fine-GRAPE: fine-Grained APi usage Extractor An Approach and Dataset to Investigate API Usage

Version of November 16, 2015

Anand Ashok Sawant

fine-GRAPE: fine-Grained APi usage Extractor An Approach and Dataset to Investigate API Usage

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Anand Ashok Sawant
born in Mumbai, India

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

©2015 Anand Ashok Sawant. All rights reserved.

fine-GRAPE: fine-Grained APi usage Extractor An Approach and Dataset to Investigate API Usage

Author:	Anand Ashok Sawant
Student id:	4048830
Email:	`A.A.Sawant@student.tudelft.nl`

Abstract

An Application Programming Interface (API) provides a specific set of functionalities to a developer, with the aim of enabling reuse. APIs have been investigated from different angles such as popularity usage and evolution, to get a better understanding of their various characteristics. For such studies software repositories are mined for API usage examples. However, the mining algorithms used for such purposes do not take type information into account, thus making the results imprecise.

In this thesis, we aim to rectify this by introducing fine-GRAPE, an approach that produces fine-grained API usage information by taking advantage of type information while mining API method invocations and annotations. fine-GRAPE establishes a connection between a method invocation and the class of the API to which the method belongs. By means of fine-GRAPE, we investigate API usages from Java projects hosted on GitHub. We select five of the most popular APIs across GitHub Java projects and collect historical API usage information by mining both the release history of these APIs and the code history of every project that uses them.

We use the resulting dataset to perform four separate analyses. The first measures the lag time of each client by leveraging the version information that has been collected. We see that in most cases clients do not upgrade the version of the API that they are using to the latest version. The consequence of this is that the lag time that each client displays is quite high. The second study investigates the percentage of API features that are used by using the type information in the dataset. The results of this study show that a very small percentage of an API is actually used by clients in the real world. Our third study aims to show the relation between popular features and software quality. Finally, the fourth study analyzes the reaction of clients to the deprecation of API artifacts. Our deprecation study shows that most clients do not really react to deprecated entities.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Bacchelli, Faculty EEMCS, TU Delft
Committee Member:	Dr. A.Orso, ARKTOS, Georgia Tech
Committee Member:	Dr. R.Robbes, Faculty PLEAID, UC Chile
Committee Member:	Dr. A.Bozzon, Faculty EEMCS, TU Delft

Preface

This thesis represents my work from the period of September 2014 to November 2015 during the second year of my Masters program at the Delft University of Technology. In my thesis I present a technique that I developed which aids in the collection of type-checked API usages from open source code hosting platforms called fine-GRAPE. Based on this approach I construct a large dataset that contains usage data for five mainstream Java APIs. I perform four studies based on this dataset, and each of the studies conducted are presented in a chapter in this thesis and is based on a research paper that is either published or under submission. Most of the work for this thesis was done in the Netherlands at the Delft University of Technology under the supervision of Dr. Alberto Bacchelli. Some work was done in Atlanta at the Georgia Institute of Technology under the supervision of Dr. Alessandro Orso. Also, a special thanks to Dr. Romain Robbes for his help and guidance for the work on the nature of deprecation that is one of the studies done on the dataset.

The creation of this thesis was a wonderful and rewarding experience. During its course I got the opportunity to move the United States for a period of five months. I also was lucky enough to get to go my first ever Software Engineering conference during my masters. I would like to thank my family and my friends for being there for me whenever I needed help and for putting up with my shenanigans.

Anand Ashok Sawant
Delft, the Netherlands
November 16, 2015

Contents

List of Figures

Chapter 1

Introduction

An Application Programming Interface (API) is a set of functionalities provided by a third-party component (*e.g.*, library and framework) that is made available to software developers. APIs are extremely popular as they promote reuse of existing software systems [47].

The research community has used API usage data for various purposes such as measuring of popularity trends [66], charting API evolution [27], and API usage recommendation systems [61].

For example, Xie *et al.* have developed a tool called MAPO wherein they have attempted to mine API usage for the purpose of providing developers API usage patterns [91]. Based on a developers' need MAPO recommends various code snippets mined from other open source projects. This is one of the first systems wherein API usage recommendation leveraged open source projects to provide code samples. Another example is the work by L´ammel *et al.* wherein they mined data from Sourceforge and performed an API usage analysis of Java clients. Based on the data that they collected they present statistics on the percentage of an API that is used by clients.

One of the major drawbacks of the current approaches that investigate APIs is that they heavily rely on API usage information (for example to derive popularity, evolution, and usage patterns) that is approximate. In fact, the state of the art considers as "usage" information what can be gathered from file imports (*e.g.*, `import` in Java) and the occurrence of method names in files.

This data is an approximation of the real situation as there is no type checking to verify that a method invocation truly does belong to the API in question and that the imported libraries are used. Moreover, previous work was based on small sample sizes in terms of number of projects analyzed. This could result in an inaccurate representation of the phenomena under study.

With the current work, we try to overcome the aforementioned issues by devising fine-GRAPE (fine-GRained APi usage Extractor), an approach able to extract type-checked API method invocation information from Java programs and we use it to collect detailed historical information on five APIs and how their public methods are used over the course of their entire lifetime by 20,263 client projects.

In particular, we collect data from the open source software (OSS) repositories on GitHub. GitHub in recent years has become the most popular platform for OSS developers, as it offers distributed version control, a pull-based development model, and social features [11]. We consider Java projects hosted on GitHub that offer APIs and quantify their popularity among other projects hosted on the same platform. We select 5 representative projects (from now on, we call them only *API*s to avoid confusion with client projects) and analyze their entire history to collect information on their usage. We get fine-grained information about method calls using a custom type resolution that does not require to compile the projects.

The result is an extensive dataset for research on API usage. It is our hope that our data collection approach and dataset not only will trigger further research based on finer-grained and vast information, but also make it easier to replicate studies and share analyses.

Based on the dataset that we have created, we perform four separate analyses namely: an investigation into the upgrade behavior of clients of an API, a measurement of the percentage of an API that is being used, an investigation into relation between software quality and popular parts of APIs and finally a study into the effect of deprecation of API artifacts on the clients of the API.

Structure of this thesis. The background on API usage datasets is presented in chapter 2. Chapter 3 presents the approach that has been applied to mine this data. For the ease of future users of this dataset an overview of the dataset and some introductory statistics of it can be found in the same chapter. Chapter 4 presents the first three applications on this dataset. In chapter 5 we discuss the results from our study on the reaction to deprecated API artifacts. Finally, in chapter 6 we summarize our contributions, list some future work and conclude this thesis.

Chapter 2

Background and Related Work

2.1 Background

Software Engineering is the discipline wherein a software engineer "writes a software component that will be combined with components written by other software engineers to build a system" [35]. To facilitate easy integration, components have to well documented and should be easy to reuse. This is where APIs come in, as they are essentially components written by a third party that perform a specific function.

