estimate the effort required by developers in mitigating library migration, we determined the impacts of transformations at two granularities: affected classes and affected lines of code (LOC). In order to calculate these, we located the affected classes via Find it EZ (for the initial impact point) and the developer's commit (for any secondary impacts); the affected LOC were obtained using diff comparison of the obtained classes, providing us with the additions and deletions for successive versions of the impacted classes. Finally to get the percentage change for both granularities, we required the counts of total classes and LOC for each project, which were retrieved using ProjectCodeMeter,[52] an industrial-scale estimation tool which allows the user to assess various code quality and team productivity metrics. To compare the data collected from different projects, we normalize the impact values for classes and LOC on the basis of its total number of classes and LOC, respectively. We found that on average 9.3% of total classes and 15% of total LOC were affected as a result of library migration. Tables 4.4 and 4.5 summarize the results for the percentage

[52] http://www.projectcodemeter.com/cost_estimation/index.html

of classes impacted and of LOC impacted, respectively. The trend of the results suggests that the Android projects required slightly more effort (in terms of affected classes and LOC) for integration in contrast to the Maven ones.

The discrepancy between mean and median indicates that our data distribution is skewed to the left. In order to visualize our data points, we plotted six histograms for the impacted percentages of classes and LOC: Figures 4.6, 4.7, and 4.8 show the percentage of classes impacted for Maven, Android, and all projects combined, respectively; Figures 4.9, 4.10, and 4.11 show the impacted percentages of LOC for Maven, Android, and all projects combined, respectively.

The results depict that most migrations (69.7% for classes and 61.2% for LOC) required an average impact of 1–10% of the total LOC and classes, thus skewing the results. This however makes sense as in these projects the purpose of using an external library was to fill a small void of functionality; these projects required comparatively minimal effort for migrating from a library. On the other hand, the remaining migrations (30.3% for classes and 38.8% for LOC) required significant effort for mitigation, as the library was at the core of the system, highly coupled with internal modules. There were 28 migrations that had an impact of 20% or more on the classes while there were 43 such migrations for LOC.

The outcome indicates that using an external library at the core of a system is dangerous, as this can have significant impact during the mitigation efforts involved in library migration.

4.4 Summary

We highlighted the procedure for selection of projects, data gathering and answered our Research Questions after analyzing the collected data. Our findings are as follows.

- **RQ0**: We found that many projects undergo library migration (11.9% of the studied projects).
- **RQ1**: We found that 67% of the migrations ended up breaking the dependent system. To avoid library migration in future, in 43% of the migrations their developers decided to avoid introducing a new alternative external library.

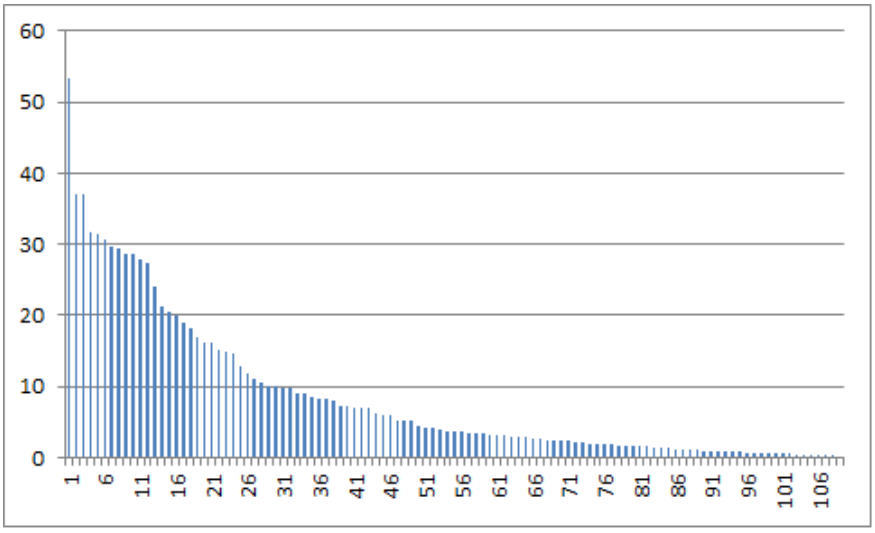

Figure 4.6: Percentage of classes impacted versus Maven project (sorted in descending order).

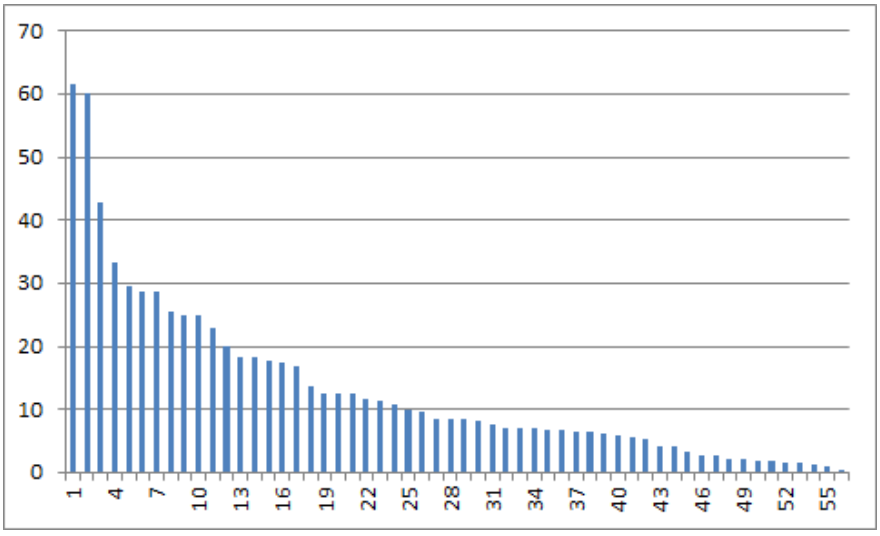

Figure 4.7: Percentage of classes impacted versus Android project (sorted in descending order).

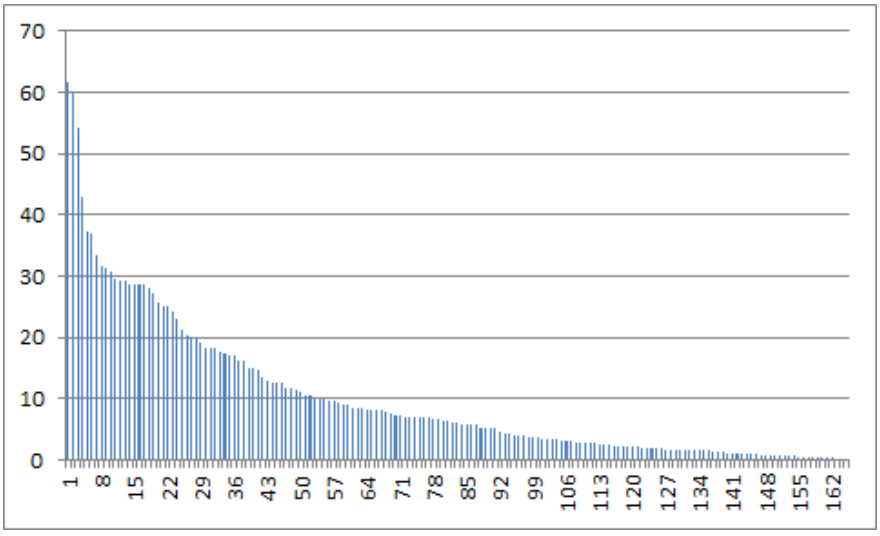

Figure 4.8: Percentage of classes impacted versus project from either platform (sorted in descending order).

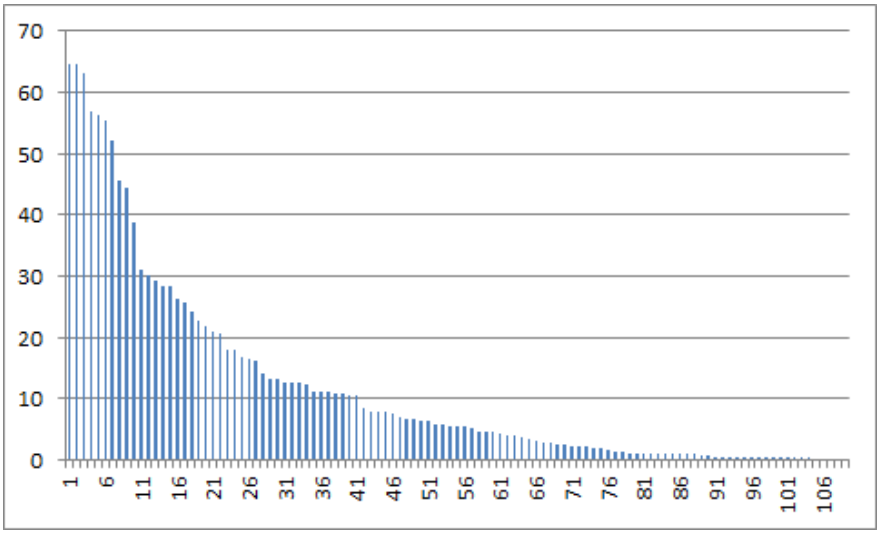

Figure 4.9: Percentage of LOC impacted versus Maven project (sorted in descending order).

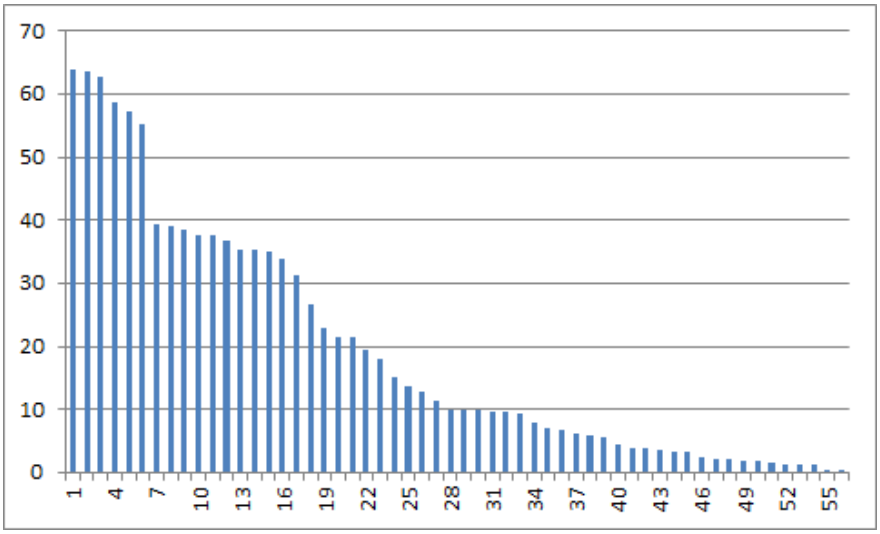

Figure 4.10: Percentage of LOC impacted versus Android project (sorted in descending order).

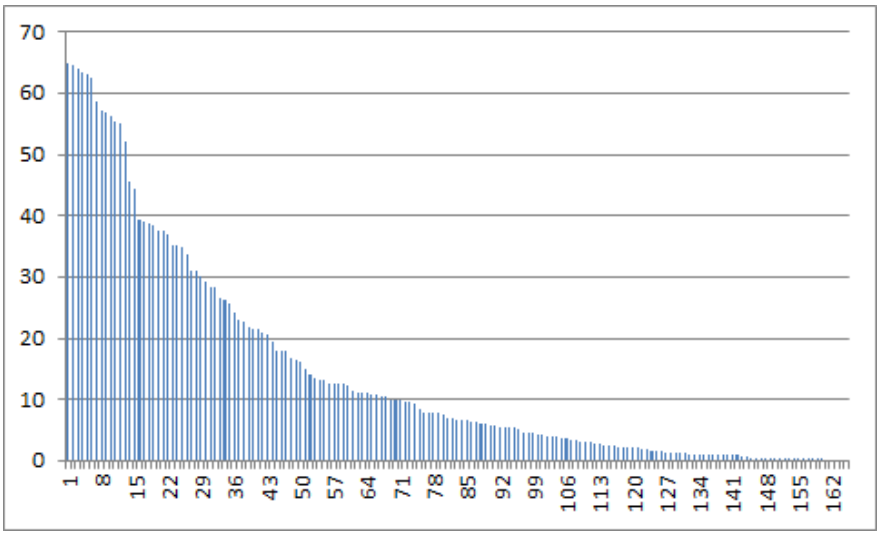

Figure 4.11: Percentage of LOC impacted versus project from either platform (sorted in descending order).

- **RQ2**: Migrations of external libraries are performed for various reasonable motivations (i.e., the existence of a better alternative), rarely attributable to whimsy. However, developers tend to be uncommunicative about these changes: they under-use release notes for notifying client users about such library migrations and they even under-use other communication channels (such as pull requests) that are more oriented towards other project developers.

- **RQ3**: 22% of library migrations impacted the dependent client projects. These migrations resulted in breaking of APIs that caused backwards compatibility problems for the client systems.

- **RQ4**: Mitigation of library migration required significant effort from the project developers. We found that on average developers had to change 9.3% of the classes in their systems while modifying 15% of the total lines of code.

Chapter 5

Tool

In Chapter 2 we presented a motivational scenario in which a developer wastes significant time and energy while migrating between libraries due to following a manual approach. To solve this problem we create a prototype tool called EDW (for External Dependency Watcher), a recommendation system that assists users by forecasting the impacts of library migration. EDW is intended to assist developers during library migration by following a systematic approach, thereby avoiding redundant rework due to information overload.

From our empirical study we found that migrating from a library can end up breaking a dependent system. The compilation errors that arise in such a broken system are localized to classes and their members that directly reference the type and fields of the library. We refer to these classes and members as the *direct impact points*, for simplicity.

EDW satisfies the following requirements. (1) Locate the direct impact points of a library. (2) Assist in locating the transitive impact points for the discovered direct impact set of a library. (3) Assist in estimating the mitigation effort for a library migration.

As demonstrated in Chapter 2, the first step during library migration is to locate these direct impact points. EDW automatically locates the directly impacted classes and members for a library to be migrated. EDW supports the discovery of direct impact points through the dependency viewer (Section 5.2) and editor (Section 5.3) modules. The dependency viewer module allows the user to

view all impacted classes for a selected library. Then the editor module can be used to locate and highlight the directly affected members for an impacted class.

Once the direct impact points have been located, their transitive impacts have to be determined in order to discover the complete impact set of a library. Without determining the transitive impact points, the developer cannot figure out the ripple effects of a library to be migrated. EDW assists in locating the transitive impact points through the impact analysis module (Section 5.4). The module calculates the ripple effects of changing the direct impact points by utilizing the impact analysis provided by the JRipples tool [Buckner et al., 2005].

The process of locating all transitive impact points for a library with several direct impact points can result in information overload. The ripple effect can become huge, making it harder for the developer to recognize the relationship between direct and transitive impact points. Therefore, a graph can assist the developer to direct his search during library migration. EDW generates this graph through the graph builder module (Section 5.5). The generated graph visualizes the dependency links between direct and transitively impacted classes.

Before initiating a library migration, the developer has to decide whether the benefits obtained from migration outweigh the migration cost or not. Therefore, the availability of quantitative measures can provide developers with information about some aspects of the cost of migration. EDW provides metrics to estimate the cost of migration through the metrics builder module (Section 5.6). The calculated metrics can be used to estimate mitigation effort for a library migration.

Section 5.1 describes the design and structure of EDW. Section 5.2 describes the dependency viewer module. Section 5.3 describes the editor module. Section 5.4 describes the impact analysis viewer module. Section 5.5 describes the graph builder module. Section 5.6 describes the metrics builder module. Finally, Section 5.7 describes the workflow of applying EDW to a library migration.

5.1 Overview of EDW

EDW is an Eclipse plugin developed using the Plug-in Development Environment (PDE) of Eclipse.[53] It is developed for the latest Eclipse Oxygen[54] build and works for Java-based Maven projects. EDW starts its analysis with the creation of an *evolving interoperation graph* (EIG) [Rajlich, 2000]. The EIG models a software system as a set of components and their interactions. The graph components are represented as nodes while their interactions are represented as edges. As the software evolves, changes occur in components which can even propagate to other modules of the system.

The EIG supports marking its edges and nodes to record these changes for the user to view and analyze; the edges can be marked as "Impacted", "Propagating", or "Next". An "Impacted" mark indicates that the class would be changed during the IC. A "Propagating" mark indicates that the class itself will not be changed but the neighbouring nodes might be affected. Finally, a "Next" mark indicates that the marked class might change as consequence of changing the neighbouring node. The JRipples analysis ends when there are no longer any classes that are marked as "Next".

EDW comprises five modules, each having its separate graphical view (in the Eclipse IDE) for interacting with the user. The data collected during analysis is kept in a centralized data base which is accessible by all the modules. We provide a distinct Eclipse perspective for EDW that arranges the graphical views for the user, placing the editor in the centre with the impact analysis viewer to its right and the project explorer to its left; Figure 5.1 shows EDW operating on the project cron-utils[55] while using our default view arrangement. The remaining modules are available in views at the bottom of the perspective. The views can be moved in accordance to developer preferences, as per standard Eclipse customizability support.

Figure 5.2 highlights the relationships between these modules (further described in the subsequent sections). The boxes represent the five modules, one of which (the dependency viewer module) contains three submodules. The dashed arrows represent dataflow between (sub)modules. The developer is expected to start with the external JAR submodule, and this control flow is shown

[53] https://www.eclipse.org/pde/
[54] https://www.eclipse.org/oxygen/
[55] https://github.com/jmrozanec/cron-utils

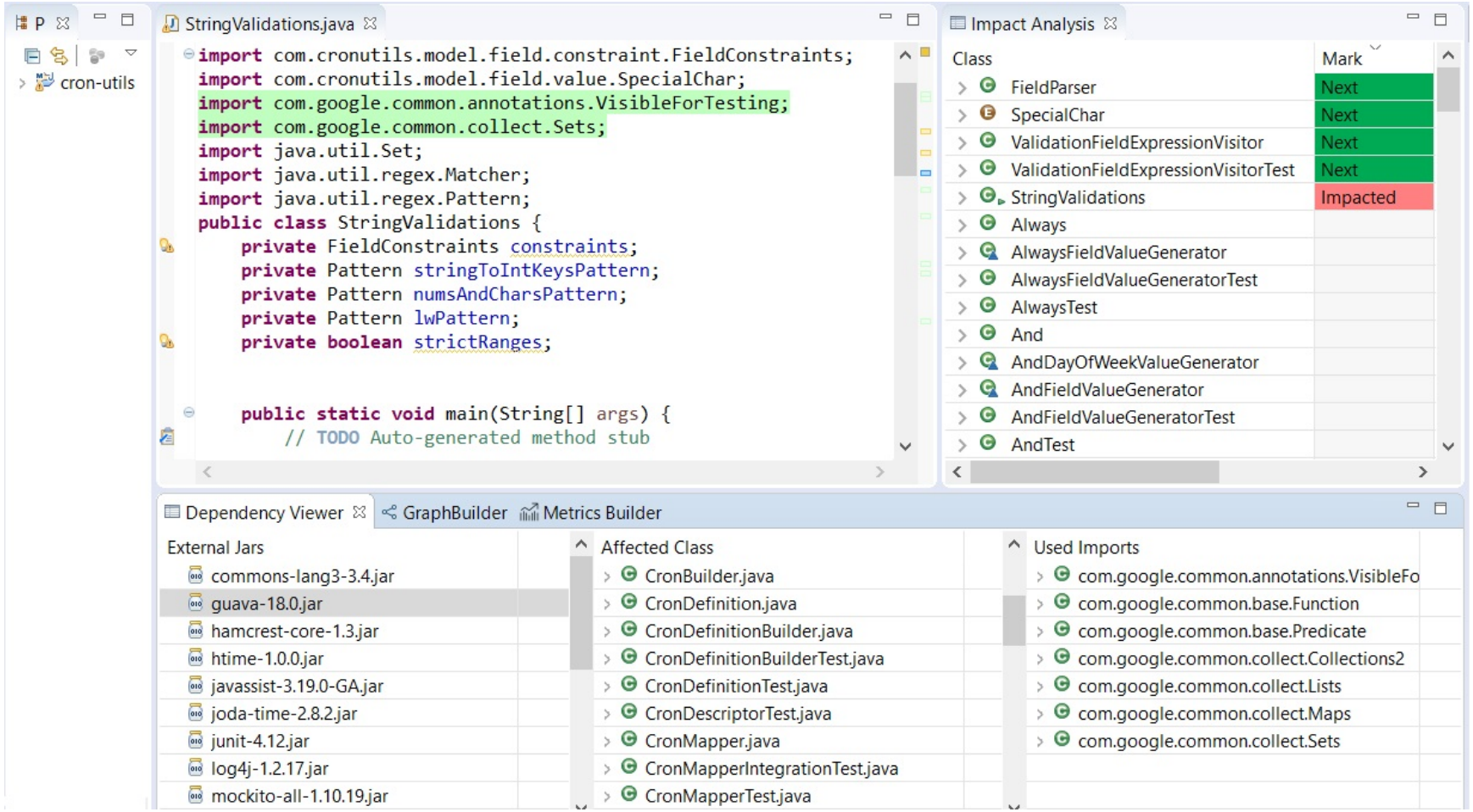

Figure 5.1: EDW analysis on the project cron-utils while using our standard view arrangement.