APIs expose their functionality in multiple ways. The most standard manner to access an API artifact is to make a method invocation on one of its classes. These method invocations can be static as in the case of the Apache Commons IO API or they could be on an instantiated class as in the case of Google Guava. Another way APIs for APIs to expose their functionality is by way of annotations. In Java, annotations are primarily used to insert metadata in a Java class. Annotations provided by APIs are a simple and lightweight manner of adding functionality to a client class. API classes can also be extended to add new functionality or replace existing functionality. Other ways to access and API can be through REST or SOAP based interfaces, however these are not the focus of our study.

Studying usage of APIs is not new. The earliest papers in this field were published starting 1999. In the early days there was an absence of open source platforms available to find API client code. As an alternative, research was conducted by mining of programming language library usages in large open source projects such as Linux or Eclipse. Despite this limitation, a number of analyses was done on the usage of the APIs. The first technique used by Michail [65] employed data mining of association rules regarding APIs. The next improvement that was made while mining APIs was to use search engines such as Google and Yahoo to find code that matches a certain structure. This technique is called structural mining. Around 2006, researchers started targeting the open source projects hosted by the Apache foundation. The techniques employed to mine data from these projects were either pattern matching, frequent itemset selection and import statement matching. Even though Apache provides a rich source of client code, the amount of data is not that large. In the modern day the rise of platforms such as Sourceforge and GitHub has provided to be a boon to researchers as they host many millions of lines of client code, a number that eclipses that of Apache. This lead to most researchers migrating to these project around the year 2008. The mining techniques generally employed here are: frequent itemset mining, AST parsing, building of code and pattern matching.

There are various end goals when it comes to mining of API usage. For each of these end goals there is a specific *granularity* of API usage that has to be collected. By granularity we consider the amount of information on the usage of the API artifact that is collected. The finest level of granularity is the *method* level and the *annotation* level usage of APIs. At this level all the method invocations on the various classes of the API and the usage of API annotations are collected. The next level is the API *class level* granularity. Here a connection is made between the imports of API classes as defined in the import statements of a class file and the API classes that are in use. The coarsest level of granularity is the *API usage*. The usage generally consists of API boiler plate code such as class instantiation and the invocation of a couple of methods that help setup a certain feature of an API. Collection of API usage can be performed on or more of these granularities based on the needs of a researcher.

Collection of API usage samples is not in itself a goal. There are generally more complex reasons behind the collection of usage. One research avenue in this area is to recommend API usage samples based on the past usage of the API; For example, there are multiple search engines that can take the specification of desired functionality and suggest API usage patterns based on that. API usage patterns can also be used in the field of bug detection. For instance, if a developer does not follow a typical usage pattern of an API, there is a chance that the developer is doing something wrong and a warning can be issued. One of the main reasons to mine this kind of data is to inform an API developer as to how their API is being used and as what parts of the API are popular and why that is the case.

Currently, a large number of studies try employing various techniques to mine APIs. These techniques can mine the usage data at different granularities. The granularity of the data to be collected is generally dictated by the end goal. In the following sections we take a look at the various techniques employed and the end goals that are achieved to see as to how and why API usages are mined.

2.2 Existing API mining techniques

Previous work mined API usage samples, for example in the context of code completion, code recommendation, and bug finding. We see which techniques have been applied in the past.

One of the more popular API usage mining techniques is that used by tools such as MAPO [91] by Xie *et al.*, S6 [75] by Steve Reiss and SNIFF [21] by Chatterjee *et al.*. These tools all find code samples by either querying open source repositories such as Sourceforge or by mining code search engines such as Codase [3]. They then run an AST parser over the mined code samples to extract usage patterns. The upsides of this technique is that it is fast and can be implemented in multiple languages. However, one major drawback is that the ASTs mined from these code samples are type-resolved, thus there is a chance that the patterns mined are not accurate. Tools such as S6 have tried to overcome this by creating a context aware AST parser that is able to couple type information to the various AST nodes.

Researchers have developed tools such as Dynamine [59], JADET [88], Alattin [84] and PR-Miner [55], All of which rely on the same mining technique *i.e.*, frequent itemset mining [9]. The idea behind this technique is that statements that occur frequently together can be considered to be a usage pattern. This technique just as the previous one can result in a high number of false positives, due to the lack of presence of type information for each of the statements being parsed.

The earliest technique that was employed in mining API usage was used by the tool CodeWeb [65] that was developed by Amir Michail. More recently it has been employed in the tool Sourcerer [10] as well. This technique employs a data mining technique that is called generalized association rule mining. An association rule is of the form $(\bigwedge_{x \epsilon X} x) \Rightarrow (\bigwedge_{y \epsilon Y} y)$. This implies that for an event x that takes place, then an event y will also take place with a certain confidence interval. The generalized association rule takes not just this into account but also takes a node's descendants into account as well. These descendants represent specializations of that node. This allows this technique to take class hierarchies into account while mining reuse patterns.

One of the most robust techniques developed is based on bytecode analysis. This has been employed by Bruch *et al.* [18] for the purpose of building their code completion tool. The idea behind this technique is that built class files of Java projects contain all the type based information for all the method invocations made by that java file. This information can be recovered by parsing the class files. To obtain these class files it is necessary to build every client project of an API, and this can prove to be troublesome as not all clients build due to various failures and there are often missing unresolved dependencies.

Another popularly used technique is called pattern matching and has been employed most successfully by Milvea *et al.* in their tool AKTARI [66]. First, to correctly identify the API that is being used in a Java class, the `import` statements are first parsed. Based on the API that is being used, an attempt is made to match the names of all method invocations that are used to the names of the methods in the API that is being referenced in that file. Based on the connection that is made on the names, a usage pattern is distilled. This technique can at times be imprecise due to the fact that method names are not always unique and could relate to different APIs.

Recently, Moreno *et al.* [68] presented a technique to mine API usages using type resolved ASTs. To acquire the type-resolved ASTs their technique has to build all the client projects and then use an AST parser on the Java code to retrieve the type-resolutions. As previously mentioned in the context of bytecode analysis, this could result in the loss of data, as some client projects may not build.

2.3 Use cases for API Usage datasets

There are four major lines of work related to the mining of API usage samples. They are: bug detection, code completion, API recommendation and API property inference. In this section we talk about how each line of work requires the mining of API usages.

- **Bug detection:** In the field of bug detection, API usage patterns are mined to find erroneous usage patterns in client code. The general method of operation of each tool that has been developed in this line of work is to compare the code of a client that uses a specific API against the similar usages of the API that can be mined from open source platforms. If there is a discrepancy between the client code and the popularly used usage patterns, then there might be a chance that the client code is buggy.

- **Code completion:** Code completion tools use mined API usage datasets to infer the popular ways in which API methods are invoked on API classes. With the information contained in the dataset, one can draw conclusions as to what the most popular features of an API are and these features are the ones that are suggested prominently by a code completion tool.

- **API recommendation:** In the field of API recommendation, we see a lot of tools that mine API usages on the fly. The aim of these tools is to provide a developer with API usage samples for a specific API such that it is easier for the developer to use that API. These tools ranked the API usage samples that are retrieved based on criteria such as relevance and popularity.