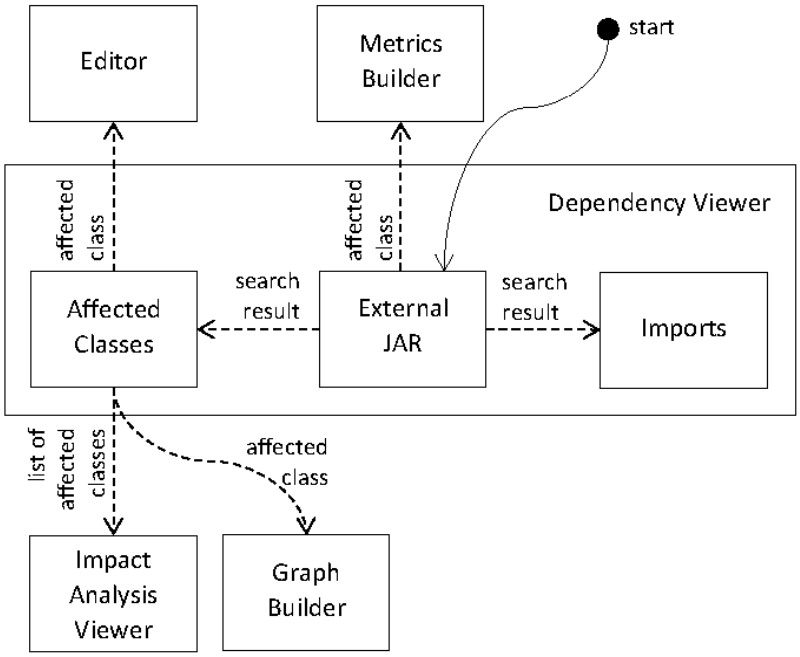

Figure 5.2: EDW structure and interaction between modules.

as a solid arrow; other control flow is flexible at the developer's discretion (i.e., the developer can switch to different views arbitrarily).

5.2 Dependency Viewer Module

The dependency viewer module is the core of EDW as it acts as an interface that allows interaction with other provided modules; it is divided into three sub-modules: (1) the external JAR submodule; (2) the affected classes submodule; and (3) the imports submodule. These submodules collectively perform discovery of direct impact points for EDW. The external JAR submodule displays all used libraries for the selected project, while the affected classes and import submodules list the impacted classes and used imports for a selected library, respectively. Figure 5.3 shows the dependency viewer after analysis on the project cron-utils. The user has selected Guava in the external JAR submodule. This selection resulted in populating the affected classes and imports submodules with impacted classes and their used imports from Guava, respectively. The affected class CronBuilder and

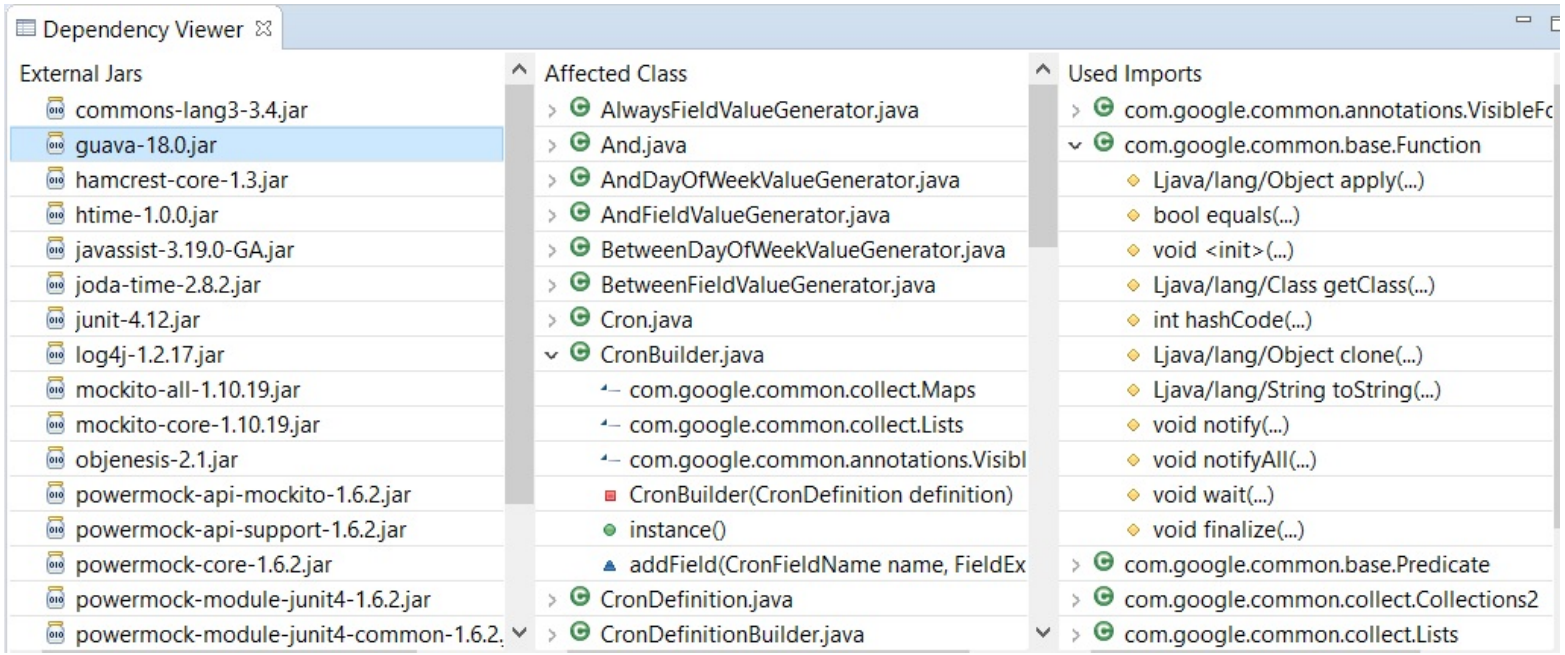

Figure 5.3: Dependency viewer after analysis on the project cron-utils.

imported class Function have also been expanded to view the affected members and API signature, respectively.

5.2.1 External JAR submodule

Maven maintains the required build information for a project including all dependency links in the POM.xml file. The external JAR submodule therefore parses the POM.xml file in order to locate all external dependencies being used in a project. Creating an EIG from scratch every time for a system of significant size can waste time and resources. By locating all the external dependencies at the start, EDW can perform its analysis for different libraries without creating an EIG every time from scratch.

In order to determine impact points, the external JAR (EJ) submodule extracts the least common parent package name for a library that users wants to migrate. This package name is referenced as an import declaration in any class that is dependent on the library. The heuristic of using the least common package name for search has two advantages. First, searching for this name is faster as the number of packages in contrast to the total number of classes is significantly lower than the alternatives. Second, using a single package name results in a unique and refined search result in contrast to using the set of packages that directly contains a class, as in the latter case, the same class can be returned more then once. The EJ submodule provides the extracted package as a query to the Eclipse search engine which in turn locates the affected classes for a proposed library migration. The Eclipse search engine requires creation of the search pattern, search scope, and result collector: EDW creates a search pattern that is limited to searching the import references while using an exact match strategy, whereas the scope of the project is restricted to the workspace of the project being analyzed. The search using the Eclipse search engine is more efficient as the IDE already holds a Java model representation of the project. This model is a lightweight and fault tolerant representation of the Java project; it does not contain as much information as the AST, but is fast to create. Searching via any other means would have required to create an AST or model from scratch which would have been expensive for our analysis. The search returns the initial

impact set of the migration, which is then provided as an input to the affected classes and imports submodules.

5.2.2 Affected classes submodule

The affected classes submodule analyzes the search result passed from the external JAR submodule, in order to locate the list of affected classes. After retrieving the list of affected classes, it traverses the EIG graph of the system to mark its edges. The affected classes' edges are marked as "Impacted" during traversal, which are then used to locate the transitive impact set by the impact analysis module. It also communicates with the editor module, in order to provide the list of affected classes as input, which are then used by it for further analysis.

5.2.3 Imports submodule

The imports submodule analyzes the search result passed from the external JAR submodule, in order to locate the used classes of the selected library. It then extracts the API signature of the located used classes from the library being migrated. These API signatures involve publicly available methods being offered by the import classes of library. The API signature can assist while understanding the usage of library and coupling in the system.

5.3 Editor Module

The editor module locates the directly affected members (methods or fields) of the discovered impact set (classes) in order to mark them as "Impacted" in the EIG. EDW analyzes the classes by creating abstract syntax trees (ASTs) for the impact set. An AST can represent Java source code as a tree, which is more convenient to analyze and modify programmatically. In a simple AST every element of the source code is mapped to a node or a subtree. However, such an AST misses out on certain information such as the types of variables, or parent and child classes. This information

is contained in the bindings (from uses of names to their declarations). EDW utilizes bindings to support polymorphism; therefore, it can also locate the impacted overridden methods in a class.

In order to save resources EDW creates ASTs by following the conservative strategy of lazy instantiation: an AST is only created for an affected class when the user selects it for analysis the very first time. Without following this strategy EDW would have had to have created ASTs of all affected classes at the same time, likely resulting in running out of memory in addition to causing a slow startup period.

The ASTVisitor class[56] is used for traversing and searching the nodes of the instantiated AST. ASTVisitor conforms to the Visitor design pattern[57] that allows efficient search of nodes in an AST. The EDW visitor locates the AST nodes that depend on the referenced classes of the library to be migrated. The immediate parent (method or field) of the found node is stored in the database along with its statement line number. In order to highlight the impacted code segments for modification, EDW attaches Eclipse markers to them. The editor module allows the user to modify these impacted code segments while mitigating a library migration.

Figure 5.4 shows an instance of the editor after selection of the affected class StringValidations. The direct impacted code segments after AST traversal are highlighted using markers.

5.4 Impact Analysis Viewer Module

EDW utilizes the hierarchical view of JRipples to perform CIA on the initial impact set. As the JRipples code base is incompatible with the latest Eclipse build, we extracted the CIA module and updated it to make it compatible.

As already discussed, the directly affected classes and their members are marked as "Impacted" in the EIG by the preceding modules. The impact analysis viewer module traverses the EIG edges in order to determine the transitively affected members of the migration. The transitive members

[56] https://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.cdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fcdt%2Fcore%2Fdom%2Fast%2FASTVisitor.html [accessed 2018/10/11]

[57] https://sourcemaking.com/design_patterns/visitor [accessed 2018/10/11]

StringValidations.java

```java
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import com.google.common.annotations.VisibleForTesting;
import com.google.common.collect.Sets;
public class StringValidations {
    private FieldConstraints constraints;
    private Pattern stringToIntKeysPattern;
    private Pattern numsAndCharsPattern;
    private Pattern lwPattern;
    private boolean strictRanges;

    @VisibleForTesting
    Pattern buildStringToIntPattern(Set<String> strings){
        return buildWordsPattern(strings);
    }

    @VisibleForTesting
    public String removeValidChars(String exp){
        exp = exp.toUpperCase();
        Matcher numsAndCharsMatcher = numsAndCharsPattern.matcher(exp);
        Matcher stringToIntKeysMatcher = stringToIntKeysPattern.matcher(numsAndCharsMatcher.replaceAll(""));
        Matcher specialWordsMatcher = lwPattern.matcher(stringToIntKeysMatcher.replaceAll(""));
        return specialWordsMatcher.replaceAll("").replaceAll("\\s+", "").replaceAll(",", "").replaceAll("-", "");
    }


    @VisibleForTesting
    Pattern buildLWPattern(Set<SpecialChar> specialChars){
        Set<String> scs = Sets.newHashSet();
```

Figure 5.4: Editor after selection of the affected class StringValidations.

are automatically marked as “Next” which indicate that they should be manually inspected by the user. The user can change the “Next” marked members to either “Propagating”, “Impacted”, or “Unchanged” based on his observation during library mitigation. Further details of this module are available online.[58]

Figure 5.5 shows the impact analysis view after selecting the affected class StringValidations. The directly affected class members are marked as “Impacted” while any probably affected one is marked as “Next”.

[58] http://jripples.sourceforge.net/jripples/manual/reference/views/hierarchical_view.html [accessed 2018/10/11]

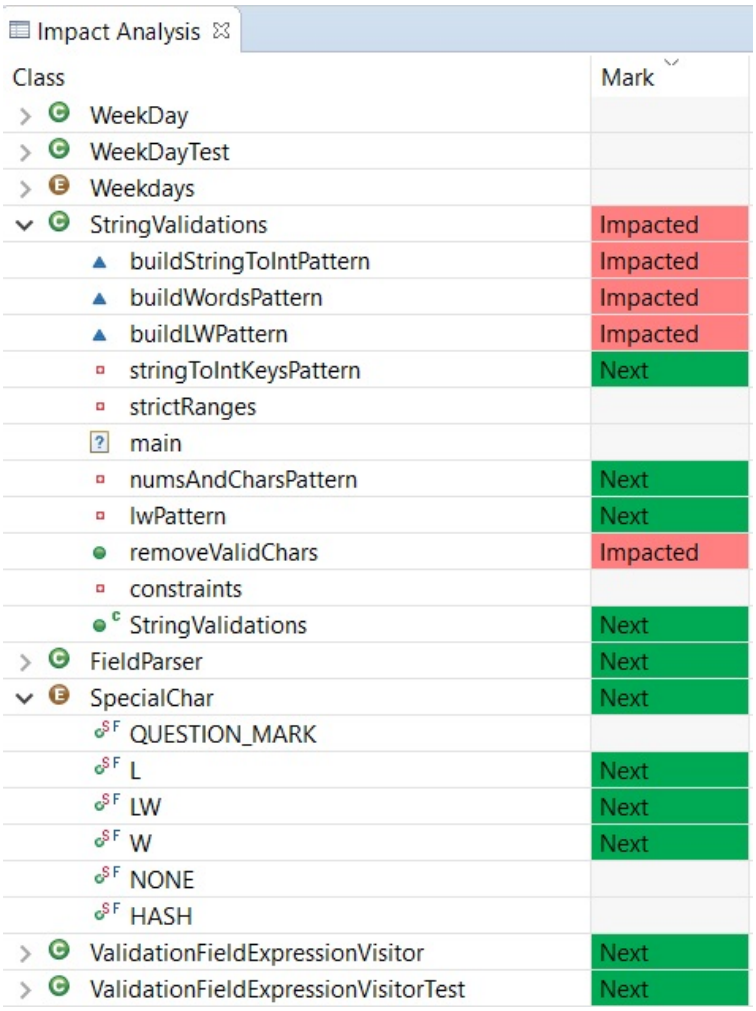

Figure 5.5: Impact analysis view after selecting the affected class StringValidations from the dependency viewer.

5.5 Graph Builder Module

EDW provides a graph builder module that highlights the dependency links between affected classes. EDW supports the calculation of complete CIA which calculates probable impacts for all the affected classes of the initial impact set at the same time. Therefore, performing a complete CIA on a project of significant size can result in information overload. Figure 5.6 shows the impact analysis view after performing complete CIA on Guava for the project cron-utils. It can be seen that there are several classes marked as "Impacted" along with their probable impact segments marked as "Next". This much information can easily become overwhelming as it is very hard to determine which "Next" class depends on which "Impacted" class. On the other hand, by using the dependency graph, the user can direct their search during library migration.

EDW creates a graph using the Eclipse Zest framework,[59] an Eclipse visualization toolkit, which has a set of graph components built for Java. The module creates a directed graph (specifically, a tree) relative to an affected class. Each graph node represents a class, subdivided into three compartments, with its top compartment showing affected fields and its bottom one showing affected methods. The graph itself is organized into three parts: the top-level nodes depict the classes of the system that depend on an affected class of library migration; the bottom-level nodes show the classes referred to by the library upon which the affected class depends; and the middle node shows the affected class itself. The tree structure is designed to make the dependency link between artifacts more evident. For example, Figure 5.7 shows the dependency graph for the project cron-utils class with affected class StringValidations. (Note: any interdependencies between the nodes, other than with the affected class, are not shown.)

5.6 Metrics Builder Module

The metrics builder module allows the user to estimate the minimum effort required for migrating a selected library. It provides quantitative measures for the supported metrics that can provide the

[59] https://www.eclipse.org/gef/zest/ [accessed 2018/10/11]

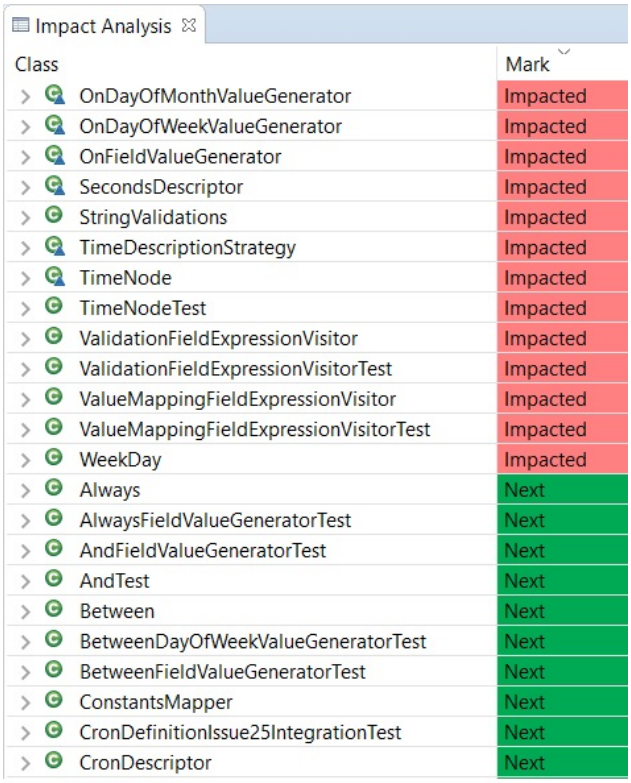

Figure 5.6: Impact analysis view after performing complete CIA on Guava for cron-utils.

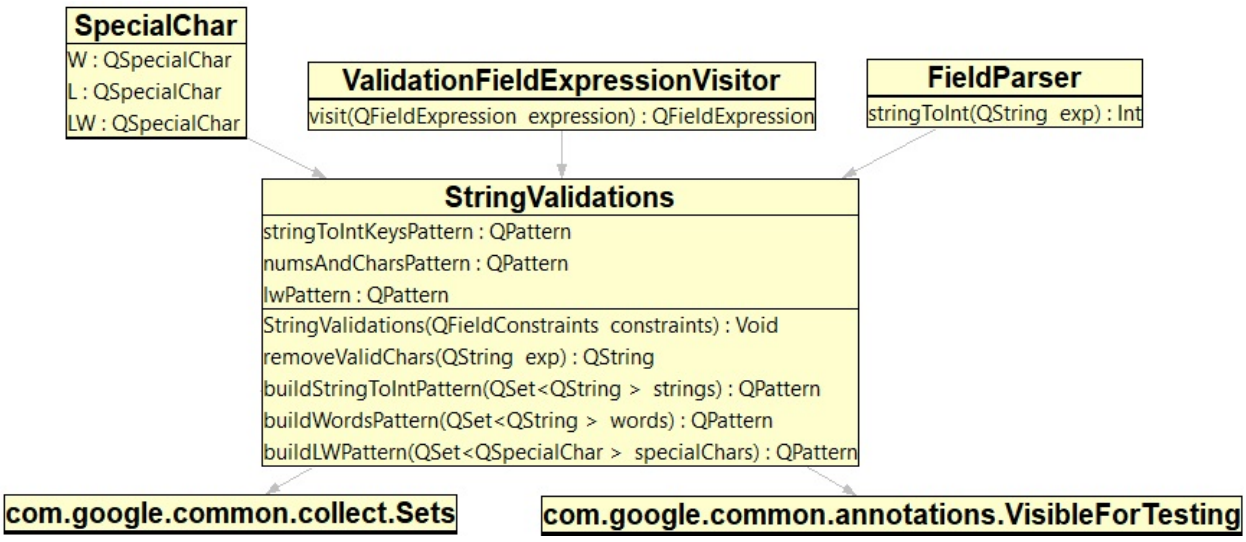

Figure 5.7: Dependency graph for the class StringValidations in the project cron-utils.

developer with information about some aspects of the cost of migration. The developer can then decide whether the benefits obtained from migration outweigh the migration cost or not.

Currently, EDW supports four different metrics related to library migration, all measuring the average change in one kind of item: classes, methods, fields, or LOC. For example, Figure 5.8 shows the four metrics calculated for Guava on the cron-utils project. These metrics are calculated using the affected and total number for each metric respectively. As the supported metrics are preliminary, we have provided an Eclipse extension point for future development.

Metrics Builder

Metric	Affected	Total	% Change
Classes	43	117	36.75
Methods	108	883	12.23
Fields	7	322	2.17
LOC	1006	10662	9.44

Figure 5.8: Four metrics calculated for the Guava library in the project cron-utils.

5.7 Workflow

To illustrate the workflow of applying EDW to library migration, we use the motivational example (from Section 2.1) of the cron-utils project in which the developer wants to migrate from Guava. Before performing the migration, the developer starts with the metrics builder module (see Figure 5.8), which estimates the impacts of this library migration. The metrics show the estimated percentage change required to migrate cron-utils away from Guava; from this information, the developer can decide whether the benefits obtained from migration outweigh the migration cost or not. The metrics show that migrating cron-utils away from Guava would impact 36% of the total classes while affecting 9% of the total code base.

During migration, the developer needs to locate the affected classes that are dependent on Guava. They can achieve this goal through EDW by selecting Guava in the external JAR submodule, which would then locate the affected classes of the library, passing them as input to the affected

classes submodule; the external JAR submodule would also pass the used imports of Guava as input to the import submodule. Figure 5.3 shows the dependency viewer module with its populated subviews after performing analysis on cron-utils: the developer has selected the Guava dependency in the external JAR subview, which results in populating the other two subviews respectively.

Once EDW has located the directly impacted classes for the developer, the next step is to determine the affected members of each class. The developer can perform this by selecting a class from the affected classes submodule, which would open it in the editor module, locating and highlighting the affected members with directly impacted LOC for the selected class. Figure 5.4 shows an instance of the editor after selection of the affected class StringValidations; the direct impact points are highlighted in green for the user to view.

After EDW has located all affected classes along with their members, the developer must consider the transitive impact points of change. They can do this by performing a CIA on the located classes through the impact analysis view. Figure 5.5 shows the impact analysis view after selecting the affected class StringValidations: it shows the already located affected classes and members as "Impacted", while recommending the potentially transitively impacted points as "Next". The developer must explore the recommended points in order to consider whether these will truly be impacted after migration. While performing CIA on a project of significant size, the recommended points can result in information overload (e.g., see Figure 5.6). To avoid this, the developer can create a dependency graph via the graph builder view, highlighting the relationship between an impacted class and its transitively affected classes and members; Figure 5.7 shows the dependency graph for the affected class StringValidations. Once the developer has understood the consequences and impacts of library migration through EDW, they can finally replace Guava with an alternative.

5.8 Summary

In this chapter, we introduced our prototype tool EDW that can assist developers while performing a library migration. The plugin is built as a recommendation system and requires developer to work along with it while performing the change. Our plug-in works on Maven Java projects in the context of software evolution where a developer wants to perform a library migration.

EDW is based on a modular approach as it distributes the functionality among five modules, each with a corresponding graphical view. A developer can perform: discovery of direct impact points, impact analysis, graph generation, and metrics generation, using the respective modules. In order to support future development, we have provided Eclipse extension points for our modules.

Chapter 6

Simulation Study

This chapter discusses our evaluation of EDW through simulation. Our research questions are as follows.

RQ5. How well is EDW able to locate the initial impact set for library migration?

RQ6. What factors impact how well EDW performs?

As a point of comparison, we also evaluate JRipples on the same projects, in order to determine how well EDW does in contrast for locating the initial impact set. We followed a similar procedure and setup for collecting initial impact set for both tools, while evaluating them on three different granularities. Section 6.1 explains our criteria for the selection of evaluated systems. Section 6.2 discusses the procedure followed for evaluation. Section 6.3 presents the results of our evaluation.

6.1 Selection of Projects

We evaluate EDW on projects obtained from GitHub. In order to reduce bias, the selection of systems was based on a set of criteria as follows.

- Each selected project must not be already studied in our retrospective study from Chapter 4.
- Each selected project must have a library migration between any two of its successive released versions.

Table 6.1: Systems for evaluation.

ID	Name	Library replaced
1	graylog2-server[60]	JSON.simple[61]
2	Esper[62]	Apache Commons Logging[63]
3	Choco solver[64]	SLF4J API Module[65]
4	WebProtégé[66]	Google Guice Core Library[67]
5	GeoWave[68]	Apache Log4j[69]
6	HAPI-FHIR[70]	JSR 374 (JSON Processing) API[71]
7	Commands[72]	Guava: Google Core Libraries For Java[73]
8	azure-cosmosdb-java[74]	JSON in Java[75]
9	Apache Dubbo[76]	EasyMock[77]
10	ta4j-origins[78]	Joda-Time[79]

- Each selected project must possess successive released versions that were both compilable under the latest Eclipse build. (We wanted to make sure that the systems were compatible with our tool.)
- Each selected project must not be deprecated.
- Each selected project must have a unique library being migrated (amongst the set of selected projects) to avoid redundancy.

[60] https://github.com/Graylog2/graylog2-server [accessed 2018/10/11]
[61] https://code.google.com/archive/p/json-simple/ [accessed 2018/10/11]
[62] http://www.espertech.com/esper/ [accessed 2018/10/11]
[63] https://commons.apache.org/proper/commons-logging/ [accessed 2018/10/11]
[64] http://www.choco-solver.org/ [accessed 2018/10/11]
[65] https://mvnrepository.com/artifact/org.slf4j/slf4j-api [accessed 2018/10/11]
[66] https://webprotege.stanford.edu/ [accessed 2018/10/11]
[67] https://mvnrepository.com/artifact/com.google.inject/guice [accessed 2018/10/11]
[68] https://github.com/locationtech/geowave [accessed 2018/10/11]
[69] https://logging.apache.org/log4j/ [accessed 2018/10/11]
[70] http://hapifhir.io/ [accessed 2018/10/11]
[71] https://static.javadoc.io/javax.json/javax.json-api/1.1/overview-summary.html [accessed 2018/10/11]
[72] https://github.com/aikar/commands [accessed 2018/10/11]
[73] https://opensource.google.com/projects/guava [accessed 2018/10/11]
[74] https://github.com/Azure/azure-cosmosdb-java [accessed 2018/10/11]
[75] https://github.com/stleary/JSON-java [accessed 2018/10/11]
[76] https://github.com/apache/incubator-dubbo [accessed 2018/10/11]
[77] http://easymock.org/ [accessed 2018/10/11]
[78] https://github.com/mdeverdelhan/ta4j-origins [accessed 2018/10/11]
[79] https://www.joda.org/joda-time/ [accessed 2018/10/11]

We used GitHub search for locating the projects of our study. We searched for Maven projects that with at least 100 stars; the returned projects were sorted based on popularity of the repository. The first 10 projects that fulfilled our criteria were selected for our study. Table 6.1 lists these systems while Appendix D specifies their details.

6.2 Procedure

We evaluated EDW and JRipples on three different granularities, as follows.

- Class-granularity: Project classes which directly depended on the library under consideration.
- Method-granularity: Any method in the directly affected classes that was changed due to migrating the library under consideration.
- Field-granularity: Any field in the directly affected classes that was changed due to migrating the library under consideration.

6.2.1 Ground truth

To evaluate the selected projects, we first collected the "ground truth" (i.e., the actual impact set) from the actual replacement from developers' commits. The library migrations in each project were accompanied by a specific online commit on GitHub, e.g., a specific commit for the Joda-Time replacement in the project cron-utils.[80] Ideally, such a commit only contains changes to the project resulting from a migration, and we assume that this ideal is met (i.e., we ignore the possibility of needed changes missed by the committer or performed changes that are not related to the migration that were extraneously included in the commit). We recorded the full path of each artifact (affected classes, methods, and fields) in the impact set. For example, the following instance of artifacts were collected for each respective granularity during the evaluation of the Choco solver project.

[79] https://github.com/search [accessed 2018/11/10]

[80] https://github.com/jmrozanec/cron-utils/pull/107/commits/289577660c55a80baada1ca397c10af980464f9f [accessed 2018/11/10]

- Class-granularity: org.chocosolver.util.objects.graphs.MultivaluedDecisionDiagram
- Field-granularity: org.chocosolver.solver.constraints.Propagator.LOGGER
- Method-granularity: org.chocosolver.memory.EnvironmentException.EnvironmentException

6.2.2 Estimated impact set for EDW

We collected the estimated impact set for EDW by running it on each selected project in turn. Within EDW, we selected the library away from which the project was known to have been migrated in the past. EDW located and marked the impacted classes, methods, and fields of the library to be migrated. The impacted classes were retrieved from the affected classes submodule (Section 5.2.2) in the dependency viewer (Section 5.2), while the affected members were located through the editor module (Section 5.3). The reported results were recorded for each granularity and an estimated impact set with the full path of the artifacts was created. The full path for method- and field-granularity contains the path up to the parent class as a prefix, as shown in the example results for Choco solver in the preceding section.

6.2.3 Estimated impact set for JRipples

JRipples requires manual search of initial impact points; therefore, we assumed to know each system's key class with the import declarations of the library to be migrated. We collected the estimated impact set for JRipples on the projects through the following procedure.

1. With JRipples, we first located the initially impacted classes through manual, Eclipse-based search. Since a developer is free to follow whatever procedure they deem appropriate, we followed three independent strategies to generate queries, to be used for the search. The strategies vary in their selection of different combination of library import prefixes, and we deem each to be a likely and feasible option. Each strategy was used to locate all affected classes, working from scratch each time.

- Strategy 1: Iteratively use all unique import declarations of the library to locate their affected classes. For the Guava example in the motivational scenario (Section 2), this strategy would entail using each of the 10 imports being used in the system (starting from com.google.common.base.joiner to com.google.common.primitives.Int).
- Strategy 2: Iteratively use the set of common prefix names among the import declarations. For the Guava example in the motivational scenario, this strategy would be followed using the set of common prefix names com.google.common.base, com.google.common.collect, and com.google.common.primitives.
- Strategy 3: Use the prefix common amongst the used import declarations. For the Guava example in the motivational scenario, this strategy would be performed using com.google.common.

2. We then reviewed the returned search results for each strategy to verify its impact. The verified affected classes were then manually marked in JRipples as "Impacted". There were overlaps in searches where the reviewed class was already marked from a previous search result. We recorded such classes as an "overlap" for the search strategy.

3. Once all affected classes were marked in JRipples, we iteratively searched for the affected methods and fields inside each affected class based on its used import declarations. The located members were then manually marked as "Impacted" in JRipples.

4. Similarly to EDW, we found the marked results at the end for each granularity and recorded an estimated impact set with the full path of each artifact.

6.2.4 Comparing estimated impact set and ground truth

We ignored all "Next" marked artifacts, as these require human intervention for confirming any impact of migration on them. To reduce the possibility of human error, we wrote a simple program that compared the estimated and actual impact sets. If a tool identified an artifact as impacted and

it was part of the ground truth set, the comparator labelled it as a true positive (TP). Otherwise if an artifact was identified by a tool as affected but it was not part of the ground truth set, the comparator labelled it as a false positive (FP). Finally if an affected artifact was present in the ground truth but it was not identified by a tool, it was labelled as a false negative (FN).

We obtained the average precision and recall for each granularity with both tools, by calculating the unweighted arithmetic mean of the precisions and recalls for individual projects.

6.3 Results

6.3.1 RQ5: How well is EDW able to locate the initial impact set for library migration?

Table 6.2 lists the true positives, false positives, and false negatives at class-granularity for both tools, across the ten projects; precision (P), recall (R), and F_1 score (F_1) are thus calculated as averages across the ten projects. "MSI" (manual search interpretation) represents the total number of search hits that the user has to manually interpret in order to verify their impacts, across all ten projects. Table 6.3 lists the equivalent results for method- and field-granularity for both tools, where "PTR" (points to remember) represents the total number of direct impact points that the user must keep in mind, in the absence of tool support. The user needs to use some means (such as a piece of paper or whiteboard) for storing and retrieving the impact points; we refer to this process of storing and retrieving information as "memorization" and "remembering", for simplicity. However, Appendix E summarizes the results for individual projects for each granularity. JRipples under Strategies 2 and 3 cannot be applied to locate the affected members due to their abstract nature. The strategies do not specify the particular class of the third party library being used in the system, hence Eclipse is unable to locate any points of impact. EDW has a precision of 1 for all granularities. For recall, it achieved over 0.99 for class- and field-granularity, and ~0.86 for method-granularity. EDW produces no overlap, no need to manually interpret search results, and no points that must be remembered.

Table 6.2: Approach mean accuracy for class-granularity.

Approach	TP	FP	FN	P	R	F_1	Overlap	MSI
EDW	102.2	0	0.4	1	0.99	0.99	0	0
JRipples: Strategy 1	102.2	2.6	0.4	0.89	0.99	0.93	50	232
JRipples: Strategy 2	102.6	7.7	0	0.86	1	0.92	15.34	391
JRipples: Strategy 3	102.6	39.3	0	0.86	1	0.92	0	502.6

Table 6.3: Approach mean accuracy for method- and field-granularity.

Approach	Granularity	TP	FP	FN	P	R	F_1	PTR
EDW	method	32.4	37.1	0	1.00	0.86	0.92	0
JRipples: Strategy 1	method	32.4	9.8	4.7	0.85	0.86	0.82	155
JRipples: Strategy 2	method	0	0	0	0	0	0	N/A
JRipples: Strategy 3	method	0	0	0	0	0	0	N/A
EDW	field	149.5	0	0.67	1	0.99	0.99	0
JRipples: Strategy 1	field	149.67	6.5	1.67	0.93	0.99	0.95	512
JRipples: Strategy 2	field	0	0	0	0	0	0	N/A
JRipples: Strategy 3	field	0	0	0	0	0	0	N/A

For comparison, JRipples places a much greater burden on the developer. While its recall is essentially the same as that of EDW (which should not be surprising since EDW builds atop it), it produces significant false positives (as evidenced by reduced precision), overlap for two of the three search strategies, the need for frequent manual interpretation of search results, and many points to remember independently from the tool. We expect that each of these burdens will be error-prone and fatiguing on the developer.

- **Class-granularity.**

  EDW and JRipples under Strategy 1 achieved promising recall for 9 of the evaluated systems. However, the recall drops to 99% for the project Esper as the tools did not find four of the affected classes. The reason for this drop is related to the usage of an on-demand import declaration, i.e., import org.apache.commons.logging.*. This issue is discussed in detail in the next section. On the other hand, JRipples under Strategies 2 and 3 was able to locate all affected classes and hence achieved perfect recall.

  EDW achieved a precision of 1 for the evaluated systems. For the search strategies of JRipples,

precision declines as we move from more specific strategies to being more general. EDW was able to achieve better precision, without any manual search interpretation needed. JRipples strategies on the other hand, required the developer to review an average of 232, 391, and 502.6 MSI, respectively.

Each FP, MSI, and overlap would require human attention; thus, they represent an opportunity that can confuse the developer during migration. The JRipples strategies require manual interpretation and attention of the developer to verify the results, which require extra decisions and interpretation. Search strategy 1 has slightly better precision in contrast to the other two approaches; however, it requires the developer to review 50% of the retrieved classes redundantly due to overlap. Since Eclipse search is not directly linked with JRipples, the developer is required to first review the search results and verify whether they are already marked or not. The review of the same search results manually and redundantly is an opportunity for confusion.

- **Member-granularity.**

  Method-granularity recall. EDW and JRipples under Strategy 1 achieved promising recall for seven of the evaluated systems. However, their recall dropped for Choco solver (0.43), GeoWave (0.35), and azure-cosmosdb-java (0.87). Therefore the average recall comes out to be 0.86. For the project Choco solver, four directly impacted methods were not recalled by EDW, due to their nested nature, as explained in the next section. JRipples under Strategies 2 and 3 cannot be applied to locate affected members in some cases, due to their abstract nature. Table 6.4 summarizes the recall with their resulting FNs and "Next" (transitively affected) marked methods. Both tools were unable to directly mark transitively affected methods as "Impacted", which affected their recall. We discuss this issue in the next section.

  Field-granularity recall. EDW and JRipples achieved promising recall for five of the evaluated systems. The project graylog2-server did not have any affected fields while the projects WebProtégé', HAPI-FHIR, and azure-cosmosdb-java had only one affected field each; therefore,

Table 6.4: Recall of projects at method-granularity.

Project	TP	FN	Next	Recall
1	12	0	0	1
2	8	0	0	1
3	24	31	27	0.43
4	23	0	0	1
5	6	11	11	0.35
6	37	0	0	1
7	30	0	0	1
8	62	9	9	0.87
9	69	0	0	1
10	53	0	0	1

they were not considered while calculating average precision and recall. For the affected fields, recall drops to 99% only for the project Esper as the tools did not find four of the affected classes that contained the fields. JRipples under Strategy 1 did not find these four classes; under the other two strategies, it did.