- **API property inference:** Finally, API property inference research is done to expose certain usage characteristics of APIs. For example, a study into the trends of API usage was performed by Mileva *et al.* [66]. Here they mined API usages to see as to what version of an API a client was using and what factors led to the client upgrading or downgrading that version. Other work includes that of Stylos *et al.* [82] where they tried to enhance documentation of APIs by looking at popular usages of the API.

Chapter 3

The Approach - fine-GRAPE

3.1 Foreword

The approach detailed here was originally published in the paper "A dataset for API Usage" in the data track at MSR 2015. This paper has been subsequently invited to a journal extension to the EMSE journal where it is currently under submission with the title "fine-GRAPE: fine-Grained APi usage Extractor – An Approach and Dataset to Investigate API Usage". I was the first author on both these papers and they were written in collaboration with Dr. Alberto Bacchelli.

3.2 Introduction

In the past we have seen approaches such as frequent itemset mining and pattern matching to be the preferred API usage mining techniques. Other than the technique that is based on bytecode analysis none of the other techniques take type information into account. This can lead to inaccuracies in the analysis that is performed on the data that is gathered using one of these techniques. We aim to rectify this by presenting a powerful, scalable and type-aware approach to mine API usages from open source platforms.

In this chapter we present the 2-step approach that we use to collect fine-grained type-resolved API usage information. (1) We collect data on project level API usage from projects mining open source code hosting platforms (we target such platforms due to the large number of projects they hosted) and use it to rank APIs according to their popularity in order to select an interesting sample of APIs to form our dataset; (2) We apply our technique, fine-GRAPE, to gather fine-grained type-based information on API usages and collect historical usage data by traversing the history of each file of each API client.

3.3 Mining of coarse grained usage

In the construction of this dataset, we limit ourselves to the Java programming language, one of the most popular programming languages currently in use [7]. This reduces the types of programs that we can analyze, but has a number of advantages: (1) Due to the popularity of Java there would be a large source of API client projects available for analysis (2) Java is a statically typed language, thus making the collection of type-resolved API usages easier. (3) It allows us to have a more defined focus and more thoroughly test and refine fine-GRAPE Future work can be conducted to extend it to other typed-languages, such as C#.