Method- and field-granularity precision. EDW achieved a precision of 1 for locating the impacted members in evaluated projects. The precision of JRipples is comparatively worse than EDW for locating affected members. The decline in precision is due to the limitations of Eclipse search of members inside a class. JRipples also requires developer to remember an average 155 and 512 points for methods and fields, respectively, during migration. JRipples cannot directly mark the affected LOC and hence the developer is required to remember these manually. These points are spread out in different classes, which can make it difficult and impractical for developer to manually memorize them. Therefore, each direct impact point represents an opportunity that require human attention and can create confusion.

Discussion

The precision and recall of JRipples can be attributed to our assumption that the developer is clearly aware of the used imports. However, in a scenario where this assumption is not true,

the developer would have to first manually figure out the used imports. As demonstrated by our motivational scenario, the discovery of imports require extra steps and decisions that can result in further confusion and mistakes. EDW on the other hand does not assume anything of this sort; it does not require the developer to be aware of the imports of the library to be migrated.

Locating the initial impact set through JRipples results in higher FP, searches, MSI, and overlap. Each of these require human interpretation for decision and verification, which can result in more mistakes and confusion. Moreover, for method- and field-granularity, the developer has to remember a significant number of PTR manually. Therefore, the approach with JRipples can become impractical to use for a project that is significantly impacted by library migration. For example, in Esper, the three strategies resulted in 1,405, 2,949, and 2,976 MSI respectively, while there were a total of 2831 PTR for affected members. Manually going through these with JRipples requires more decisions and hence can create greater difficulty and confusion.

On the other hand, EDW automates the steps for locating the initial impact set, with comparatively better precision. The developer is not required to manually and redundantly go through any search result and hence makes fewer decisions at this stage. It also marks the affected class or members, so the developer does not have to remember the direct impact points.

6.3.2 RQ6: What factors impact how well EDW performs?

We found that all false negatives discovered during our evaluation were due to three limitations of our tool, as follows.

- **On-demand import declarations.**

  Our tool ignores any on-demand import declarations. Such declarations import all the classes in a package at once without requirement of being specific about a class. We decided to ignore these import declarations since our empirical investigations suggested that most stabilized system prefer to fully specify their imported classes as it reduces extra dependencies. Moreover, EDW is a recommendation system that needs the developer to work along with it

during migration; he should be aware of the library classes being used which is a requirement for performing a successful transformation.

These issues can be worked around by replacing such import declarations through the programmatic support provided by the Eclipse IDE (i.e., "organize imports") even on a temporary, internal basis. This can also be integrated directly into the tool by adding an extension point to the current search strategy.

- **Granularity.**

  As already discussed, our tool works at the granularity of class members (methods and fields). However, Java supports declaring nested classes as well. Currently our impact analysis module is unable to mark such nested members. For example in Figure 6.1 the field Logger from library is defined inside a class that also defines a nested type (an enumeration type called Trace). The methods of Trace use this field; however, they are declared at a granularity which is one level down of our tool's reach. These methods are not marked by EDW as "Impacted" in the impact analysis view. The user would have to mark these members

```
final Logger LOGGER = LoggerFactory.getLogger(IPropagationEngine.class);

public enum Trace {
    ;

    public static void printPropagation(Variable v, Propagator p) {
        LOGGER.debug("[P] {}", "(" + v + "::" + p + ")");
    }

    public static void printModification(Variable v, IEventType e, ICause c) {
        LOGGER.debug("\t[M] {} {} ({})", v, e, c);
    }


    public static void printSchedule(Propagator p) {
        LOGGER.debug("\t\t[S] {}", p);
    }

    public static void printAlreadySchedule(Propagator p) {
        LOGGER.debug("\t\t[s] {}", p);
    }
}
```

Figure 6.1: The field Logger used in a nested member.

manually during the analysis. This limitation can impact the recall of methods and members defined inside a nested element.

- **Transitively affected methods.**

  EDW and JRipples do not directly mark transitively affected methods as "Impacted", which resulted in increasing the FN's and hence reducing recall of the tools for this granularity. These transitively affected methods are first marked as "Next" by the impact analysis module. For example, Figure 6.2 shows five marked artifacts(method, field, or class) in the class CronParser; the method parse() is marked "Impacted" by the tool while cronDefination() is labelled as "Next". An artifact marked "Next" is considered to have a transitive dependency on a method marked "Impacted". For example, Figure 6.3 shows a method AlwaysFieldValueGenerator in project cron-utils, that directly depends on the class List from the external library Guava. Figure 6.4 shows another method from the class AlwaysFieldValueGeneratorTest that depends on the method from Figure 6.3. The tools considers this as a transitive dependency between the method of AlwaysFieldValueGeneratorTest and the external library. Human intervention is required to decide whether the method of AlwaysFieldValueGeneratorTest actually depends on the library or not. In reality, the method in Figure 6.4 *does* transitively depend on the library since the Guava class List is being used in the highlighted line.

  In calculating recall during evaluation of the tools, we did not consider "Next" marked artifacts to be in our estimated impact set. This is reasonable since, analogous to the example, the developer has to review the source code in order to figure out whether these artifacts would be impacted or not. The total number of "Next" marked artifacts for a project

Figure 6.2: Areas marked "Next" and "Impacted" by the impact analysis view.

```java
@Override
protected List<Integer> generateCandidatesNotIncludingIntervalExtremes(int start, int end) {
    List<Integer> values = Lists.newArrayList();
    for(int j = start+1; j<end; j++){
        values.add(j);
    }
    return values;
}
```

Figure 6.3: Method from the class AlwaysFieldValueGenerator in project cron-utils that is dependent on the class List from the Guava library.

```java
public void testGenerateCandidatesNotIncludingIntervalExtremes() throws Exception {
    List<Integer> values = fieldValueGenerator.generateCandidatesNotIncludingIntervalExtremes(0, 10);
    for(int j = 1; j<10; j++){
        assertTrue(values.contains(j));
    }
    assertEquals(9, values.size());
}
```

Figure 6.4: Method from the class AlwaysFieldValueGeneratorTest that depends on the class AlwaysFieldValueGenerator.

can become extensive, since all dependent neighbour nodes of an impacted artifact are first marked as "Next" by both tools. The developer has to then intervene to verify whether these recommended artifacts are actually transitively impacted or not. Considering all of them to be positives would have increased the recall of both tools at the expense of the precision. Therefore, we decided to ignore all "Next" marked artifacts during our evaluation.

For the three mentioned projects, some of their "Next" marked methods were part of the ground truth, as they were indeed transitively impacted. These methods were therefore changed by developers in their commit for library migration. Ignoring all "Next" marked methods resulted in reducing the recall of our tool for these projects. However, the tools still recommended these transitively affected methods to the developer, which could have been verified after exploring the impacts of library migration. For most of these methods, developers would have been required to make only one extra decision in order to locate the complete impact set.

6.4 Summary

In this chapter we evaluated EDW on real world Java projects, by assessing precision and recall relative to the assumed ground truth data. Our results demonstrate that EDW achieves promising results with perfect precision for all considered granularities, and a recall of 0.86 for the affected methods while performing almost perfectly for the other two granularities. We discussed three factors that can end up affecting the performance of our tool so that they can be improved in future. The recall of our tool can be impacted either by usage of on-demand import declarations or by having affected members at nested granularity; since these are engineering details, these limitations can be easily addressed in future. We also evaluated JRipples to locate the initial impact set of library migration. We found that using Eclipse search with JRipples has comparatively lower precision and results in more searches that require human interpretation. Hence, there are more opportunities for mistakes and confusion for the developers. It remains to be seen if actual mistakes and confusion happen in practice.

Chapter 7

Human-Subjects Experiment

This chapter presents our controlled experiment with human participants, in order to contrast the effectiveness and efficiency of EDW with JRipples. The experiment was approved by the Conjoint Faculties Research Ethics Board at the University of Calgary, under ethics approval REB18-0239.

The rest of this chapter is organized as follows. Section 7.1 states the null and alternative hypotheses for the study. Section 7.2 presents the selected system for our experiment along with the criteria that led to its selection. Sections 7.3, 7.4, and 7.5 discuss the tasks, participants, and data collection of our experiment, respectively. Section 7.6 highlights the procedure we follow. Section 7.7 presents the results of our experiment, followed by limitations of our study in Section 7.8.

7.1 Hypotheses

The null and alternative hypotheses, with respect to RQ5, that we want to answer quantitatively are as follows.

$\mathbf{H}_0^1$: There is no difference in time while locating the initial impact set of a library by using either EDW and JRipples.

$\mathbf{H}_\mathbf{A}^1$: Participants using EDW to locate the initial impact set of a library will complete their task in less time as compared to performing the task using JRipples.

$\mathbf{H}_0^2$: There is no difference in precision and recall for locating the initial impact set of a library by using either EDW and JRipples.

$\mathbf{H}_\mathbf{A}^2$: Participants using EDW to locate the initial impact set of a library achieve better precision and recall for their task as compared to JRipples.

7.2 Subject System

We selected an open source Java project called JUNG[81] (Java Universal Network/Graph Framework) for our experiment. JUNG supports the modelling, analysis, and visualization of data which can also be represented as a graph or network. We selected its released version 2.1, which uses a third-party library named Guava. JUNG is a Maven project with a modular structure; therefore, we selected two of its modules, namely "api" and "io" for our experimental tasks. These modules have different code bases, but follow a similar coding standard due to them being from the same developers. The api module contains 30 classes with 10 kLOC, while io contains 54 classes with 6 kLOC. The selection of JUNG for our experimental tasks is based on the following reasons.

- JUNG is open source with its code publicly available on GitHub; this was a requirement for the tasks in our experiment.
- JUNG is a popular project on GitHub; its code base is standardized with proper commenting and documentation. The presence of self-explanatory variables, classes, and comments can assist an unfamiliar developer in comprehending the general structure of the system.
- We selected Guava for the library migration of our experimental tasks. Guava is a famous utility and collection library provided by Google. Moreover, since the library is an extension to the standard Java collection library, it can be easier for an unfamiliar person to comprehend its usage in a system.

[81] https://github.com/jrtom/jung

Table 7.1: Guava impact set for tasks.

Task	Module	Granularity	Affected
1	api	Class	7
1	api	Method	4
1	api	Field	9
2	io	Class	15
2	io	Method	51
2	io	Field	36

- The selected modules are of reasonable sizes, meaning that the direct impact set for Guava in the modules is also of a size locatable in the available time for the tasks.

7.3 Tasks

There were two different tasks that were performed by the participants during our experiment. Task 1 was performed on the api module, while Task 2 was performed on the io module. Participants were provided 7.5 minutes for the tasks with EDW while they had 30 minutes with JRipples. Table 7.1 lists the size of the Guava direct impact set for the three granularities of the two modules.

The participants were asked to find the complete set of classes and their members (i.e., fields and methods) that directly referenced the types and fields from Guava. Participants were asked to perform this task with both tools on each module, first EDW and then JRipples, to allow any bias (such as learning effects) to benefit JRipples. They were provided with detailed instructions for the tasks with each tool. Since the tasks involved location of the direct impact set, no prior understanding and comprehension of the code base was required before the experimental trials. The description of tasks and their instructions are available in Appendix F.1.

7.4 Participants

We recruited eight graduate students from the University of Calgary for our experiment; Table 7.2 lists their characteristics. The selected participants were familiar with the Eclipse IDE and the Java

Table 7.2: Overview of participants' experience.

Participant	Familiarity			Experience	
	Eclipse	Java	libraries	industry	development
P1	5	4	4	0	5
P2	4	4	4	3	7
P3	4	4	4	0	4
P4	5	5	5	6	8
P5	4	4	3	0	3
P6	3	4	5	0.6	5
P7	4	4	5	8	12
P8	4	4	5	4	10

programming language. All the participants were also familiar with third-party libraries in general and had used them in their recent projects. Four participants had industry experience of 3–8 years. Participant P1 had industry experience of 6 months, while two participants (P4 and P7) were also employed in industry contemporarily with being a graduate student. Moreover, participants P1 and P4 indicated that they used Eclipse and Java on a daily basis. The rest of the participants indicated that they had actively used Eclipse and Java in the recent past.

7.5 Experimental Procedure

The experiment started with the participant reading and signing a consent form; we then asked them to fill a quick pre-study questionnaire regarding their experience. Participants were then given a short verbal overview of the experimental setup and the terminology to be used.

The participants were then given training sessions to familiarize them with the tools (EDW and JRipples). First, they were given a demonstration of how the tools work along with a description of the features to be used. Participants were then set to work on a sample task for both tools respectively. During training sessions, participants were allowed and encouraged to ask any questions they had.

Once the participants were comfortable with the tools, we started with the actual experimental trials. The participants were provided with the source code, task description, and instructions to perform each task. We first explained the task and encouraged them to ask any question regarding

the tasks before starting. The participants first performed Task 1 with EDW, for which they were given a time limit of 7.5 minutes. After completing Task 1 with EDW, participants then performed the same task with JRipples for which they were given 30 minutes. They were asked to fill a post-task questionnaire after completing Task 1 for each tool. The participants were then asked to perform Task 2 with both tools in the same order, filling a post-task questionnaire after it. The participants were asked to announce their results once they were done, after which we recorded the completion time while saving the state of the used tool. The participants recorded their results by populating the hierarchical view of the tools; this view contains all classes and members of a project under analysis. The located classes and members were marked as "impacted" by the participants during trials; we recorded the state of the view after each task. The announcement of result means that the participant had populated the view and had a final look at it to verify that the results roughly looked reasonable.

After completion of both tasks with each tool, the participants were given a final questionnaire for comparing their performance with the tools. Each participant spent roughly 2–3 hours in total for the experiment and were given an honorarium of 100 CAD as compensation for their time. The details of task description, instructions, and questionnaires are available in Appendix F.

7.6 Data Collection

A variety of data was collected during the experiment; some of it came from our notes while observing the participants' decisions and comments during their trials, while the rest came from the state of the Eclipse IDE after each task completion. We did not record any audio or video of the participants during the experiment, as it was deemed too expensive to analyze and too difficult to get ethics approval for as it would have possibly made the participants less comfortable during the experimental trials.

7.7 Results

This section presents the quantitative and qualitative results for our study.

7.7.1 Quantitative

Hypotheses H_0^1 and H_A^1

Table 7.3 lists the individual time taken in minutes by the participants for performing the two tasks with EDW and JRipples. Table 7.4 lists the average of the time taken in minutes for performing the two tasks with tools respectively. Participants on average took ~3 minutes for performing a task with EDW; they took 12.8 and 25.9 minutes for performing Tasks 1 and 2, respectively, with JRipples. While using JRipples, participants were able to perform Task 1 in comparatively less time than Task 2, as Task 1 had an overall smaller direct impact set. However, with EDW the participants on average required roughly the same amount of time for both tasks.

We conduct a *t*-test [Helmert, 1875] for dependent samples, in order to test the null hypothesis

Table 7.3: Time taken by individual participants for tasks (in minutes).

Participant	EDW Task 1	JRipples Task 1	EDW Task 2	JRipples Task 2
P1	2	11	3	27
P2	4	10	3	30
P3	3	13	3	30
P4	3	19	4	29
P5	3	15	2	19
P6	2	10	3	21
P7	2	11	2	27
P8	2	13	2	24

Table 7.4: Average time taken in minutes for tasks.

Tool	Task 1	Task 2
EDW	2.6	2.8
JRipples	12.8	25.9

(we use an online calculator[82] to perform the calculation). The assumptions of t-tests for dependent samples are that the data are normally distributed and belong to the same experimental block. The time collected for EDW and JRipples is paired, as the same tasks were performed using both tools. We use the Kolmogorov–Smirnov (KS) goodness-of-fit test [Karson, 1967] to test the normality of our collected data (we use an online calculator[83] for the calculation); we use the null hypothesis that the data *is* normally distributed and the alternative hypothesis that it is not. Table 7.5 lists the p-value for the KS test results with an α level of 0.01. (The α-level specifies the probability of detecting a difference from the null hypothesis.) As $p > 0.01$ in all cases, the null hypothesis cannot be rejected: the data is normally distributed.

Table 7.5: Results of the Kolmogorov–Smirnov test for elapsed time.

Data source	EDW p-value	JRipples p-value
Task 1	0.33	0.71
Task 2	0.51	0.67
combined	0.13	0.62

Table 7.6 lists the results of one-tailed t tests on the time taken for the tasks, as given in Table 7.3. At an α level of 0.01, $p < 0.01$; hence we reject the null hypothesis H_0^1. The results show that participants using EDW to locate initial impact set of a library will complete their task with significantly lower time as compared to performing the task with JRipples.

[82] https://www.socscistatistics.com/tests/ttestdependent/Default.aspx
[83] https://www.socscistatistics.com/tests/kolmogorov/Default.aspx

Table 7.6: Results of t tests on time taken.

Data source	mean	std deviation	t	p
Task 1	7.68	5.65	9.55	0.00003
Task 2	14.2	12.2	17.1	0.00001
combined	11	9.99	8.87	0.00001

Hypotheses H_0^2 and H_A^2

Table 7.7 lists the average precision, recall, and F-score for the three granularities at which we apply JRipples. Appendix G lists the details of eight participants for each granularity of the two tasks with JRipples. EDW achieved a precision and recall of 1 for the selected system. On the other hand, participants made mistakes with JRipples due to manual work and errors.

For Task 1 JRipples achieved a recall of 0.82, 0.85, and 0.69 for class-, method-, and field-granularity, respectively. For Task 2 JRipples achieved a recall of 0.78, 0.52, and 0.65 for class, method, and field granularity, respectively. The precision suffered for method- and field-granularity of Task 1. None of the participants were able to find all impacted methods for Task 2; only P7 was able to achieve a recall of 0.78 with a precision of 0.86. In contrast, with EDW, all participants were able to achieve perfect precision and recall. The effectiveness of performing a task with JRipples apparently depends on its complexity: a task with a bigger impact set and system can lead the developer to more FPs and FNs.

The data for the F-score is not normally distributed because EDW achieved an F-score of 1 for all participants. Therefore, we can not perform a *t*-test for calculating significance. We instead use the alternative Wilcoxon signed rank test [Wilcoxon, 1945] (we use an online calculator[84] to perform the calculation), a non-parametric test to evaluate if two dependent samples were selected from populations having the same distribution; it does not require the data to be normally distributed. Table 7.8 lists the results of the Wilcoxon test on the F measure for the three granularities, calculated

[84] https://www.socscistatistics.com/tests/signedranks/Default2.aspx

Table 7.7: Average JRipples task result.

Granularity	Task	P	R	F
Class	1	1	0.82	0.86
Method	1	0.72	0.85	0.74
Field	1	0.75	0.69	0.73
Class	2	0.97	0.78	0.85
Method	2	0.92	0.50	0.63
Field	2	0.99	0.65	0.73

Table 7.8: Results for Wilcoxon test on F-score data combined from Task 1 and 2.

Granularity	p
Class	0.02
Method	0.0004
Field	0.002

individually. Note that, for class-granularity, there are only 6 non-tied samples (two from Task 1 and four from Task 2); therefore, the p-value calculated by the Wilcoxon test is not exact.

For method- and field-granularity, there was a significant impact on F-score while using EDW in contrast to JRipples at an α-level of 0.01, as $p < 0.01$. For class-granularity, there was a significant impact at an α-level of 0.05, as $p < 0.05$. Because of this, we reject the null hypothesis H_0^2. The results show that the participants using EDW to locate the initial impact set of a library migration achieve better precision, recall, and F-score in their tasks as compared to performing the task with JRipples.

7.7.2 Qualitative

Table 7.9 summarizes the answer to the question "How did you find the task", asked in the post-task questionnaire after each task (see Section F.3). Note that "1" means the participant found the task to be difficult, whereas "5" means the participant found the task to be easy. The table shows that while using JRipples, most of the participants (except P4, and P3 for Task 1) found the tasks to be difficult or moderately difficult. With EDW, all participants found the tasks to be easy or moderately easy. This shows that the participants perceived the performed tasks to be relatively easier with EDW in contrast to JRipples.

For the question "Did the task take more time or less then you anticipated at the beginning?", asked in the post-task questionnaire after each task (see Section F.3), all participants answered "less" for the tasks with EDW and most answered "more" for the tasks with JRipples (except P4 in Task 1). This is consistent with the results in Table 7.4, which show that participants took more time for the tasks when using JRipples. For the last 4 questions in the final questionnaire (see

Table 7.9: Results for the post-task questionnaire.

Participants	EDW Task 1	JRipples Task 1	EDW Task 2	JRipples Task 2
P1	5	1	5	1
P2	5	1	5	1
P3	5	2	5	1
P4	5	4	5	3
P5	5	3	5	1
P6	4	2	5	1
P7	4	1	5	1
P8	5	2	5	1

Section F.4), most participants strongly preferred EDW; they strongly agreed that they were more likely to succeed with EDW in estimating the changes needed; they strongly agreed that they could attempt larger changes and modification tasks with EDW; and they strongly agreed that they would use EDW again. The only exception was P4 who only agreed (not strongly agreed) for the last three questions. This shows that the participants found EDW to be helpful for the tasks, in contrast to JRipples.

Observations during the experiment

For JRipples, all participants used a similar approach for locating the direct impact set, by relying on the search functionality provided by the Eclipse IDE. At first they started with manually going through the classes iteratively, until they found a class that was directly referencing Guava. Once the participants had a general idea about the usage of Guava in the system, they used IDE search for locating the impacted classes while marking them manually in the JRipples view. Apart from P1, all other participants used a variant of either search strategy 2 or 3 (see Section 6.2.3) as a query for IDE search. P1 used search strategy 1, while wrongly assuming that the only import usage of Guava in the system was the one they had found in the first manually located class; however, P1 discovered some other imports during the task which they had initially missed. P1 then performed another search while using the newly discovered imports, while commenting that they were not sure about the success of the task any more. Participants P3 and P6 wrongly localized their IDE

search for Task 1 to only one of the packages of the system; therefore, they were unable to get a complete impact set of the system. P3 corrected this mistake for Task 2, but still did not mark the complete set of impacted classes due to confusing classes with similar names. P8 initially tried to remove Guava from the build path of the system, in order to inject errors into the system; their idea was to locate the classes that would cause errors after removal of the library. However, P8 had to stop after a few unsuccessful tries due to limitation of time. P8 then decided to use Eclipse IDE search for completing the task.

Once the participants found the directly referencing classes, they searched the directly referencing members. Apart from P7 and P8 all other participants did this by either manual traversal or IDE file search of a marked class for each of its Guava imports. Participants P7 and P8 used an interesting approach to locate the affected members based on their experience; they removed all the Guava imports from a class in order to inject errors. They then searched the members for which Eclipse was giving them errors. This approach was comparatively faster; however, it resulted in more false positives: the IDE would signal an error for any method that uses a defined field from Guava in the class. The approach therefore signals errors for both directly and transitively impacted methods; manually validating the large number of methods resulted in more false positives. Even though participants P7 and P8 were careful to mark only the direct impacted members, they still ended up marking some false positives during the task.

The participants complained that they were not confident about the success of their task while doing it with JRipples. The JRipples view does not distinguish between overloaded methods defined inside a class; therefore, the user is required to manually cross-check each overloaded method. Some participants thus complained that they got confused when a class contained overloaded methods.

On the other hand for EDW, the participants were able to perform both tasks easily due to its automation. Some participants were skeptical about the results of EDW; therefore, they decided to randomly check a few classes and announced the results only after making sure that the tool was working fine.

Characterizing good tasks as bad and vice versa. In order to review how sure participants were about their performance of a task with a tool, we explored their answer to the post-task question, "How well do you think you did the task?" We categorized the actual performance of a participant as good if they were able to achieve more than 0.7 recall in any two granularities. Table 7.10 lists the categorization of both tasks for each participant for JRipples. The columns partially labelled "us" show our evaluation of the results, while those partially labelled "them" show participants' self-evaluation after completing the task. The cell values "G", "B", and "A" signify that the participant's performance in the task was good, bad, or average, respectively. "NS" means that the participants were not sure about how they did. Therefore, we interpret any cases of "NS" as the developer having performed a bad task in their own opinion.

For EDW all participants said they thought they performance was good, because the tool automated the task for them. However, for JRipples the participants were not clear about their performance. For Task 1, only P4 correctly characterized the good performance as good in the post-task questionnaire; the rest of the participants either had good performance for the task but characterized it as bad, or vice versa. Participants P1, P2, P5, P7, and P8 characterized that they had bad performance for Task 1, even though they actually performed it pretty well. Participants P3 and P6 said the opposite, that they had good performance while in reality they did not. For Task 2 participants P1 and P6 correctly characterized their bad performance. P3 and P5 claimed that they had average performance while in reality they did not. Participants P2, P4, P7, and P8 had good performance for Task 2, but none of them were aware of this.

These results show that the majority of the participants were not sure about the success of a task after completing it with JRipples, and thus that developer opinion is not a trustworthy measure of success for determining the direct impact set during library migration.

Interesting comments and quotes. We selected some interesting comments made by the participants regarding the used tools for further discussion.

Some participants pointed out the advantages of using JRipples. P6 said that JRipples helps in

Table 7.10: JRipples task categorization.

Participant	Task 1: us	Task 1: them	Task 2: us	Task 2: them
P1	G	B (NS)	B	B
P2	G	B (NS)	G	B (NS)
P3	B	G	B	G
P4	G	G	G	B (NS)
P5	G	A	B	A
P6	B	G	B	B
P7	G	B (NS)	G	B
P8	G	B	G	B

showing the code structure of the selected project. Similarly, P4 commented that "JRipples helps in record keeping" of the impact set. P3 stated that "JRipples finds automatically the dependent class after marking a class impacted". Note that all these features are inherently available in EDW as well, as it is built on top of JRipples.

Some participants pointed out disadvantages of using JRipples. P2 commented that, based on their experience, "In a complex project with too many dependencies on a library, it is impossible for JRipples because of manual search of impacted classes"; they were concerned about generalization of JRipples to bigger systems that have a significant number of classes. P6 got confused while marking overloaded methods in a class, as JRipples does not distinguish between them in its view; they commented that "In JRipples there were times when I found an impacted code element but after I noticed that there is another element of code with the same name which caused problem in tagging the real impacted ones. I didn't have this problem with EDW". Similarly, P7 mentioned the issue with manual work, "For JRipples I had to get into each class and find the impacted set. Also had to comment out the import to break the code and identify which lines are impacted". While P3 faced problems while trying to find imports from Guava by stating that they were "looking for a way to find the Guava library with the right name 'google'."

For EDW the participants pointed out some benefits. P7 stated that EDW was handy for migration, as "The tool is very helpful. It is good to be able to see all the third party libraries in one place." P8 commented that "EDW was much easier in three clicks you can find the impacts". However,

some participants wanted further improvement in the process of EDW. P4 was not happy with the number of clicks and asked to further "Reduce the number of clicks". P6 believed that "Even more automation can be useful".

7.8 Limitations

EDW automates the search for the direct impact set of a library; therefore, there is minimal chance of a learning effect while performing similar tasks with JRipples. However, we took a number of steps to reduce the potential learning effect (in favour of JRipples) during our experiment. First we did not tell participants that they would perform similar tasks for both tools. This step was taken to avoid them from consciously remembering the results from EDW. Second, we tried to make sure that the participants did not thoroughly explore the results from EDW for a task. As the participant were given 7.5 minutes with EDW for the tasks, it was highly unlikely for them to consciously memorize the impact set in this time.

Our study consisted of only 8 participants with all being graduate students. Only half of the participants had significant industry experience; therefore a study with more participants and specifically industrial developers is needed to generalize the results.

7.9 Summary

We performed a controlled experiment with 8 participants, in order to compare the effectiveness and efficiency of EDW with JRipples. Participants were asked to find the complete set of classes and their members that directly referenced the types and fields from Guava. The experimental trials required the participants to perform this task on two modules namely "api" and "io" of the project JUNG. Participants first used EDW for performing the task followed by doing it again with JRipples. Participants were asked to provide feedback on the used tool after each trial, and at the end to provide final feedback on both tools. We recorded time and confusion matrix as measures of efficiency and effectiveness.

Participants were able to perform the tasks with higher efficiency and effectiveness while using EDW, in comparison to JRipples. Compared to JRipples, EDW significantly improves the time required for the task. Participants on average took only ~3 minutes for performing a task with EDW. On the other hand, they spent on average 12.8 and 25.9 minutes for performing Tasks 1 and 2 with JRipples, respectively. Similarly, EDW significantly improves precision and recall in contrast to JRipples. For Task 1, JRipples achieved an F score of 0.86, 0.74, and 0.73 for class-, method-, and field-granularity, respectively. For Task 2, JRipples achieved an F score of 0.85, 0.63, and 0.73 for class-, method-, and field-granularity, respectively. However, with EDW participants achieved a F score of 1 due to the automation provided by the tool.

Chapter 8

Discussion

In this chapter we discuss remaining issues related to our empirical study on library migration and the proposed tool. Section 8.1 discusses trends in the usage of external libraries. Section 8.2 explains the breaking of source code as a consequence of migration and highlights the type of libraries that have a high probability of breaking client code. Section 8.3 highlights the approaches used for notification of library migration. Section 8.4 describes issues with our tool and our evaluation thereof. Sections 8.5 and 8.6 discuss threats to validity of our work, and future work of our study, respectively.

8.1 Trends in Software Library Usage

We found a trend in the choice of external libraries being used by developers of Android projects. The libraries being used mostly belonged to the domains of network communication, graphical user interfaces, data encoding, and data structures. In contrast, we found no such trends for the Maven projects. The libraries being used in Maven projects belonged to diverse domains and were harder to classify. We note that the available library domains in Android are restricted due to the structure of the platform, as most of the used libraries perform narrower tasks compared to their counterparts found in Maven. This observation is strengthened by the fact that library migration did not result in breaking transitively dependent client code in any of the Android projects.

Table 8.1: Preferred libraries involved in Android migrations.

Library	Alternative	Projects
OkHttp	standard Java	9
NineOldAndroids	standard Android support	6
ActionBarSherlock	standard Android support	5
Retrofit	standard Java	4
Picasso	Glide	3

Furthermore in Android, we also found a trend in the selection of alternative libraries being introduced to replace an existing one. Table 8.1 summarizes these libraries involved in Android migrations along with the alternative that replaced them. Different developers consistently opted for the same alternative, but we note that there are not many libraries available for the Android platform. It is interesting to note that a large majority of projects had to migrate to the standard SDK support, due to scarcity of external libraries. The trend shows that the Android community is still maturing in terms of software libraries and there are not that many viable substitutes available in the market.

8.2 Code Breaking of Systems and Their Clients

Our findings regarding breaking of source code show that sometimes the system can couple strongly with a software library; as a result their migration can have severe impacts on the dependent code. Our analysis shows that these impacts can even propagate further to transitively dependent systems in cases where APIs are coupled with the external library.

Our results highlight two important points: first, it shows that library migration would often break dependent code; second, it raises a red flag for developers to be careful while using an external library. While using a library has short-term benefits, in the long-term migration of this dependency can have severe consequences for both the developer and their clients.

Table 8.2: Domain of libraries that have a higher probability of breaking transitively dependent clients.

Domain	Examples
data encoding	JSON.simple,JAXB
data structures	Guava, Apache Commons
web asset	Jetty, Spring

8.2.1 Categories of Libraries that Broke Clients

We found three categories of libraries that had a higher probability of breaking transitively dependent clients. Breaking of client projects mostly occurred due to dependencies belonging to the three categories listed in Table 8.2. These categories are taken from Maven Central Repository, which uses them in organizing and searching the repository.

Generally libraries from these categories are directly used by clients and their removal from the system can result in discrepancies. For example, Java Script Object Notation (JSON) is a format for data encoding , that is commonly used for communication among platforms. JSON.simple is a library for JSON provided by Google. JSON.simple provides a data structure named JSONObject, that contains the attribute-value mapping of the encoded data. A client project receiving or utilizing JSONObject from a systems' old version would expect the same object containing the similar format of data mapping from the newer version; therefore, a change in object type or format due to migration to an alternative JSON library would most likely result in breaking the client code. Note: Each JSON library has a different data structure for containing the attribute-value mapping. For example the library GSON provides an object named Gson for this purpose.

We recommend that these libraries should be used with a dependency injection framework to limit their impact on client code.

8.3 Notification of Software Library Migration

We found that developers are uncommunicative about library migration. Even though release notes are a standard way of informing client about changes, developers still under use them. They prefer

using pull requests and internal threads for discussing about migrations. Similarly, the rationale behind migration, if available, is more frequently offered through pull requests than in release notes.

These findings do not coincide with the analysis that dependency migration ended up breaking client code, since one would expect the project developers to notify their clients about migration, in order to warn them of incompatibility issues while using the new release. We speculate that due to lack of awareness of their external importance, project developers do not use release notes that often. The project developers do not consider migration of dependencies as a major change and as a result avoid informing the clients about it, even though the reality runs contrary to this assumption. As a result, clients face incompatibility issues when using new releases of systems that have migrated from external libraries.

This under use of notification resources might be either due to lack of knowledge or the unavailability of a systematic approach for library migration. If the project developers start following a systematic incremental process to handle this change, the migration could be performed properly with appropriate notification to all involved stakeholders.

8.4 Evaluation of EDW

In our evaluation, EDW works well for locating the affected impact points, due the fact that EDW makes use of direct import references for locating the impacted classes. The recall for EDW can be impacted negatively by on-demand import declarations and nested granularity, as explained in Section 6.3.2. However, since these are both minor engineering details, the limitations can be easily addressed in future.

We evaluated JRipples for locating the initial impact set of library migration, in contrast to EDW. We found that EDW significantly outperforms JRipples in three areas. First, EDW improves the time required for locating the direct impact points of a library. Second, the precision and recall of EDW is comparatively better. Third, EDW does not require human intervention while locating the direct impact points; as it automates the process. On the other hand, JRipples relies on

manual Eclipse search for locating the affected classes. Moreover, it also requires the developer to manually record/memorize the point of impacts of affected members; therefore, JRipples requires more decisions from developer, each of which represents an opportunity for confusion and making mistakes.

8.5 Threats to Validity

In this section we discuss the threats to validity of our research. Section 8.5.1 highlights the threats to validity of our empirical study on software library migration, while Section 8.5.2 discusses the threats to validity for the evaluation of EDW.

8.5.1 Threats to validity of the library migration study

The accuracy of our empirical study on dependency migration depends on the selection of systems, the accuracy of the tools we used, and the measures we used for effort estimation.

Selected systems

We defined and followed criteria while selecting projects for our study; we note three issues that arose due to our choices. (1) We chose only open source systems; therefore, our findings may not reflect how external dependencies are treated in closed and corporate systems. It is possible that closed and corporate systems might be hesitant to use external libraries due to issues with licensing and copyright; further study would be needed to test this conjecture. (2) We only selected Maven and Android projects for our study; as a result, our findings might not generalize across other languages and platforms. (3) We only considered popular and stable products with a history of at least two years; therefore, our results might not be applicable to relatively new or unpopular projects.

The accuracy of third party tools

Our findings are dependent on the third party software that we used: Find it EZ, JAPICC, and Code Compare. We used Find it EZ to locate the affected classes which were then utilized to figure out the impacts of library migrations. JAPICC was used to find the probability of API breakage along with the compatibility issues that arose in client code due to migrations. Code Compare was used to determine the discrepancies resulting from migration between the impacted classes of the project. The accuracy of our analysis depends on how well these tools performed their respective functions. To validate the results, we manually checked a few random results from these tools and found them to be accurate. However, checking all the outputs from the tools was infeasible; as a result, the accuracy of the third party software remains a threat to validity for our study.

Measurement of effort

Two measures (the number of changed classes and lines of code) were used to estimate the effort required by developers during library migration. We acknowledge that estimating migrations through these measures might not be the best indicator of effort. Certainly, our own experience indicates that migrations involving fewer affected artifacts (classes and LOC) can be more complex and time consuming than those that impact more artifacts. Our choice for these measures was due to lack of a better alternative options that were easily achievable. Therefore, the results of our study should be taken as an estimate of effort.

8.5.2 Threats to validity for the evaluation of EDW

The accuracy of the evaluation of EDW is impacted by two factors: (1) selection of evaluation data set and (2) manual analysis by a single experimenter

Selection of evaluation data set

We evaluated our tool on the first ten projects that satisfied our criteria mentioned in Section 6.1. However, the results of our evaluation still depend on the projects we selected. The size of the evaluated data set is also not large and could be improved in future work.

Manual evaluation by a single experimenter

The evaluation of EDW was performed manually by the author. To avoid researcher bias, we used developers' commits of the library migration as the ground truth, and we wrote a comparator script for compilation of the data that populated the confusion matrix and that generated precision and recall for our results. However, the qualitative analysis of the data sources was performed by a single researcher. Therefore, this stands as a threat to validity for our evaluation of EDW.

8.6 Future Work

In this section we discuss the future scope of our work.

8.6.1 Knowledge base for external libraries

A large number of software libraries providing similar functionalities are available online. These libraries vary in their performance, platform, resource consumption, and supported use cases. Currently developers have to search various forums and repositories for finding an optimal library that fits their systems need. This search results in developer wasting a lot of time and resources on a non-productive task. Lack of time can even result in developer making a rash selection, which would create repercussions for the product in future.

We would like to create a queryable central knowledge base consisting of all the available libraries. The knowledge base would contain real time data regarding the history of performance, bug, vulnerabilities, patches, and popularity of the classified libraries. Such a system would allow the developers to easily compare popular alternatives available for a domain before making a final

decision. This centralization can guide developers while making selection of an external library and can reduce the required effort and time.

8.6.2 Improve effort estimation

The measure used for estimating effort invested by developers for mitigating library migration needs further improvement. We would use better alternatives such as expert- and combination-based models for measuring effort in future work.

8.6.3 Improvements to EDW

Our tool does fairly well in terms of precision and recall. However, there is still scope for future work. Some of the areas that can be improved are as follows.

Search strategy

The search strategy being used to locate affected classes in EDW is rather simple. We make use of Eclipse IDE search to locate any reference of external library in the system. The search cannot be customized per use case as it does not support filters or wildcards of any kind. Therefore in future work we would improve the basic search by supporting filters, customization, and wildcards for the user.

Support for nested granularity

The current version of EDW does not support marking a class member that is declared at a nested granularity. The support of nested members would make the analysis performed through EDW more thorough. Incorporating this into EDW would require changing our impact analysis module. Integrating a mode for selection of different granularity level would allow this the tool to support nested granularity in future.

Extension Points

Eclipse extension points allow contribution of further functionality to an existing plugin. We have provided extension points in EDW to support future improvements. The graph builder and metrics builder modules can be extended by future researchers through these extension points. The graph builder module can create a dependency graph for one affected class at a time. Therefore, a possible extension can be to support graph generation for multiple affected classes present in a module, package or the complete system. This extension can allow the user to feasibly switch between different granularity of graph representation while performing the library migration. Similarly, the metrics builder module supports metrics that are related to usability of a single library; therefore, that module can be extended to support system-wide metrics such as the total number of dependencies, unnecessary dependencies, or coupling of dependencies. The system-wide metrics could provide an overview to the developers regarding the management and usability of dependencies in a project.