To ease the collection of data regarding project dependencies on APIs, we found it useful to focus on projects that use build automation tools. In particular, we collect data from projects using Maven, one of the most popular Java build tools. Maven employs the use of a Project Object Model (POM) files to describe all the dependencies and targets of a certain project. POM files contain artifact ID and version of each project's dependency, thus allowing us to know exactly which APIs (and version) a project uses. The following is an example of a POM file entry:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.8.2</version>
</dependency>
```

3.4 Fine-grained API usage

To ensure that precise API usage information is collected, one has to reliably link each method invocation or annotation usage to class in the API to which it belongs. For our purpose this can be achieved in four ways:

Text matching: This is one of the most frequently used techniques to mine API usage. For example, it has been used in the investigation into API popularity performed by Mileva *et al.* [66]. The underlying idea is to match explicit imports and corresponding method invocations directly in the text of source code files.

Bytecode analysis: Each Java file when compiled produces one or more class files. Class files contain Java bytecode that is platform independent. Another technique to mine API usage is to parse byte code in these class files to find all method invocations and annotation usages along with the class to which they belong to. This approach guarantees accuracy as the class files contain all information related to Java program in the Java file in question.

Partial program analysis: Dagenais *et al.* have created an Eclipse plugin called Partial Program Analysis (PPA) [25]. This plugin is able to parse incomplete files and recover type bindings on method invocations and annotations, thus identifying the API class to which a certain API usage belongs.

AST analysis: Syntactically correct Java files can be transformed into an Abstract Syntax Tree (AST). An AST is a tree based representation of code where each variable declaration, statement or invocation forms a node of the tree. This AST can be parsed by using a standard Java AST parser. The Java AST parser can also recover type based information at each step which aids in ensuring accuracy when it comes to making a connection between an API invocation and the class it belongs to.

All four of the aforementioned approaches can be applied for the purpose of collecting API usage data, but come with different benefits and drawbacks.

The text-matching-based approach proves especially problematic in the case of imported API classes that share method names, because method invocations may not be disambiguated without type-information. Although some analysis tools used in dynamic languages [30] handle these cases through the notion of 'candidate' classes, this approach is sub-optimal for typed languages where more precise information is available.

The bytecode analysis approach is more precise, as bytecode is guaranteed to have the most accurate information, but it has two different issues:

1. Processing class files requires these files to be available, which, in turn, requires being able to compile the Java sources and, normally, the whole project. Even though the projects under consideration use Maven for the purpose of building, unfortunately, this does not guarantee that they can be built. If a project is not built, then the class files associated with this project cannot be analyzed, thus resulting in a dropped project.

2. To analyze the history of method invocations it is necessary to checkout each version of every file in a project and analyze it. However, checking out every version of a file and then building the project can be problematic as there would be an ultra-large number of project builds to be performed. In addition to the time costs, there would still be no warranty that some data would not be lost due build failure.

The partial program analysis approach has been extensively tested by Dagenais *et al.* to show that method invocations can be type resolved in incomplete Java files. This is a massive advantage as it implies that even without building each API client one can still conduct a thorough analysis of the usage of an API artifact. However, the implementation of this technique relies on Eclipse context, thus all parsing and type resolution of Java files can only be done in the context of an Eclipse plugin. This requires that each and every Java file is imported into an Eclipse workspace before it can be analyzed. Unfortunately, this approach does not scale to a large number of projects.

Due to the various issues related to first three techniques, we find that the most suitable technique to employ is the AST based one. This technique utilizes the JDT Java AST Parser [86], *i.e.*, the parser used in the Eclipse IDE for continuous compilation in background. This parser handles partial compilation: When it receives in input a source code file and a Java Archive (JAR) file with possibly imported libraries, it is capable of resolving the type of methods invocation and annotations of everything defined in the code file or in the provided jar.

We created fine-GRAPE that utilizes the aforementioned AST parsing technique that is able to collect the entire history of usage of API artifacts over different versions. In practice, we downloaded all the JAR files corresponding to the releases of the API projects chosen. This had to be done manually from the Maven central site, however in the future we plan on automating this process. Then, fine-GRAPE uses Git to obtain the history of each file in the client projects and runs on each file retrieved from the repository and the JAR with the corresponding version of the API that the client project declares in Maven at the time of the commit of the file. The fine-GRAPE leverages the visitor pattern that is provided by the JDT Java AST parser to visit all nodes in the AST of a source code file of the type method invocation or annotation. These nodes are type resolved and are stored in a temporary data structure while we parse all files associated with one client project. This results in accurate type-resolved method invocation references for the considered client projects through their whole history. Once the parsing is done for all the files and their respective histories in the client, all the data that has been collected is transformed into a relational database model and is written to the database.