Furthermore, currently EDW only supports Java- and Maven-based projects. It can be extended to other platforms to further support library migration. The ideal candidate would be platforms that store the information related to external libraries such as Apache Ant, Gradle, and Android.

Human-Subjects Experiment

The human study introduced in chapter 7 involved only 8 participants. The participants were all graduate students with only half having significant industry experience. Therefore, a study with more participants and especially industry developers is required to generalize the results.

8.7 Summary

In this chapter we highlighted several remaining topics related to our research. We discussed the found trends for the domain of libraries being used in android projects. We highlighted the breakage of source code in the system and their clients resulting from migration. We also stated the found

patterns in the domain of libraries that have a higher probability of breaking transitively dependent systems. We further discussed our findings regarding the notification of library migrations from developers. We also explained the evaluation of EDW and highlighted the limitations of JRipples and EDW.

We highlighted the threats to validity of our study. Our data analysis on library migration is affected by the choice of selected systems, accuracy of third party tools and the used measures for effort estimation. On the other hand, the threats to validity of our tool are due to selected system for evaluation and analysis by a single experimenter.

Finally we discussed the future work for our research and highlighted how it can be further improved.

Chapter 9

Conclusion

Third party software libraries are used by developers for the sake of reuse [Ebert, 2008], resulting in improved development time and maintainability [Mohagheghi and Conradi, 2007]. However, libraries are subject to deprecation and evolution, resulting in newer releases. As a consequence, developers have to consider migrating to better alternatives in the market. But the difficulties of dependency management are greatly underestimated by developers, and most software systems keep their outdated libraries [Kula et al., 2018b]. Existing work on library migration is restricted to updates between versions of the same library, and does not considering when, whether, or why software systems migrate from one library to another. Thus, until now, we had little evidence and understanding of this phenomenon.

The first part of this thesis comprises a retrospective study of library migration in 114 open source, Java-based software systems. The study explores the consequences of migration on both directly and transitively dependent systems, the developers' rationales behind such migrations and the effort required to mitigate such changes.

We found that many projects undergo library migration (11.9% of the studied projects) and that 67% of the migrations ended up breaking the dependent system. To avoid library migration in future, in 43% of the migrations their developers decided to avoid introducing a new alternative external library. Migrations of external libraries are performed almost always for various, identifiable,

reasonable motivations (i.e., the existence of a better alternative). However, developers tend to be uncommunicative about these changes: they under-use release notes for notifying client users about such library migrations and they even under-use other communication channels (such as pull requests) that are more oriented towards other project developers. 22% of the studied library migrations impacted the dependent client projects; these migrations resulted in breaking of APIs that caused backwards compatibility problems for the client systems. Mitigation of library migration required significant effort from the consuming developers; on average, these developers had to change 9.3% of the classes in their systems while modifying 15% of the total lines of code.

As a corollary to the lack of knowledge about library migration, we lack specialized tool support aimed at helping developers during library migration. Therefore, the second part of the thesis deals with solving the problem of library migration by providing a prototype tool, called EDW. EDW utilizes the concept of change impact analysis for locating areas affected by library migration. Through EDW, the user can view all external dependencies of a project along with their direct and transitive impacts. Developers are notified about the impacted LOC by highlighting, and the tool allows them to change affected artifacts through the editor. The user can also estimate the effort required for transformation through metric module, this can allow them to perform a cost worth analysis before performing a migration. Our evaluation shows that EDW has a promising precision of 1 across all considered granularities. For recall, it obtained over 0.99 for class- and field-granularity while achieving ~0.86 for method-granularity. Using Eclipse search with JRipples on the other hand, results in similar recall but with comparatively lower precision. Moreover, JRipples results in more searches that require human interpretation; hence, there are more opportunities for mistakes and confusion for the developers.

We conducted a controlled experiment with human participants, in order to compare the efficiency and effectiveness of EDW against JRipples, an existing tool for structurally-based change impact analysis. Participants were able to perform the tasks with higher efficiency and effectiveness while using EDW, and in significantly lower time. Participants on average took only ~3 minutes for performing a task with EDW, but with JRipples they spent on average 12.8 and 25.9 minutes

for performing the two tasks posed to them. Similarly, EDW significantly improves precision and recall in contrast to JRipples: EDW resulted in perfect precision and recall in terms of the initial impact points, while JRipples achieved F-scores varying between 0.63 and 0.86, depending on the task pursued and the granularity at which the measurements are taken.

The two contributions of this thesis are:

- We empirically explored library migration in 114 Java-based projects, selected from both Maven and Android platforms in order to diversify the sample.
- We developed a prototype tool, EDW, to assist the developer in performing library migration. EDW utilizes change impact analysis for locating the impacted areas of change for the user, and provides resources through which library migration can be performed systematically. We conducted both a simulation and an experiment with human participants, in order to evaluate EDW. The high precision and recall along with significant time savings of our tool showed that it has promise in guiding developers during the process of library migration.

9.1 Future Work

In order to support developers in their search for an appropriate software library, there is need for a centralized knowledge base containing real time data regarding the history of performance, bugs, vulnerabilities, and popularity of classified libraries. The knowledge base would allow developers to compare the available alternatives for a library domain before making a final selection.

EDW does well in terms of precision and recall; however, there is still scope for its future improvement. The search strategy being used by EDW to locate affected classes is rather basic. The basic search can be extended in future to support filters, customization, and wildcards for the user. Similarly, the current EDW version does not support marking a nested class member; such support could be incorporated into EDW to make its analysis performed more thorough. In order to support such future work, we have provided extension points in EDW. The graph builder and metric modules currently support a singular selected library. Therefore, they can be further extended to support

system-wide graphs and metrics. The system-wide graphs and metrics can provide an overview to the developers regarding the management and usability of dependencies in a project.

The prospects for improving support beyond locating the initial impact points are limited, due to the fundamental limits of computability for change impact analysis. Any future improvements in the general context of change impact analysis could be applied here, too.

The human study performed for the evaluation of EDW, only involved 8 graduate students, with only half having significant industry experience. Therefore, a study with more participants, having significant industry experience is required to generalize the results. Finally, EDW only supports Java- and Maven-based projects. It can be extended to support other platforms such as Apache Ant, Gradle, and Android.

Bibliography

Nemitari Ajienka and Andrea Capiluppi. Semantic coupling between classes: Corpora or identifiers? In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pages 40:1–40:6, 2016. doi: 10.1145/2961111.2962622.

Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

L. Badri, M. Badri, and D. St-Yves. Supporting predictive change impact analysis: A control call graph based technique. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 167–175, 2005. doi: 10.1109/APSEC.2005.100.

Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the Apache community upgrades dependencies: An evolutionary study. *Empirical Software Engineering*, 20(5):1275–1317, October 2015. doi: 10.1007/s10664-014-9325-9.

C. Bogart, C. Kästner, and J. Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Proceedings of the International Conference on Automated Software Engineering Workshop*, pages 86–89, 2015. doi: 10.1109/ASEW.2015.21.

Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, 2016. doi: 10.1145/2950290.2950325.

Shawn A. Bohner and Robert S. Arnold. *Software change impact analysis*. 1996.

B. Breech, A. Danalis, Stacey Shindo, and Lori Pollock. Online impact analysis via dynamic compilation technology. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 453–457, 2004.

J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich. JRipples: A tool for program comprehension during incremental change. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pages 149–152, 2005.

Gerardo Canfora and Luigi Cerulo. Impact analysis by mining software and change request repositories. In *Proceedings of the IEEE International Software Metrics Symposium*, pages 29:1–29:9, 2005. doi: 10.1109/METRICS.2005.28.

Bradley E. Cossette and Robert J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 55:1–55:11, 2012. doi: 10.1145/2393596.2393661.

Fernando López de la Mora and Sarah Nadi. Which library should I use?: A metric-based comparison of software libraries. In *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results Track*, pages 37–40, 2018. doi: 10.1145/3183399.3183418.

Danny Dig and Ralph Johnson. How do APIs evolve?: A story of refactoring. *Journal of Software Maintenance and Evolution*, 18(2):83–107, March 2006. doi: 10.1002/smr.v18:2.

C. Ebert. Open source software in industry. *IEEE Software*, 25:52–53, May 2008. doi: 10.1109/MS.2008.67.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its

use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987. doi: 10.1145/24039.24041.

Istvan Forgacs and Tibor Gyimothy. An efficient interprocedural slicing method for large programs. Technical report, 1997.

Brian Fox. Now available: Central download statistics for OSS projects, 14 December 2010. URL https://blog.sonatype.com/2010/12/now-available-central-download-statistics-for-oss-projects/.

Simos Gerasimou, Maria Kechagia, Dimitris Kolovos, Richard Paige, and Georgios Gousios. On software modernisation due to library obsolescence. In *Proceedings of the International Workshop on API Usage and Evolution*, pages 6–9, 2018. doi: 10.1145/3194793.3194798.

Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean GHTorrent: GitHub data on demand. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 384–387, 2014. doi: 10.1145/2597073.2597126.

Jurgen Graf. Speeding up context-, object- and field-sensitive SDG generation. In *Proceedings of the IEEE Working Conference on Source Code Analysis and Manipulation*, pages 105–114, 2010. doi: 10.1109/SCAM.2010.9.

Alex Gyori, Shuvendu K. Lahiri, and Nimrod Partush. Refining interprocedural change-impact analysis using equivalence relations. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 318–328, 2017. doi: 10.1145/3092703.3092719.

Frederick M. Haney. Module connection analysis: A tool for scheduling software debugging activities. In *Proceedings of the Fall Joint Computer Conference, Part I*, pages 173–179, 1972. doi: 10.1145/1479992.1480016.

F. R. Helmert. Über die bestimmung des wahrscheinlichen fehlers aus einer endlichen anzahl

wahrer beobachtungsfehler. *Zeitschrift für Angewandte Mathematik und Physik*, 20:300–303, 1875.

A. Hora and M. T. Valente. Apiwave: Keeping track of API popularity and migration. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 321–323, 2015. doi: 10.1109/ICSM.2015.7332478.

André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse. How do developers react to API evolution? a large-scale empirical study. *Software Quality Journal*, 26(1):161–191, March 2018. doi: 10.1007/s11219-016-9344-4.

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990. doi: 10.1145/77606.77608.

James W. Hunt and M. Douglas McIlroy. An algorithm for differential file comparison. Computer Science Technical Report 41, Bell Laboratories, June 1976. URL http://www.cs.dartmouth.edu/~doug/diff.pdf.

J. Jasz, A. Beszedes, T. Gyimothy, and V. Rajlich. Static execute after/before as a replacement of traditional software dependencies. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 137–146, 2008. doi: 10.1109/ICSM.2008.4658062.

Kamil Jezek, Jens Dietrich, and Premek Brada. How Java APIs break: An empirical study. *Information and Software Technology*, 65(C):129–146, September 2015. doi: 10.1016/j.infsof.2015.02.014.

Puneet Kapur, Brad Cossette, and Robert J. Walker. Refactoring references for library migration. volume 45, pages 726–738, October 2010. doi: 10.1145/1932682.1869518.

Marvin Karson. *Handbook of Methods of Applied Statistics*, volume 1: Techniques of Computation

Descriptive Methods, and Statistical Inference. John Wiley, 1967. doi: 10.1080/01621459.1968.11009335.

R. G. Kula, C. D. Roover, D. German, T. Ishio, and K. Inoue. Visualizing the evolution of systems and their library dependencies. In *Proceedings of the IEEE Working Conference on Software Visualization*, pages 127–136, 2014. doi: 10.1109/VISSOFT.2014.29.

R. G. Kula, D. M. German, T. Ishio, and K. Inoue. Trusting a library: A study of the latency to adopt the latest Maven release. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 520–524, 2015. doi: 10.1109/SANER.2015.7081869.

R. G. Kula, C. De Roover, D. M. German, T. Ishio, and K. Inoue. A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 288–299, 2018a. doi: 10.1109/SANER.2018.8330217.

Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, February 2018b. doi: 10.1007/s10664-017-9521-5.

William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992. doi: 10.1145/161494.161501.

M. Lee, A. J. Offutt, and R. T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems*, pages 61–70, 2000. doi: 10.1109/TOOLS.2000.868959.

M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the European Workshop on Software Process Technology*, pages 108–124, 1996.

Donglin Liang and M. J. Harrold. Slicing objects using system dependence graphs. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 358–367, 1998. doi: 10.1109/ICSM.1998.738527.

Bennet P. Lientz. Issues in software maintenance. *ACM Computing Surveys*, 15(3):271–278, September 1983. doi: 10.1145/356914.356919.

Wayne C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5): 23–30, September 1994. doi: 10.1109/52.311048.

Andrian Marcus, Vaclav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static techniques for concept location in object-oriented code. In *Proceedings of the International Workshop on Program Comprehension*, pages 33–42, 2005. doi: 10.1109/WPC.2005.33.

Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of API stability and adoption in the android ecosystem. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 70–79, 2013. doi: 10.1109/ICSM.2013.18.

Hao Men. *Fast and Scalable Change Propagation through Context-Insensitive Slicing*. PhD thesis, University of Calgary, Calgary, Canada, November 2018.

Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: A review of industrial studies. *Empirical Software Engineering*, 12(5):471–516, October 2007. doi: 10.1007/s10664-007-9040-x.

M. Petrenko and V. Rajlich. Variable granularity for improving precision of impact analysis. In *Proceedings of the IEEE International Conference on Program Comprehension*, pages 10–19, 2009.

Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1): 5–32, February 2009. doi: 10.1007/s10664-008-9088-2.

S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning versus breaking changes: A study of the Maven repository. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 215–224, 2014. doi: 10.1109/SCAM.2014.30.

V. Rajlich and P. Gosavi. Incremental change in object-oriented programming. *IEEE Software*, 21 (4):62–69, 2004. doi: 10.1109/MS.2004.17.

Václav Rajlich. Modeling software evolution by evolving interoperation graphs. *Annals of Software Engineering*, 9(1–4):235–248, January 2000. doi: 10.1023/A:1018933026438.

Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *SIGSOFT Software Engineering Notes*, 19(5):11–20, December 1994. doi: 10.1145/195274.195287.

Romain Robbes, Michele Lanza, and Mircea Lungu. An approach to software evolution based on semantic change. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pages 27–41, 2007.

Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. *SIGPLAN Notices*, 42(6): 112–122, June 2007. doi: 10.1145/1273442.1250748.

David ten Hove, Arda Göknil, Ivan Ivanov, Klaas van den Berg, and Koos de Goede. Change impact analysis for SysML requirements models based on semantics of trace relations. In *Proceedings of the ECMDA Traceability Workshop*, pages 17–28, 6 2009.

B. Todorov, R. G. Kula, T. Ishio, and K. Inoue. SoL Mantra: Visualizing update opportunities based on library coexistence. In *Proceedings of the IEEE Working Conference on Software Visualization*, pages 129–133, 2017. doi: 10.1109/VISSOFT.2017.23.

P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering*, 29(6):495–509, 2003. doi: 10.1109/TSE.2003.1205178.

Joost Visser, Arie van Deursen, and Steven Raemaekers. Measuring software library stability through historical version analysis. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 378–387, 2012. doi: 10.1109/ICSM.2012.6405296.

Robert J. Walker and Gail C. Murphy. Implicit context: Easing software evolution and reuse. pages 69–78, 2000. doi: 10.1145/355045.355054.

W. Wen, C. Tao, S. Zhang, B. Li, and X. Sun. Change impact analysis based on a taxonomy of change types. In *Proceedings of the IEEE Computer Software and Applications Conference*, pages 373–382, 2010. doi: 10.1109/COMPSAC.2010.45.

Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the International Conference on Software Engineering*, pages 551–560, 2011. doi: 10.1145/1985793.1985868.

Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945. URL http://www.jstor.org/stable/3001968.

L. Xavier, A. Brito, A. Hora, and M. T. Valente. Historical and impact analysis of API breaking changes: A large-scale study. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 138–147, 2017. doi: 10.1109/SANER.2017.7884616.

Appendix A

Studied Systems

Tables A.1 and A.2 respectively list the Maven and Android projects studied in our research. The tables contain general information such as stars, commits, releases, etc. for each project. The URL column for each project lists the group and project name of the system; prepending each with "https://github.com/" would create the URL that can be used to directly access the project repository. We retrieved this corpus before August 2017.

Table A.1: Maven projects.

Project	Stars	Commits	Issues	Releases	URL
admiral	100	2383	20	6	vmware/admiral
allure1	696	1379	5	60	allure-framework/allure1
apollo	1670	1457	11	77	ctripcorp/apollo
auto	5414	826	88	32	google/auto
bitsquare	1401	3359	132	42	bitsquare/bitsquare
brave	666	906	24	60	openzipkin/brave
cassandra-lucene-index	399	1112	25	18	Stratio/cassandra-lucene-index
checkstyle	2174	7060	384	79	checkstyle/checkstyle
commafeed	1457	2497	90	13	Athou/commafeed
crawler4j	2152	299	77	3	yasserg/crawler4j
crawljax	354	1468	68	18	crawljax/crawljax
cron-utils	195	665	15	28	jmrozanec/cron-utils
DCMonitor	189	95	1	4	shunfei/DCMonitor
density-converter	167	123	1	16	patrickfav/density-converter

Continued on next page

Table A.1 (continued from previous page)

Project	Stars	Commits	Issues	Releases	URL
disconf	2732	1156	115	24	knightliao/disconf
Discord4J	237	1235	10	41	austinv11/Discord4J
druid	5185	8009	657	396	druid-io/druid
EasyReport	263	339	11	6	xianrendzw/EasyReport
erlyberly	404	356	19	30	andytill/erlyberly
fb-android-dagger	296	121	6	7	adennie/fb-android-dagger
ff4j	266	698	10	18	clun/ff4j
freedomotic	209	1340	45	8	freedomotic/freedomotic
FXGL	263	1690	56	22	AlmasB/FXGL
galen	1037	1421	107	82	galenframework/galen
geo	183	358	4	12	davidmoten/geo
GeoIP2-java	251	459	4	29	maxmind/GeoIP2-java
GoogleAuth	330	186	5	13	wstrange/GoogleAuth
google-java-format	1002	349	35	5	google/google-java-format
grakn	270	1916	6	18	graknlabs/grakn
greenmail	180	653	25	8	greenmail-mail-test/greenmail
ha-bridge	664	454	45	69	bwssytems/ha-bridge
hangout	198	595	5	17	childe/hangout
hawkular-apm	109	1658	2	27	hawkular/hawkular-apm
Hive2Hive	317	1967	32	6	Hive2Hive/Hive2Hive
HotSUploader	168	837	18	23	eivindveg/HotSUploader
hprose-java	297	436	0	44	hprose/hprose-java
htmlelements	182	464	17	10	yandex-qatools/htmlelements
jacoco	748	1404	56	35	jacoco/jacoco
JCloisterZone	152	896	55	9	farin/JCloisterZone
jmx_exporter	172	169	3	8	prometheus/jmx_exporter
jparsec	174	331	7	11	jparsec/jparsec
jstorm	3055	330	140	23	alibaba/jstorm
kumuluzee	147	387	9	14	kumuluz/kumuluzee
lambadaframework	191	91	15	5	lambadaframework/lambadaframework
leshan	140	643	21	18	eclipse/leshan
lolibox	131	170	2	8	chocotan/lolibox
light-task-scheduler	1139	1032	50	31	ltsopensource/light-task-scheduler
maven-jaxb2-plugin	158	459	45	24	highsource/maven-jaxb2-plugin
metrics	5039	2218	114	62	dropwizard/metrics
MongoDB-Plugin	154	108	15	19	T-baby/MongoDB-Plugin

Continued on next page

Table A.1 (continued from previous page)

Project	Stars	Commits	Issues	Releases	URL
mpush	948	1518	6	8	mpusher/mpush
mvvmFX	179	982	17	19	sialcasa/mvvmFX
netty-zmtp	197	272	0	8	spotify/netty-zmtp
newts	146	552	0	14	OpenNMS/newts
objenesis	199	571	0	14	easymock/objenesis
PixelController	184	2521	16	32	neophob/PixelController
RankSys	146	446	8	7	RankSys/RankSys
requests	159	182	4	15	clearthesky/requests
retrofit	22788	1494	58	40	square/retrofit
rtree	500	1019	11	43	davidmoten/rtree
seyren	861	640	40	8	scobal/seyren
simple-java-mail	188	474	3	14	bbottema/simple-java-mail
SlimFast	170	112	2	12	HubSpot/SlimFast
solo	2954	1829	6	10	b3log/solo
speedment	945	2852	82	33	speedment/speedment
symphony	1694	5083	43	11	b3log/symphony
tablesaw	567	666	40	23	lwhite1/tablesaw
underscore-java	132	535	0	28	javadev/underscore-java
webmagic	4960	1005	97	23	code4craft/webmagic
weixin-java-tools	1794	380	9	21	chanjarster/weixin-java-tools
weixin-popular	714	203	8	33	liyiorg/weixin-popular
xembly	146	319	6	45	yegor256/xembly
YCSB	1684	1110	117	45	brianfrankcooper/YCSB
zipkin	6242	1209	152	69	openzipkin/zipkin

Table A.2: Android projects.

Project	Stars	Commits	Issues	Releases	URL
AboutLibraries	1823	964	0	97	mikepenz/AboutLibraries
android	1303	1380	211	38	cSploit/android
Android-ActionItemBadge	1087	191	0	29	mikepenz/Android-ActionItemBadge
AndroidScreencast	175	107	12	12	xSAVIKx/AndroidScreencast
AndroidPicker	2406	296	37	29	gzu-liyujiang/AndroidPicker
Android-Remote	128	738	41	11	clementine-player/Android-Remote
android-sdk	369	705	4	42	qiniu/android-sdk
AndroidViewAnimations	7915	140	29	14	daimajia/AndroidViewAnimations
andstatus	120	1636	104	113	andstatus/andstatus
AntennaPod	1608	3939	268	77	AntennaPod/AntennaPod
Applozic-Android-SDK	375	1224	55	17	AppLozic/Applozic-Android-SDK
AppUpdate	100	31	0	15	fccaikai/AppUpdate
CircularReveal	2082	88	8	19	ozodrukh/CircularReveal
cwac-pager	101	47	0	9	commonsguy/cwac-pager
Dragger	1102	191	0	12	ppamorim/Dragger
FileDownloader	4585	626	44	53	lingochamp/FileDownloader
glide	16810	1684	412	23	bumptech/glide
Iron	101	216	1	16	FabianTerhorst/Iron
java-telegram-bot-api	202	287	2	24	pengrad/java-telegram-bot-api
LicensesDialog	541	220	1	13	PSDev/LicensesDialog
LollipopShowcase	1776	83	0	6	mikepenz/LollipopShowcase
material-menu	2328	118	3	20	balysv/material-menu
MusicUU	308	59	22	15	Qrilee/MusicUU
Ok2Curl	131	99	2	15	mrmike/Ok2Curl
openxc-android	199	571	0	14	openxc/openxc-android
OkHttpFinal	608	101	13	2	pengjianbo/OkHttpFinal
PanicButton	1107	1046	4	18	PanicInitiative/PanicButton
PictureSelector	1853	995	14	54	LuckSiege/PictureSelector

Continued on next page

Table A.2 (continued from previous page)

Project	Stars	Commits	Issues	Releases	URL
pulsar	184	2521	16	32	apache/incubator-pulsar
rpicheck	100	443	18	15	eidottermihi/rpicheck
RxFlux	324	62	1	2	skimarxall/RxFlux
RxJavaSamples	2672	12	18	2	rengwuxian/ RxJavaSamples
ShowcaseView	4866	496	65	14	amlcurran/ ShowcaseView
signal-cli	365	242	17	18	AsamK/signal-cli
SkyTube	142	378	143	9	ram-on/SkyTube
slidr	1505	71	8	4	r0adkll/Slidr
SlyceMessaging	876	121	20	10	Slyce-Inc/ SlyceMessaging
SMSSync	551	1816	79	44	ushahidi/SMSSync
sugar	2353	483	225	6	chennaione/sugar
ViewAnimator	1625	59	9	3	florent37/ViewAnimator

Appendix B

Studied Library Migrations

Tables B.1 and B.2 list the studied successive versions of the projects along with the two libraries involved in migrations. The tables also show whether the developer removed the dependency after migration or not.

Table B.1: Maven projects.

System	Version Migrating	Version Next	Library Migrated	Alternative	Removed Library
admiral	0.9.2	1.1.0	Jackson-Dataformat-YAML	SnakeYAML	N
allure1	1.4.10	1.4.11	Old-JAXB-Runtime	standard Java (java. nio. charset)	Y
apollo	0.6.3	0.7.0	Plexus-Archiver	Google Guice	N
apollo	0.6.3	0.7.0	UnidalFrameWork	internal implementa-tion	Y
auto	1.4	1.4.1	org. ow2. asm	module removed	Y
bitsquare	0.3.4	0.3.5	Enzo	standard Java (javafx)	Y
brave	2.4.2	3.0.0-alpha-1	Apache-Commons-Lang	AutoValue	N
brave	2.4.2	3.0.0-alpha-1	Guava	standard Java (javafx)	Y

Continued on next page

Table B.1 (continued from previous page)

System	Version Migrating	Version Next	Library Migrated	Alternative	Removed Library
cassandra-lucene-index	2.2.9.0	3.0.5.4	Data-Mapper-For-Jackson	Jackson-Core	N
checkstyle	6.16.1	6.17	Apache-Commons-Lang	extracted files	Y
checkstyle	7.1.1	7.1.2	Guava	standard Java	Y
commafeed	2.2.0	2.3.0	Swagger-Jaxrs	Swagger-Annotations	N
commafeed	2.2.0	2.3.0	Querydsl-JPA-Support	Querydsl-Core-Module	N
crawler4j	4.1	4.3	Lidalia-SLF4J-Extensions	(SLF4J)	N
crawljax	2.2	3.0.0	Apache-Commons-Configuration	internal configuration	Y
crawljax	2.2	3.0.0	Apache-Commons-IO	standard Java (java.io)	Y
cron-utils	5.0.0	5.0.1	Apache-Commons-Lang	extracted files	Y
cron-utils	4.1.3	5.0.0	Guava	standard Java	Y
cron-utils	4.1.3	5.0.0	Joda-Time	standard Java	Y
DCMonitor	0.1.2-rc	0.1.3	Influxdb	Prometheus	N
density-converter	0.9.1	0.9.2	Imgscalr	Thumbnailato	N
disconf	2.6.2	2.6.25	Jackson	knightliao.apollo.JsonUtils	N
disconf	2.6.30	2.6.31	Jersey-Core-Client	Apache-HttpClient	N
disconf	2.6.2	2.6.25	Spring-Framework	Spring-core	N
Discord4J	2.7.0	2.8.0	Gson	Jackson-Core	N
Discord4J	2.4.1	2.4.2	Java-WebSocket	Jetty-Core	N
druid	0.9.2	0.10.0	metamx:java-Util	extracted files	Y
EasyReport	1.0.15	2.0.16	Apache-Log4j-	projectlombok	N
erlyberly	0.2.3	0.3.0	ShichimiFX	javafx	N
fb-android-dagger	1.0.3	1.0.5	Guava	extracted files	Y

Continued on next page

Table B.1 (continued from previous page)

System	Version Migrating	Version Next	Library Migrated	Alternative	Removed Library
ff4j	1.4	1.5	Jackson	Jackson-Core (fasterxml)	N
freedomotic	5.5.1	5.6.0-rc3	standard Java (logging)	SLF4J	N
FXGL	0.1.9	0.2.0	Ehcache	Internal-Cache	Y
galen	1.6.4	2.0.0	Rainbow4J	extracted files	Y
geo	0.7.4	0.7.5	Guava	module segregation	N
GeoIP2-java	2.7.0	2.8.0-rc1	Google-HTTP-Client	Apache-HttpClient	N
GoogleAuth	0.4.2	0.4.3	Javax. ws	Apache-HttpClient	N
GoogleAuth	0.4.1	0.4.2	Guava	standard Java	Y
google-java-format	1.1	1.2	AutoValue	internal implementation	Y
google-java-format	1	1.1	Jcommander	internal implementatio	Y
grakn	0.11.0	0.12.0	Logback-Core	module removed	N
greenmail	1.4.1	1.5.0	Apache-Commons-Lang	extracted files	Y
ha-bridge	2.0.7	3.0.0	Jackson-Core (fasterxml)	Gson	N
hangout	0.1.2	0.1.3	Jinjava	module removed	Y
hawkular-apm	0.1.0.	0.2.0.	Swagger-Annotations	Swagger-Annotations	N
Hive2Hive	1.1.0	1.1.1-alpha1	Jansi	internal implementation	Y
HotSUploader	1.0-rc.1	1.0-rc.2	Google-HTTP-Client	standard Java (java.net)	Y
HotSUploader	2.0.5	2.1.0	Gluon-Ignite-Common	Spring-Framework	N

Continued on next page

Table B.1 (continued from previous page)

System	Version Migrating	Version Next	Library Migrated	Alternative	Removed Library
hprose-java	2.0.30	2.0.31	JavaServlet-Specification	Java-Servlet-API	N
htmlelements	1.16	1.17	Commons-Lang	Apache-Commons-Lang	N
htmlelements	1.14	1.15	Lambdaj	standard Java (java. util)	Y
jacoco	0.6.5	0.7.0	ASM-All	ASM-Debug-All	N
JCloisterZone	2.7	3.0.2	Apache-MINA-Core	standard Java (java. nio. channels)	Y
jmx_exporter	0.4	0.5	JSON. simple	SnakeYAML	N
jparsec	2.3	3.0-rc1	Cglib-Nodep	standard Java (Lambda)	Y
jstorm	0.9.6.1	0.9.6.2	Curator	Apache-Curator	N
jstorm	0.9.6.1	0.9.6.2	Fastjson	JSON. simple	N
jstorm	0.9.6	0.9.6.1	JSON. simple	Fastjson	N
kumuluzee	2.1.0	2.1.0	YamlBeans	SnakeYAML	N
lambada framework	0.0.3	0.0.4	Ini4j	module removed	Y
leshan	0.1.11-M5	0.1.11-M7	Apache-Commons-Codec	extracted files	Y
light-task-scheduler	1.6.6	1.6.7.1	ReflectASM	module removed	Y
light-task-scheduler	1.6.4	1.6.5	Curator	Apache-Curator	N
light-task-scheduler	1.6.4	1.6.5	Apache-Commons-DbUtils	extracted files	Y
lolibox	0.0.3-	0.0.4-	JavaServlet-Specification	module removed	N
lolibox	0.1.0	0.2.0	Argparse4j	module removed the library	Y
lolibox	0.1.0	0.2.0	Javax. ws. rs-Api	Spring-Web	Y
maven-jaxb2-plugin	0.12.2	0.12.3	Maven-Plugin-API	SLF4J	N

Continued on next page

Table B.1 (continued from previous page)

System	Version Migrating	Version Next	Library Migrated	Alternative	Removed Library
metrics	3.0.2	3.1.0	Caliper	Oracle-JMH	N
MongoDB-Plugin	1.0.6.3	1.0.7	SLF4J	Logback-Classic	N
mpush	0.6.1	0.7.0	Gson	Fastjson	N
mvvmFX	p-0.1.3	0.1.4	Javax. inject	Google-Guice-Core-Library	N
mvvmFX	p-0.1.3	m-0.1.4	standard Java (reflect)	TypeTools	N
netty-zmtp	0.2.0	0.2.1	JeroMQ	JeroMQ	N
newts	1.0.0	1.1.0	Dropwizard	Dropwizard	N
objenesis	2.4	2.5	Spring-OSGi-Core	OPS4J-PaxExam	N
PixelController	1.5.1	2.0.0. RC1	Processing-Core	standard Java	Y
RankSys	0.1	0.3	Apache-Commons-Lang	standard Java (java. util)	Y
requests	4.7.0	4.7.2	Apache-Log4j-API	module removed	Y
retrofit	0.6.0-rc3	0.6.0-rc4	Google-Guice-Core	Javax. inject	N
retrofit	1.6.0	2.0.0	RxJava	RxJava	N
retrofit	1.6.0	2.0.0	AppEngine-API	module removed	Y
retrofit	0.6.0-rc6	0.1.0	EasyMock	Mockito	N
retrofit	0.6.0-rc6	0.1.0	Javax. inject	module removed	Y
retrofit	0.6.0-rc6	0.1.0	Apache-HttpClient	module removed	Y
rtree	0.7.4	0.7.5	Guava	Guava-Mini	N
seyren	1.2.1	1.3.0	PagerDuty-API	PagerDuty-Incidents	N
simple-java-mail	3.1.0	4.0.0	Hamcrest-Core	AssertJ-Assertions	N
SlimFast	0.13	0.14	JetS3t	AWS-SDK	N
solo	1.3.0	1.4.0	Appengine-ApiStubs	module removed	Y
solo	2.1.0	2.2.0	Pegdown	Flexmark-Core	N
solo	1.3.0	1.4.0	MarkdownPapers-Core	Pegdown	N

Continued on next page

Table B.1 (continued from previous page)

System	Version Migrating	Version Next	Library Migrated	Alternative	Removed Library
speedment	2.0.0-EA	2.0.0EA2	Apache-Log4j	internal implementation	Y
speedment	2.3.7	3.0.0-EA	Gson	internal implementation	Y
speedment	2.2.3	2.3.0	Apache-Groovy	Internal-Json	Y
speedment	2.3.7	3.0.0-EA	ControlsFX	JavaFX-Maven-Plugin	Y
symphony	2.1.0	2.2.0	Pegdown	Flexmark-Core	N
symphony	1.3.0	1.4.0	MarkdownPapers-Core	Pegdown	N
symphony	1.3.0	1.4.0	Jetty-Websocket	WebSocket-Server-API	N
tablesaw	v0.7.1	v0.7.1.1	Opencsv	Opencsv	N
underscore-java	v1.17	v1.18	Guava	standard Java	Y
webmagic	0.4.2	0.4.3	Apache-Log4j-API	SLF4J	N
webmagic	0.5.3	0.6.0	Guava	standard Java	Y
weixin-popular	2.0.1-R	2.2.0-R	Jackson-Databind	Fastjson	N
xembly	0.16.2	0.16.3	Rexsl-Test	Jcabi-Matchers	N
YCSB	0.6.0	0.7.0RC1	yahoo-gemfire-binding	com. gemstone	N
zipkin	1.6.0	1.7.0	Okio	extracted files	Y
zipkin	1.6.0	1.7.0	Moshi	Gson	N
zipkin	1.20.0	1.22.0	elasticsearch	module removed	Y

Table B.2: Android projects.

System	Version Migrating	Version Next	Library Migrated	Alternative	Removed Library
AboutLibraries	5.5.9	5.6.0	cardsui-for-android	removed	Y
android	1.5.3	1.5.4	ActionBarSherlock	Android-support	Y
Android-ActionItemBadge	1.2.0	2.5.5	Android-Iconify	Android-Iconics-Library	N
AndroidPicker	1.4.5	1.4.6	MaterialViewPager	Android-support	Y
Android-Remote	10.1	11	ShowcaseView-Library	Android-support	Y
AndroidScreencast	0.0.9	0.0.10	Spring-Framework	Dagger	N
AndroidScreencast	0.0.9	0.0.10	Apache-Log4j	SLF4J	N
android-sdk	7.1.3	7.2.0	OkHttp	OkHttp	N
Android ViewAnimations	1.1.3	2	Nine-Old-Androids	Android-support	Y
andstatus	24.03-r	25.04-r	HttpComponents-Client-For-Android	HttpClient-Android-Library	N
AntennaPod	1.4.0.12	1.4.1.4	Apache-Commons-Lang	Android-support	Y
Applozic-Android-SDK	1.61	4.82	Apache-HttpCore	standard Java (java.net)	Y
AppUpdate	2.0.6	2.0.7	OkHttp	standard Java (java.net)	Y
CircularReveal	1.3.0	1.3.1	Nine-Old-Androids	Android-support (animation)	Y
cwac-pager	0.2.5	0.2.6	ActionBarSherlock	Android-support	Y
Dragger	1.0.5.1	1.0.5.2	Nine-Old-Androids	Android-support	Y
FileDownloader	1.3.0	1.3.9	OkHttp	standard Java (java.net)	Y
glide	3.3.1	3.4.0	Hamcrest-Library	Truth-Core	N
Iron	0.6.5	0.7.2	Kryo-Serializers	Kryo	N
java-telegram-bot-api	1.3.3	2.0.0	Retrofit	OkHttp	N
LicensesDialog	1.2.0	1.3.0	Android-Native (Log)	SLF4J	N

Continued on next page

Table B.2 (continued from previous page)

System	Version Migrating	Version Next	Library Migrated	Alternative	Removed Library
LicensesDialog	1.2.0	1.3.0	Simple-XML	XML-Pull-Parsing-API	N
LicensesDialog	1.4.0	1.5.0	SLF4J	Removed	Y
LicensesDialog	1.2.0	1.3.0	Apache-Commons-IO	standard Java (java.io)	Y
LollipopShowcase	2.0.2	2.2.0	Android-Iconify	Android-Iconics-Library	N
LollipopShowcase	2.0.2	2.2.0	AboutLibraries	AboutLibrarie Library	N
LollipopShowcase	2.0.2	2.2.0	Crouton	Snackbar	N
LollipopShowcase	2.2.0	2.3.0	Snackbar	Android-support	Y
material-menu	1.5.5	2.0.0	ActionBarSherlock	Android-support	Y
material-menu	1.5.5	2.0.0	Nine-Old-Androids	Android-support	Y
material-menu	1.5.5	2.0.0	AppCompatActivity	removed	Y
MusicUU	1.1.8	1.2.1	OkHttp	NoHttp	N
MusicUU	1.1.8	1.2.1	Retrofit	NoHttp	N
MusicUU	1.1.8	1.2.1	Zhy. OkHttpUtils	OkHttpUtils	N
Ok2Curl	0.0.4	0.1.0	OkHttp	OkHttp	N
OkHttpFinal	1.2.2	2.0.6	OkHttp	OkHttp	N
openxc-android	5.3.2	6.0.0-a1	Jackson	Gson	N
PanicButton	1.2.0	1.2.1	RoboGuice	Android-support	N
PictureSelector	1.4.5	1.4.6	OkHttp	removed	Y
PictureSelector	1.5.0	1.5.2	EventBus	self-written	Y
pulsar	1.15.7	1.16	AsynchronousHttp-Clientning	Asynchronous Http-Client	N
pulsar	1.15.7	1.16	JSON-In-Java	Gson	N
rpicheck	1.6.5a	1.7.1	Android-PullToRefresh	Android-Support	Y
rpicheck	1.6.5a	1.7.1	ActionBarSherlock	Android-Support	Y
RxFlux	0.2.1	0.4.1	OkHttp	OkHttp	N
RxFlux	0.2.1	0.4.1	Retrofit	Retrofit	N
RxJavaSamples	1.0	1.01	OkHttp	OkHttp	N
RxJavaSamples	1.0	1.01	Retrofit	Retrofit	N
ShowcaseView	4.0	5.0.0	Nine-Old-Androids	Android-support	Y

Continued on next page

Table B.2 (continued from previous page)

System	Version Migrating	Version Next	Library Migrated	Alternative	Removed Library
signal-cli	0.0.4	0.0.5	JSON-In-Java	Jackson-Core	N
SkyTube	2.3	2.4	Picasso	Glide	N
Slidr	2.0.3	2.0.4	Picasso	Glide	N
SlyceMessaging	1.0.1	1.0.2	Picasso	Glide	N
SMSSync	2.5.1	2.6	ActionBarSherlock	Android-Support	Y
sugar	1.4	1.5	Guava:-Google-Core-Libraries-For-Java	Java. net(Map)	Y
ViewAnimator	1.0.3	1.0.4	Nine-Old-Androids	Android-support	Y

Appendix C

Systems that Broke Transitively Dependent Clients

Table C.1 lists the projects that probably broke client code after library migration. The table shows the severity of data type problems (DT) and method problems (M) as found by JAPICC for each project. Details regarding the severity metric used by JAPICC can be found online.[85] The table also shows whether migration caused issues with parameters (P), return type (RT), and superinterfaces (SI), as well as the compatibility score (C) between versions.