An API usage dataset can also contain the information on the method, annotations and classes that are present in every version of every API for which usage data has been gathered such that any kind of complex analysis can be performed. In the previous steps we have already downloaded the API JAR files for each version of the API that is used by a client. These JAR files are made up of compiled class files, where each class file relates to one Java source code file. fine-GRAPE then analyzes these JAR files with the help of the bytecode analysis tool ASM [19], and for each file the method, class and annotation declarations are extracted. For each of these mined artifacts we can also see if they have been marked as deprecated. In Java a deprecated entity is generally marked in source code with the annotation `@deprecated`, the corresponding artifact is also marked as deprecated in the database.

3.5 A Dataset for API Usage

Our dataset is constructed using data obtained from the open source code hosting platform GitHub. GitHub stores more than 10 million repositories [38] written in different languages and using a diverse set of build automation tools and library management systems.

3.5.1 Coarse-grained API usage: The most popular APIs

To determine the popularity of APIs on a coarse-grained level (*i.e.*, project level), we parse POM files for all GitHub based Java projects that use Maven (ca. 42,000). The POM files were found after looking at the master branch of approximately 250,000 active Java projects that are hosted on GitHub.[1] Figure 3.1 shows a partial view of the results with the 20 most popular APIs in terms of how many GitHub projects depend on them.

This is in-line with a previous analysis of this type published by Primat as a blog post [73]. Interestingly, our results show that JUnit is by far the most popular, while Primat's results report that JUnit is just as popular as SLF4J. We speculate that this discrepancy can be caused by the different sampling approach (he sampled 10,000 projects on GitHub, while we sampled about 42,000 on GitHub), further research can be conducted to investigate this aspect more in detail.

[1]As marked by GHTorrent [38] in September 2014

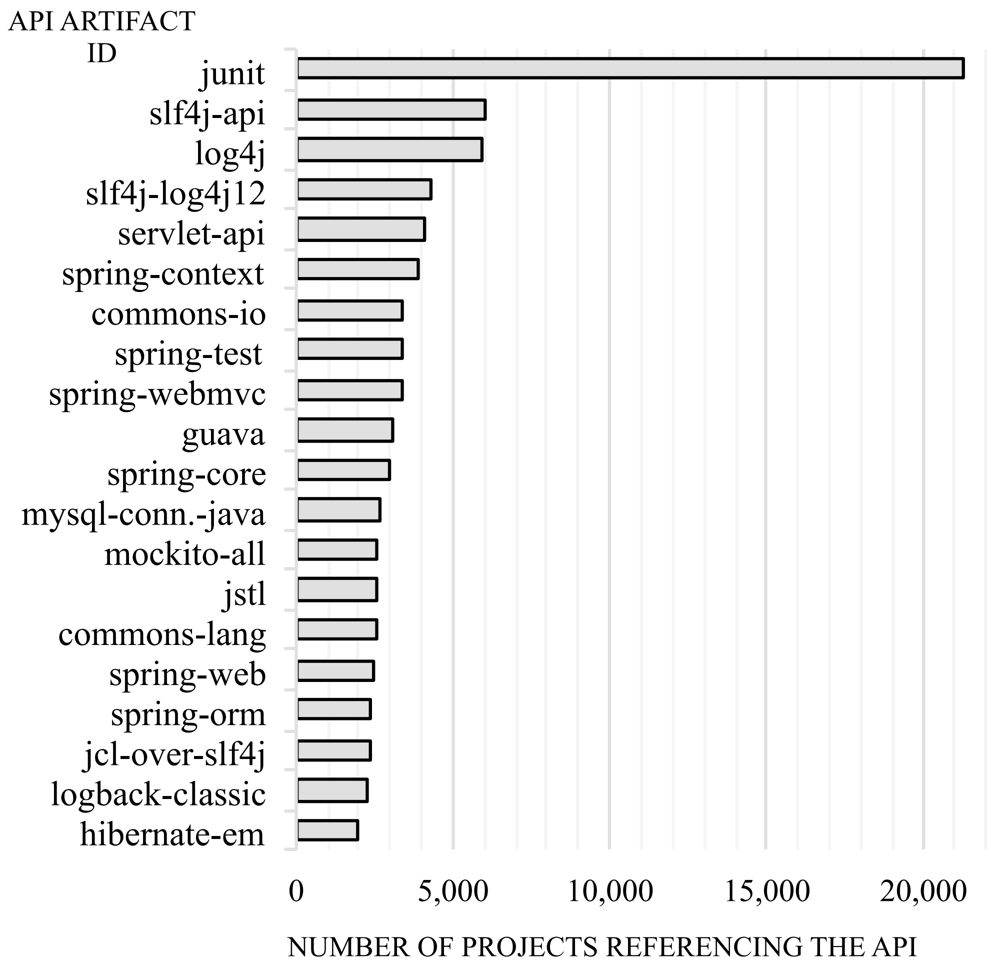

Figure 3.1: Popularity of APIs referenced on Github

3.5.2 Selected APIs

We used our coarse-grained analysis of popularity as a first step to select API projects to populate our database. To ensure that the selected API projects offer rich information on API usage and its evolution, rather than just sporadic use by a small number of projects, we consider projects with the following feature: (1) have a broad popularity for their public APIs (*i.e.*, they are in the top 1% of projects by the number of client projects), (2) have an established and reasonably large code base (*i.e.*, they have at least 150 classes in their history), (3) and are evolved and maintained (*i.e.*, they have at least 10 commits per week in their lifetime). Based on these characteristics, we eventually select the five APIs summarized in Table 3.1, namely Spring, Hibernate, Guava, and Guice and Easymock. We decide to remove JUnit, being an outlier in popularity and having a small code base that does not respect our requirements. We keep Easymock, despite its small number of classes and relatively low amount of activity in it's repository (ca. 4 commits per week) to add variety to our sample. The chosen APIs are used by clients in different ways: *e.g.*, Guice clients use it through annotations, while Guava clients instantiate an instance of a Guava class and then interact with it through method invocations.

In the following, a brief explanation of the domain of each API:

1. **Guava** is the new name of the original Google collections and Google commons APIs. It provides immutable collections, new collections such as multiset and multimaps and finally some new collection utilities that are not provided in the Java SDK. Guava's collections can be accessed by method invocations on instantiated instances of the classes built into Guava.

2. **Guice** is a dependency injection library provided by Google. Dependency injection is a design pattern that separates behavioral specification and dependency resolution. Guice allows developers to inject dependencies in their applications with the usage of annotations.

3. **Spring** is a framework that provides an Inversion of Control(IoC) container. This allows developers to access Java objects with the help of reflection. The Spring framework comprises of a lot of sub projects, however we choose to focus on just the spring-core, spring-context and spring-test modules due to their relatively high popularity. The features provided by Spring are accessed in a mixture of method invocations and annotations.

4. **Hibernate Object Relational Mapping (ORM)** provides a framework for mapping an object oriented domain to a relational database domain. It is made up of a number of components that can be used, however we focus on just two of the more popular one *i.e.*, hibernate-core and hibernate-entity manager. Hibernate exposes its APIs as a set of method invocations that can be made on the classes defined by Hibernate.

5. **Easymock** is a testing framework that allows for the mocking of Java objects during testing. Easymock exposes its API to developers by way of both annotations and method invocations.

3.5.3 Data Organization

We apply the approach outlined in Section 3.2 and store all the data collected from all the client GitHub projects and API projects in a relational database, precisely PostgreSQL. We have chosen a relational database because the usage information that we collect can be naturally expressed in forms of relations among the entities. Also we can leverage SQL functionalities to perform some initial analysis and data pruning.

Figure 3.2 shows the database schema for our dataset. On the one hand we have information for each client project: The PROJECTS table is the starting point and stores a project's name and its unique ID. Connected to this we have PROJECTDEPENDENCY table,

Table 3.1: Subject APIs

API & GitHub repo	Inception	Releases	Unique Entities	
			Classes	Methods
Guava google/guava	Apr 2010	18	2,310	14,828
Guice google/guice	Jun 2007	8	319	1,999
Spring spring-framework	Feb 2007	40	5,376	41,948
Hibernate hibernate/hibernate-orm	Nov 2008	77	2,037	11,625
EasyMock easymock/easymock	Feb 2006	14	102	623

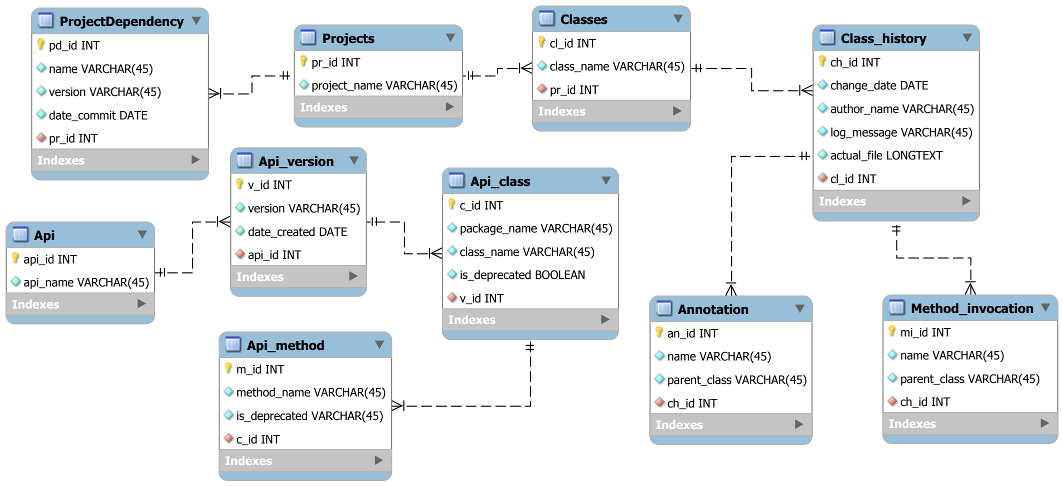

Figure 3.2: Database Schema For The Fine-grained API Usage Dataset