Table C.1: Breaking APIs.

System	DT			M			P	RT	SI	C
	High	Med	Low	High	Med	Low				
admiral	Y	N	N	N	N	N	Y	N	N	0.201
allure1	Y	N	N	N	N	N	N	N	N	0.863
Android Screencast	Y	N	N	N	N	N	Y	N	N	0.343
apollo	Y	N	N	N	N	N	N	N	Y	0.976
apollo	N	Y	N	N	N	N	N	N	Y	0.976
brave	Y	N	N	Y	N	N	Y	Y	Y	0.875
brave	Y	N	N	N	N	N	N	N	N	0.875
cassandra-lucene-index	N	N	Y	Y	N	N	Y	N	N	0.695
checkstyle	N	N	N	Y	N	N	N	Y	N	1

Continued on next page

[85] https://lvc.github.io/japi-compliance-checker/#Maintainers

Table C.1 (continued from previous page)

System	DT			M			P	RT	SI	C
	High	Med	Low	High	Med	Low				
crawljax	Y	N	N	N	N	N	Y	N	N	0.017
disconf	Y	N	N	N	N	N	Y	N	N	0.962
geo	N	N	N	N	N	N	N	N	N	1
ha-bridge	N	N	Y	Y	Y	Y	Y	N	N	0.806
hangout	Y	N	N	N	N	N	N	N	N	0.833
Hive2Hive	Y	N	N	N	N	N	N	N	N	0.74
HotS Uploader	Y	N	N	N	N	N	N	N	N	0.737
JCloister Zone	Y	N	N	N	N	N	Y	N	N	0
jmx exporter	Y	N	N	N	N	Y	N	N	N	0.985
jparsec	Y	N	N	N	N	N	Y	N	N	0
jstorm	Y	N	N	Y	N	N	Y	Y	N	0.97
jstorm	Y	N	N	Y	N	N	Y	Y	N	0.978
jstorm	N	N	Y	Y	N	N	N	Y	N	0.978
Licenses Dialog	Y	N	N	N	N	N	N	N	Y	0.922
light-task-scheduler	N	N	Y	Y	N	N	Y	Y	N	0.911
lolibox	Y	N	N	N	N	N	N	N	N	0.279
metrics	N	Y	N	N	N	Y	N	Y	Y	0.882
mpush	Y	N	N	N	N	N	Y	N	N	0.745
newts	N	N	Y	N	N	N	N	N	Y	0.985
openxc-android	Y	N	N	N	N	N	N	N	N	0.904
pulsar	Y	N	N	Y	Y	N	Y	Y	N	0.987
retrofit	Y	N	N	N	N	N	N	N	N	0.009
speedment	Y	N	N	N	N	N	N	N	N	0.001
symphony	Y	N	N	N	N	N	Y	Y	N	0.849
webmagic	Y	N	N	N	N	N	N	N	N	0.955
weixin-popular	Y	N	N	N	N	N	Y	Y	N	0.927
YCSB	Y	N	N	N	N	N	N	N	N	0
zipkin	Y	N	N	N	N	N	N	N	N	0.998

Appendix D

Systems Used for Evaluation

Table D.1 lists the projects that were used for evaluation of EDW, along with their successive versions and involved libraries. The table also shows the total classes (C), methods (M), fields (F), and lines of code (LOC) for each evaluated project.

Table D.1: Systems for evaluation.

System	First version	New version	Migrated library	Alternative library	C	M	F	LOC
Graylog2-server	0.13.0-rc.1	0.20.0	JSON. simple	FasterXML	226	1089	651	18389
Esper	5.4.0	6.0.0	Apache Commons Logging	SLF4J	5086	39652	13726	539559
Choco-solver	3.3.1	3.3.2	SLF4J	Removed	797	6673	2478	121959
Webprotege	2.6.0	3.0.0	Google Guice Core Library	Dagger	1987	11216	4016	110105
Geowave	0.9.3	0.9.4	Apache Log4j	SLF4J	1422	11533	4843	192608
Hapi-fhir	1.6	2	JSR 374 (JSON Process-ing)	Gson	5428	4079	1332	59193
Commands	commit	0.1 (master)	Guava	Java native	144	950	317	12396

Continued on next page

Table D.1 (continued from previous page)

System	First version	New version	Migrated library	Alternative library	C	M	F	LOC
Azure-cosmosdb-java	1.0.2	2.0.0	JSON in Java	Jackson Core	274	2277	1274	35973
Incubator-dubbo	2.61	2.6.2	EasyMock	Mockito	546	4462	1371	54919
Ta4j-origins	0.8	1	Joda Time	Java native	298	989	489	20590

Appendix E

Individual Project Results for Evaluation

Tables E.4, E.1, E.2, and E.3 list the TP, FP, FN, precision, recall, F-score, and MSI (manual search interpretation) at class-granularity for EDW and JRipples search strategies, respectively. Tables E.6, E.5, E.8, and E.8 list the TP, FP, FN, precision, recall, F-score, and PTR (points to remember) at method- and field-granularity, for EDW & JRipples search strategy 1, respectively.

Table E.1: Individual project results for class-granularity, JRipples search strategy 1.

Project	TP	FP	FN	P	R	F_1	MSI
1	10	2	0	0.83	1	0.91	24
2	636	0	4	1	0.99	0.99	1405
3	31	0	0	1	1	1	78
4	54	18	0	0.75	1	0.86	70
5	186	0	0	1	1	1	592
6	2	2	0	0.5	1	0.67	15
7	23	0	0	1	1	1	38
8	43	0	0	1	1	1	48
9	23	4	0	0.85	1	0.92	31
10	14	0	0	1	1	1	25
sum	1022	26	4	—	—	—	2326
mean	102.2	2.6	0.4	0.89	0.99	0.93	232.6

Table E.2: Individual project results for class-granularity, JRipples search strategy 2.

Project	TP	FP	FN	P	R	F_1	MSI
1	10	2	0	0.83	1	0.91	24
2	640	0	0	1	1	1	2949
3	31	0	0	1	1	1	47
4	54	18	0	0.75	1	0.85	88
5	186	45	0	0.81	1	0.89	651
6	2	2	0	0.5	1	0.67	8
7	23	0	0	1	1	1	42
8	43	6	0	0.87	1	0.93	48
9	23	4	0	0.85	1	0.92	31
10	14	0	0	1	1	1	25
sum	1026	77	0	—	—	—	3910
mean	102.6	7.7	0	0.86	1	0.91	391

Table E.3: Individual project results for class-granularity, JRipples search strategy 3.

Project	TP	FP	FN	P	R	F_1	MSI
1	10	2	0	0.83	1	0.91	21
2	640	26	0	0.96	1	0.98	2976
3	31	0	0	1	1	1	47
4	54	18	0	0.75	1	0.85	88
5	186	337	0	0.36	1	0.52	1621
6	2	0	0	1	1	1	130
7	23	0	0	1	1	1	42
8	43	6	0	0.87	1	0.93	51
9	23	4	0	0.85	1	0.92	27
10	14	0	0	1	1	1	23
sum	1026	393	0	—	—	—	2026
mean	102.6	39.3	0	0.86	1	0.91	502.6

Table E.4: Individual project results for class-granularity, EDW.

Project	TP	FP	FN	P	R	F_1	MSI
1	10	0	0	1	1	1	0
2	636	0	4	1	0.99	0.99	0
3	31	0	0	1	1	1	0
4	54	0	0	1	1	1	0
5	186	0	0	1	1	1	0
6	2	0	0	1	1	1	0
7	23	0	0	1	1	1	0
8	43	0	0	1	1	1	0
9	23	0	0	1	1	1	0
10	14	0	0	1	1	1	0
sum	1022	0	4	—	—	—	0
mean	102.2	0	0.4	1	0.99	0.99	0

Table E.5: Individual project results for method-granularity, JRipples search strategy 1.

Project	TP	FP	FN	P	R	F_1	PTR
1	12	2	0	0.85	1	0.92	50
2	8	0	0	1	1	1	18
3	24	0	27	1	0.47	0.64	94
4	23	0	0	1	1	1	258
5	6	0	11	1	0.35	0.52	12
6	37	18	0	0.67	1	0.80	128
7	30	5	0	0.86	1	0.92	88
8	62	27	9	0.69	0.87	0.75	161
9	69	18	0	0.79	1	0.88	594
10	53	28	0	0.65	1	0.79	147
sum	324	98	47	—	—	—	1550
mean	32.4	9.8	4.7	0.85	0.86	0.82	155

Table E.6: Individual project results for method-granularity, EDW.

Project	TP	FP	FN	P	R	F_1	PTR
1	12	0	0	1	1	1	0
2	8	0	0	1	1	1	0
3	51	0	31	1	0.43	0.60	0
4	23	0	0	1	1	1	0
5	17	0	11	1	0.35	0.52	0
6	37	0	0	1	1	1	0
7	30	0	0	1	1	1	0
8	71	0	9	1	0.87	0.93	0
9	69	0	0	1	1	1	0
10	53	0	0	1	1	1	0
sum	324	371	0	—	—	—	0
mean	32.4	37.1	0	1	0.99	0.86	0

Table E.7: Individual project results for field-granularity, JRipples search strategy 1.

Project	TP	FP	FN	P	R	F_1	PTR
2	636	31	4	0.95	0.99	0.97	2813
3	16	0	1	1	0.94	0.97	17
5	202	0	0	1	1	1	202
7	15	7	0	0.68	1	0.81	15
9	18	1	0	0.94	0.97	1	18
10	7	0	0	1	1	1	7
sum	898	39	5	—	—	—	3072
mean	149.67	6.5	1.67	0.93	0.99	0.95	512

Table E.8: Individual project results for field-granularity, EDW.

Project	TP	FP	FN	P	R	F_1	PTR
2	636	0	4	1	0.99	0.99	0
3	16	0	0	1	1	1	0
5	202	0	0	1	1	1	0
7	15	0	0	1	1	1	0
9	18	0	0	1	1	1	0
10	7	0	0	1	1	1	0
sum	897	0	4	—	—	—	0
mean	149.5	0	0.67	1	0.99	0.99	0

Appendix F

Experimental Materials

This appendix provides all the materials provided to participants during the experimental trials described in Section 7.5.

The detailed description and instructions for the tasks are provided in F.1. The pre-study, post-task, and final questionnaire are provided in Sections F.2, F.3, and F.4, respectively.

F.1 Task Descriptions and Instructions

Task 1

Task 1

You received a change request to migrate from third party library Guava in JUNG API module. The library is being used for its data structures and utilities. The task is to locate the initial impact set (classes and members) of Guava to start the change.

Task Description: Find all the classes and members directly referencing the types and fields used from Guava in the JUNG API module.

Steps:

1. Setup Tool.

2. Locate and Mark the classes directly referencing Guava.

3. Locate and Mark the affected members (Fields \& Methods) in discovered classes, that are directly referencing Guava.

4. Finalize and announce the list of classes and their members directly referencing Guava.

5. Fill Task questionnaire.

Task 2

Task 2

You received a change request to migrate from third party library Guava in JUNG IO module. The library is being used for its data structures and utilities. The task is to locate the initial impact set (classes and members) of Guava to start the change.

Task Description: Find all the classes and members directly referencing the types and fields used from Guava in the JUNG IO module.

Steps:

1. Setup Tool.

2. Locate and Mark the classes directly referencing Guava.

3. Locate and Mark the affected members (Fields \& Methods) in discovered classes, that are directly referencing Guava.

4. Finalize and announce the list of classes and their members directly referencing Guava.

5. Fill Task questionnaire.

F.2 Pre-Study Questionnaire

Pre-Study Questionnaire

Participant ID:

Current occupation?

Graduate Student

Industrial Developer

If you are a student, have you ever been employed as a developer in industry?

Yes

No

If yes, for how long?

Years of development experience (note: basic, pascal et al. do not count)?

Years developing with Java?

How familiar are you with Java?

1 - Not familiar, have never used it

2 - Have used it once or twice

3 - I have used it for a short periods in the past

4 - I often use it, or have actively used it in the past

5 - It is my everyday development language

Do you regularly use an IDE while developing software?

Yes

No

If yes, which IDE do you primarily use?

Have you ever used the Eclipse IDE?

How familiar are you with Eclipse?

1 - Not familiar, have never used it

2 - Have used it once or twice

3 - I have used it for a short period in the past

4 - I often use it, or have actively used it in the past

5 - It is my everyday development environment

Are you familiar with the term Third Party Libraries?

1 - Not familiar, have never heard about it

2 - Have heard about it once or twice

3 - I have used it for a short periods in the past

4 - I often use it, or have actively used it in the past

5 - It is my everyday development

Have you implemented change requests before?

Yes

No

If yes, for how often?

If yes, was it targeting similar components or different components for each change?

Mention up to three reasons that would stop you from implementing a change request

1)

2)

3)

Have you ever terminated a change request after starting it?

Yes

No

If yes, what are the reasons?

How do you evaluate the feasibility of applying code changes?

Additional Comments:

F.3 Post-Task Questionnaire

Post-Task Questionnaire

Participant ID:

Task: **Approach:** JRIPPLE / EDW

How did you find the task?

1) Difficult
2) Moderately Difficult
3) Okay
4) Moderately Easy
5) Easy

How well do you think you did the task?

Did the task take more time or less than you anticipated at the beginning?

More
Less

What three things (difficulties) made the task hard?

1)
2)
3)

What three things made the task easy?

1)
2)
3)

Did the approach used, help you or hinder you?

What did you dislike about the used approach?

Do you have any suggestions for improving the used approach?

Do you have any other comments?

F.4 Final Questionnaire

Final Questionnaire

Participant ID:

Compare and contrast between the kinds of problems you faced when using the JRIPPLE vs. those faced when using our tool (EDW).

Was the task easier or harder to perform with our tool support (EDW)? Why?

Which approach do you think would be easier to estimate the required changes?

Which approach do you think you were more confident with when estimating the required changes?

- ☐ Strongly preferred JRipples approach
- ☐ Somewhat preferred JRipples approach
- ☐ Neither was better
- ☐ Somewhat preferred the tool support (EDW)
- ☐ Strongly preferred the tool support (EDW)

Why?

Using the tool (EDW), I was more likely to succeed in estimating the changes needed.

1) Strongly disagree
2) Disagree
3) Neither agree nor disagree
4) Agree
5) Strongly agree

I feel I could attempt larger changes and modification tasks using the novel tool support (EDW) than if I were to perform them without it.

1) Strongly disagree
2) Disagree
3) Neither agree nor disagree
4) Agree
5) Strongly agree

I would use the novel tool (EDW) with its existing functionality again.

1) Strongly disagree
2) Disagree
3) Neither agree nor disagree
4) Agree
5) Strongly agree

Do you have any other comments?

Appendix G

Detailed Experimental Results

Tables G.1, G.2, and G.3 list the confusion matrix, precision, recall, and F-sore achieved by JRipples for each participant in Task 1 for class-, method-, and field-granularity, respectively.

Tables G.4, G.5, and G.6 list the confusion matrix, precision, recall, and F-score achieved by JRipples for each participant in Task 2 for class-, method-, and field-granularity, respectively.

Note that EDW achieved perfect precision and recall in all cases.

Table G.1: Effectiveness: JRipples on Task 1 at class-granularity.

Participant	TP	FP	FN	P	R	F
P1	7	0	0	1	1	1
P2	7	0	0	1	1	1
P3	2	0	5	1	0.29	0.44
P4	7	0	0	1	1	1
P5	7	0	0	1	1	1
P6	2	0	5	1	0.29	0.44
P7	7	0	0	1	1	1
P8	7	0	0	1	1	1

Table G.2: Effectiveness: JRipples on Task 1 at method-granularity.

Participant	TP	FP	FN	P	R	F
P1	4	0	0	1	1	1
P2	4	0	0	1	1	1
P3	3	0	1	1	0.75	0.86
P4	4	2	0	0.67	1	0.8
P5	4	1	1	0.8	0.8	0.8
P6	2	1	2	0.67	0.5	0.57
P7	4	11	0	0.27	1	0.42
P8	3	5	1	0.38	0.75	0.5

Table G.3: Effectiveness: JRipples on Task 1 at field-granularity.

Participant	TP	FP	FN	P	R	F
P1	9	0	0	1	1	1
P2	9	0	0	1	1	1
P3	0	0	9	0	0	0
P4	9	0	0	1	1	1
P5	7	0	2	1	0.78	0.88
P6	0	0	9	0	0	0
P7	9	0	0	1	1	1
P8	7	1	2	0.73	0.78	1

Table G.4: Effectiveness: JRipples on Task 2 at class-granularity.

Participant	TP	FP	FN	P	R	F
P1	8	2	7	0.89	0.53	0.67
P2	15	0	0	1	1	1
P3	11	0	4	1	0.73	0.85
P4	15	0	0	1	1	1
P5	8	0	7	1	0.53	0.70
P6	7	1	8	0.88	0.47	0.61
P7	15	0	0	1	1	1
P8	15	0	0	1	1	1

Table G.5: Effectiveness: JRipples on Task 2 at method-granularity.

Participant	TP	FP	FN	P	R	F
P1	25	0	26	1	0.49	0.65
P2	40	0	11	1	0.78	0.88
P3	25	0	26	1	0.49	0.66
P4	26	4	25	0.86	0.50	0.64
P5	14	0	37	1	0.27	0.43
P6	13	6	38	0.68	0.25	0.37
P7	40	6	11	0.86	0.78	0.82
P8	30	1	21	0.96	0.58	0.73

Table G.6: Effectiveness: JRipples on Task 2 at field-granularity.

Participant	TP	FP	FN	P	R	F
P1	23	0	13	1	0.64	0.78
P2	28	0	8	1	0.78	0.88
P3	16	1	20	0.94	0.44	0.60
P4	35	0	1	1	0.97	0.99
P5	2	0	34	1	0.06	0.11
P6	13	0	23	1	0.36	0.53
P7	34	0	2	1	0.94	0.97
P8	36	0	0	1	1	1