which stores information collected from the Maven POM files about the project's dependencies. We use a DATE_COMMIT attribute to trace when a project starts including a certain dependency in its history. The CLASSES table contains one row per each uniquely named class in the project; in the table CLASS_HISTORY we store the different versions of a class (including its textual content, ACTUAL_FILE) and connect it to the tables METHOD_INVOCATION and ANNOTATION where information about API usages are stored. On the other hand, the database stores information about API projects, in the tables prefixed with API. The starting point is the table API that stores the project name and it is connected to all its versions (table API_VERSION, which also stores the date of creation), which are in turn connected classes (API_CLASS) and their methods (API_METHOD) that also store information about deprecation. Note that in the case of annotations we do not really collect them in a separate table as annotations are defined as classes in Java.

A coarse-grained connection between a client and an API is done with a SQL query on the tables PROJECTDEPENDENCY, API and API_VERSION. The finer-grained connection is obtained by also joining METHOD_INVOCATION/ANNOTATION and API_CLASS on parent class names & METHOD_INVOCATION/ANNOTATION and API_METHOD on method names.

The full dataset is available as a PostgreSQL data dump on FigShare [78], under the CC-BY license. For platform limitations on the file size the dump has been split in various tar.gz compressed files, for a total download size of 51.7 GB. The dataset uncompressed requires 62.3 GB of disk space.

3.5.4 Introductory Statistics

Table 3.2 shows an introductory view about the collected usage data. In the case of Guava for example, even though version 18 is the latest (see Table 3.1), version 14.0.1 is the most popular among clients. APIs such as Spring, Hibernate and Guice predominantly expose their APIs as annotations, however we see also a large use of the methods they expose. The earliest usages of Easymock and Guice are outliers as GitHub as a platform was launched in 2008, thus the repositories that refer to these APIs were moved to GitHub as existing projects.

Table 3.2: Introductory usage statistics

API	Most popular release	Usage across history	
		Invocations	Annotations
Guava	14.0.1	1,148,412	—
Guice	3.0	59,097	48,945
Spring	3.1.1	19,894	40,525
Hibernate	3.6	196,169	16,259
EasyMock	3.0	38,523	—

3.6 Comparison to existing datasets

The work of Lammel *et al.* is the closest to the dataset we created with fine-GRAPE. They target open source Java projects hosted on the Sourceforge platform and their API usage mining method relies on type resolved ASTs. To acquire these type resolved ASTs they build the APIs client projects and resolve all of its dependencies. This dataset contains a total of 6,286 client projects that have been analyzed and the invocations for 69 distinct APIs have been identified.

Our dataset as well as that of Lammel *et al.* target Java based projects, though the clients that have been analyzed during the construction of our dataset were acquired from GitHub as opposed to Sourceforge. Our approach also relies on type resolved Java ASTs, but we do not build the client projects as fine-GRAPE is based on a technique able to resolve parsing of a standalone Java source file. In addition, the dataset by Lammel *et al.* only analyzes the latest build. In terms of size this dataset is comprised of usage information gleaned from 20,263 projects as opposed to the 6,286 projects that make up the Lammel *et al.* dataset.

However, this dataset contains information on only 5 APIs whereas Lammel *et al.* identified usages from 69 APIs.

Chapter 4

Investigating Properties of APIs

4.1 Foreword

The first two studies that have been performed here have been included in the paper "fine-GRAPE: fine-Grained APi usage Extractor – An Approach and Dataset to Investigate API Usage" which is currently in submission at the EMSE journal. On this paper I was the first author and I was also in charge of aggregating the results and performing analysis. This paper was co-written with Dr. Alberto Bacchelli. The third study is based on my work in Atlanta at the Georgia Institute of Technology under the supervision of Dr. Alessandro Orso. This study is still under work and is targeted for MSR 2016.

4.2 Introduction

In chapter 3 we see the technique fine-GRAPE has been developed which aids in the gathering of type-checked API usages from large scale open source platforms such as GitHub. Using this technique, we have constructed our own dataset for 5 APIs. An API usage dataset and the approach used to create said dataset is not a large enough contribution, to this end we perform a couple of analyses that showcase the data in this dataset.

API usage datasets are generally a by-product of some kind of empirical analysis study that is performed or the development of a tool. An example of this can been seen in the work of Sourcerer [10] where a large API usage database has been created and based on which an API recommendation system has been developed. In this context we conduct experiments based on an already created database. Our hope is that this shows the versatility of the dataset constructed using our technique.

In this chapter we perform three studies on our dataset. The first study looks into the API version upgrade behavior of clients of APIs. We see as to whether or not clients lag behind the latest version of an API by a long time or by a short time. This can give us an insight into whether or not clients find it beneficial to upgrade the version of the API being used. The second study looks at the proportion of the various APIs that is being used. We try to see as to what parts of an API are popular and what parts are unpopular among users. We try to analyze as to what may contribute to making some of the features more popular than the other. The third study looks at the link between the popularity of API artifacts and the code quality of those features. We postulate that the more popular the feature the more bug reports will be filed about that feature. We investigate this hypothesis by way of four research questions.

Structure of the chapter. Section 4.3 presents the first study about the nature of client migration to new versions of an API. In section 4.4 we present the second study on the percentage of features of an API that are used. Section 4.5 elaborates on the third analysis performed on the connection between API popularity and software quality. In section 4.6 we discuss the limitations of the dataset when it comes to performing the aforementioned case studies. Finally we conclude the chapter in section 4.7.

4.3 Study 1: Do clients of APIs migrate to a new version of the API?

As with other software systems, APIs also evolve over time. A newer version may replace an old functionality with a new one, may introduce a completely new functionality, or may fix a bug in an existing functionality. Some infamous APIs, such as Apache Commons-IO, are stagnating since long time without any major changes taking place, but to build our API dataset we took care of selecting APIs that are under active development, so that we could use it to analyze as to whether clients react to a newer version of an API.

4.3.1 Methodology

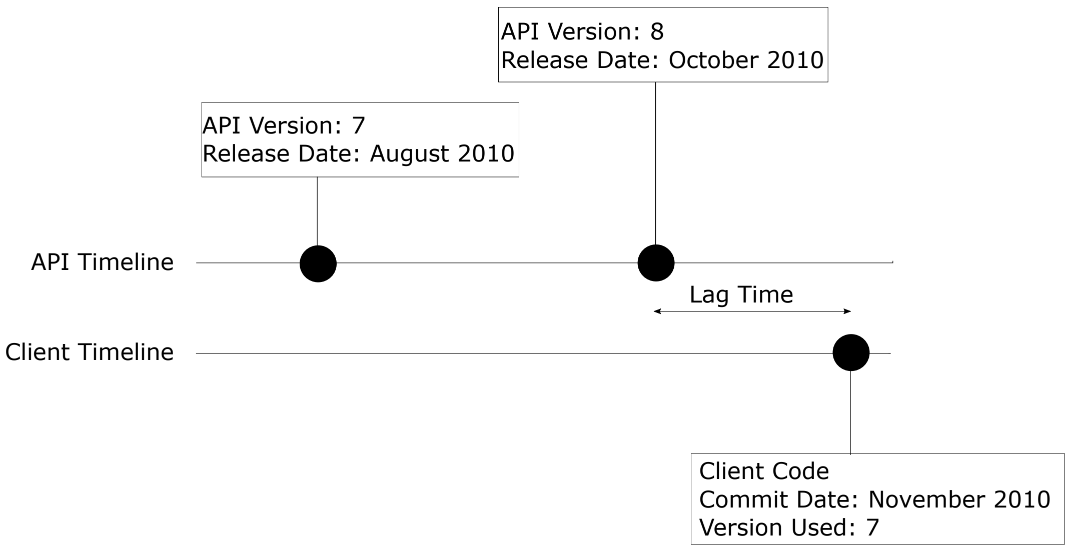

Figure 4.1: An example of the lag time metric inspired by McDonnell *et al.* [64]

We use the *lag time* metric, as previously defined by McDonnell *et al.* [64], to determine the amount of time a client is behind the latest release of an API that it is using. Lag time is defined as the amount of time elapsed between a client's API reference and the release date of the latest version. A client lags if it uses an old version of an API when a newer version has already been released. For example, in Figure 4.1, client uses API version 7 despite version 8 being already released. The time difference between the client committing code using an older version and the release date of a newer version of the API is measured as the lag time.

In practice, we consider the commit date of each method invocation (this is done by performing a join on the METHOD_INVOCATION and CLASS_HISTORY tables), determine the version of the API that was being used by the client at the time of the commit (the PROJECT_DEPENDENCY table contains information on the versions of the API being used by the client and the date at which the usage of a certain version was employed), then consider the release date of the latest version of the API that existed at the time of the commit (this data can be obtained from the API_VERSION table in the database), and finally combine this information to calculate the lag time for each reference to the API and plot the probability density.

Lag time can indicate how far a project is at the time of usage of an API artifact, but it does not give a complete picture of the most recent state of all the clients using an API. To this end, we complement lag time analysis with the analysis of the most popular versions of each API, based on the latest snapshot of each client of the API (we achieve this by querying the PROJECT_DEPENDENCY table to get the latest information on clients).

4.3.2 Results

Results are summarized in four figures. Figure 4.2 shows the probability density of lag time in days, as measured from API clients, and Figure 4.3 shows the distribution of this lag time. Figure 4.3 shows frequency of adoption of specific releases: the three most popular ones, the latest release (available at the creation of this dataset), and all the other releases. Table 4.1 further specify the dates in which these releases were made public and provides absolute numbers. Finally, Figure 4.5 depicts the frequency and number of releases per API. The data we have ranges from 2004 to 2014, however for space reasons we only depict the range 2009 to 2014. Each year is divided into 3 slots of 4 month periods, and the number of releases in each of these periods is depicted by the size of the black circle.

Guava. In the case of the 3,013 Guava clients on GitHub the lag time varies between 1 day and 206 days. The median lag time for these projects is 67 days. The average amount of time a project lags behind the latest release is 72 days. Figure 4.2 shows the cumulative distribution of lag time across all clients. Since Guava generally releases 5 versions on average per year, it is not entirely implausible that some clients maybe one or two versions behind at the time of usage of an API artifact.

Although the latest (as of September 2014) version of Guava is 18, the most popular one is 14 with almost one third of the clients using this version (as shown in Figure 4.3). Despite 4 versions being released after version 14 none of them figure in the top 5 of most popular versions. Version 18 has been adopted by very few clients

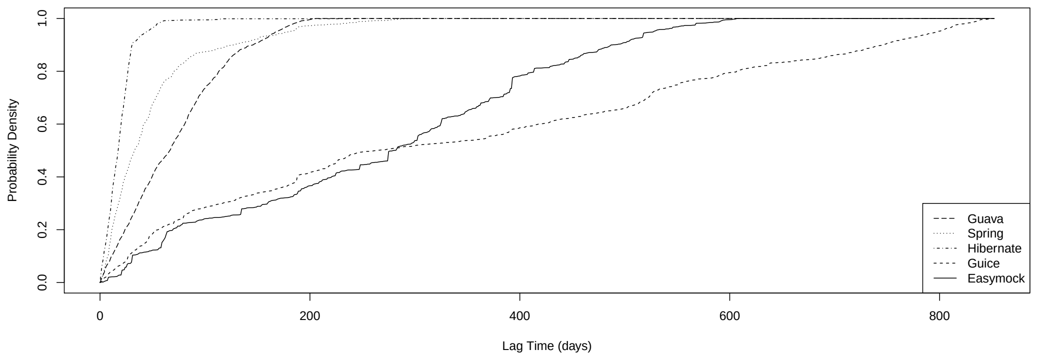

Figure 4.2: Probability density of lag time in days, by API

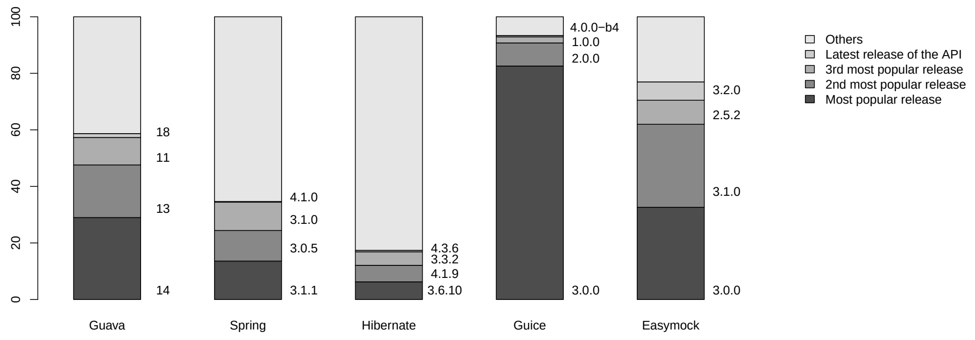

Figure 4.3: Proportion of release adoption, split in the 3 most popular, the latest, and all the other releases, by API

(41 out of 3,013). None of the other newer versions (16 and 17) make it in the top 5 either.

Spring. Spring clients lag behind the latest release up to a maximum of 304 days. The median lag time is 33 days and the first quartile is 15 days. The third quartile of the distribution is 60 days. The average amount of lag time for the usages of various API artifacts is 50 days. Spring is a relatively active API and releases an average of 7 versions (including minor versions and revisions) per year (Figure 4.5).

At the time of collection of this data, the Spring API had just released version 4.1.0 and only a small portion (30) of projects have adopted it. The most popular version is 3.1.1 (2,013 projects) as is depicted in Figure 4.3. We see that despite the major version 4 of the Spring API being released in December 2013, the most popular major

Table 4.1: Publication date, by API, of the 3 most popular and latest releases, sorted by the number of their clients

API	Release	Release Date	Num of clients	(%)
Guava	14	03-2013	868	(29%)
	13	08-2012	557	(19%)
	11	02-2012	291	(10%)
	18	08-2014	41	(1%)
Spring	3.1.1	02-2012	2,013	(14%)
	3.0.5	10-2010	1,602	(11%)
	3.1.0	12-2011	1,489	(10%)
	4.1.0	10-2014	30	(0.2%)
Hibernate	3.6.10	02-2012	376	(6%)
	4.1.9	12-2012	352	(6%)
	3.3.2	06-2009	288	(5%)
	4.3.6	07-2014	32	(0.5%)
Guice	3.0.0	03-2011	536	(83%)
	2.0.0	07-2009	53	(8%)
	1.0.0	05-2009	14	(2%)
	4.0.0-b4	03-2014	3	(0.5%)
Easymock	3.0.0	05-2010	211	(33%)
	3.1.0	11-2011	190	(29%)
	2.5.2	09-2009	55	(9%)
	3.2.0	07-2013	42	(6%)

version remains 3. In our dataset, 344 projects still use version 2 of the API and 12 use version 1.

Hibernate. The maximum lag time observed over all the usages of Hibernate artifacts is 271 days. The median lag time is 18 days, and the first quartile is just 10 days. The third quartile is also just 26 days. The average lag time over all the invocations is 19 days. We see in Figure 4.2 that most invocations to Hibernate API do not lag behind the latest release considerably, especially in relation to the other APIs, although a few outliers exist. Hibernate releases 17 versions (including minor versions and revisions) per year (Figure 4.5).

Version 4.3.6 of Hibernate is the latest release that available on Maven central at the dataset creation time. A very small portion of projects (32) use this version, and the most popular version is version 3.6.10, *i.e.*, the last release with major version 3. We see that a large number of clients have migrated to early versions of major version 4. For instance, version 4.1.9 is almost (352 projects versus 376 projects) as popular as version 3.6.10 (shown in Figure 4.3). Interestingly, in the case of Hibernate, from our data we see that there is not a clearly dominant version as all the other versions of Hibernate make up about three fourths of the current usage statistics.

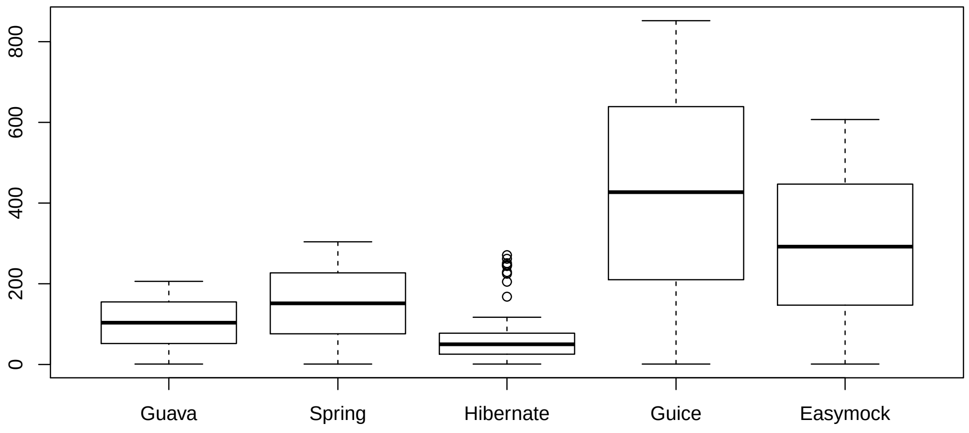

Figure 4.4: Lag time distribution in days, by API

Guice. Among all usages of the Guice API, the largest lag time is 852 days. The median lag time is 265 days and the first quartile of the distribution is 80 days. The average of all the lag times is 338 days. The third quartile is 551 days, showing that a lot of projects have a very high lag time. Figure 4.2 shows the cumulative distribution of lag times across all Guice clients. Guice is a young API and, relatively to the other APIs, releases are few and far between (10 releases over 6 years, with no releases on 2010 or 2012, Figure 4.5).

The latest version of Guice that has been released, before the construction of our dataset, is the fourth beta of version 4 (September 2014). Version 3 is unequivocally the most adopted version of Guice, as seen in Figure 4.3. This version was released in March of 2011 and since then there have been betas for version 4 released in 2013 and 2014. We speculate that this release policy may have led to most of the clients sticking to an older version and preferring not to transition to a beta version.

Easymock. Clients of Easymock display a maximum, median, and average lag time of 607, 280, and 268 days, respectively. The first quartile and third quartile in the distribution are 120 and 393 days, respectively. Figure 4.2 shows the large number of projects that have a large amount of lag, relatively to the analyzed projects. Easymock is a small API, which had 12 releases, after the first, over 10 years (Figure 4.5).

The most recent version of Easymock is 3.3.1, released in January 2015. However, in our dataset we record use of neither that version nor the previous one (3.3.0). The latest used version is 3.2.0, released in July 2013, with 42 clients. Versions 3.0.0 and 3.1.0 are the most popular (211 and 190 clients) in our dataset, as seen in Figure 4.3.

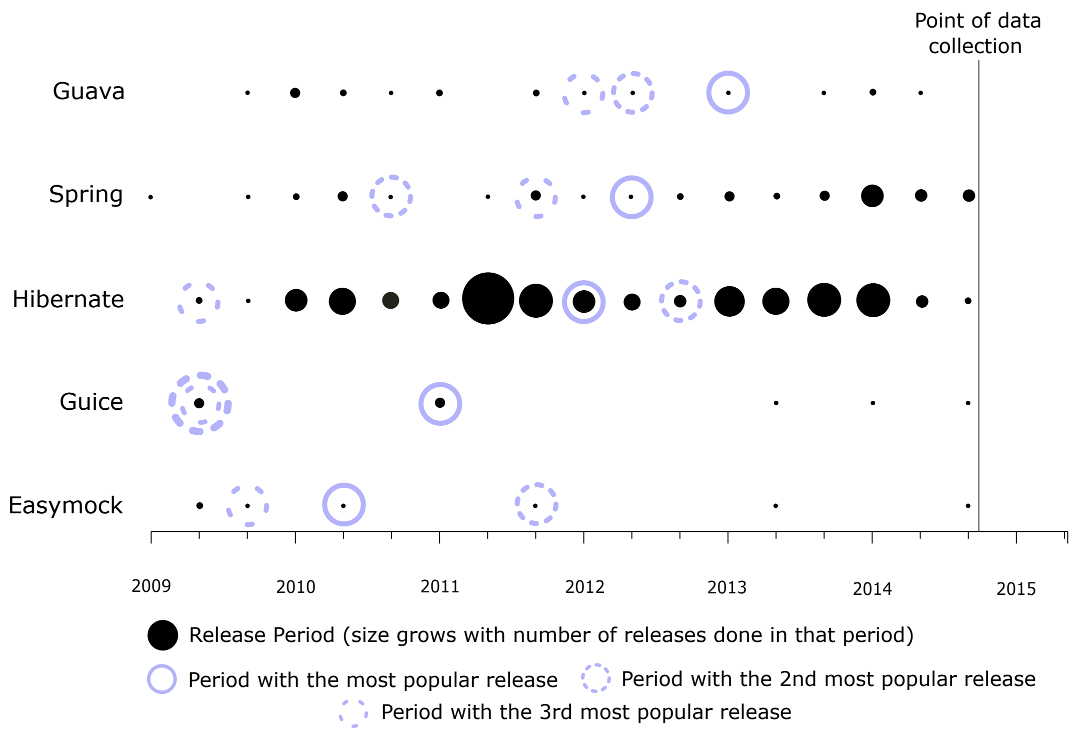

Figure 4.5: Release frequency for each API from 2009 (the dataset covers from 2004)

> Version 2.5.2 and 2.4.0 also figure in the top three in terms of popularity, despite being released in 2009 and 2008.

4.3.3 Discussion

Our analysis lets emerge an interesting relation between the frequency of releases of an API and the behavior of its clients. By considering the data summarized in Figure 4.5, we can clearly distinguish two classes of APIs: ‘frequent releaser’ APIs (Guava, Hibernate and Spring) and ‘non-frequent releaser’ APIs (Guice and Easymock).

For all the APIs under consideration we see that there is a tendency for clients to hang back and to not upgrade to the most recent version. This is especially apparent in the case of the ‘frequent releaser’ APIs Guava and Spring: For these APIs, the older versions are far more popular and are still in use. In the case of Hibernate, we cannot get an accurate picture of the number of clients willing to transition because the version popularity statistics are quite fractured. This is a direct consequence of the large number of releases that take place every year.

For Guice and Easymock (‘non-frequent releaser’ APIs), we see that the latest version is not popular. However, for Guice the latest version is a beta and not an official release, thus we do not expect it to be high in popularity. In the case of Easymock, we see that the latest version (*i.e.*, 3.3.1) and the one preceding that (*i.e.*, 3.3.0) are not at all be used.

In general, we do see that most clients of 'non-frequent releaser' APIs use a more recent version compared to clients of 'frequent releaser' APIs.

By looking at Figures 4.2 and 4.4, we also notice how the lag time of 'frequent releaser' APIs' clients is significantly lower than of 'non-frequent releaser' APIs' clients. This relation may have different causes: For example, 'non-frequent releaser' APIs' clients may be less used to update the libraries they use to more recent versions, they may also be less prone to change the parts of their code that call third-party libraries, or code that calls APIs that have non-frequent release policy may be more difficult to update. Testing these hypothesis goes beyond the scope of this paper, but with our dataset researchers can do so to a significant extent. Moreover, using fine-GRAPE, information about more APIs can be collected to verify whether the aforementioned relations hold with statistically significant samples.

4.4 Study 2: How much of an API is broadly used?

Many APIs are under constant development and maintenance. Some API producers do this to evolve features over time and improve the architecture of the API; others try to introduce new features that were previously not present. All in all, many changes take place in APIs over time [41]. Here we analyze which the features (methods and annotations) introduced by API developers are taken on board by the clients of these APIs.

This analysis is particularly important for developers or maintainers to know whether their efforts are useful and to decide to allocate more resources (e.g., testing, refactoring, performance improvement) in more used parts of their API, as resulting returns on investment may be greater. Moreover, API users may have more interest in reusing popular API features, as they are probably better tested through users [83].

4.4.1 Methodology

For each of the APIs, we have a list of features in the API_METHOD and API_CLASS tables [78]. We also have the usage data of all features per API that has been accumulated from the clients in the METHOD_INVOCATION and ANNOTATION tables. Based on this, we can mark features of the API have been used by clients. We can also count how many clients use a specific feature, thus classifying each feature as: (1) *hotspot*, in the top 15% of features in term of usage; (2) *neutral*, features that have been used once or more but not in the top 15% and (3) *coldspot*, if not used by any client. This is the same classification used by Thummalapenta and Xie [83] in a similar study (based on a different approach) on the usage of frameworks' features.

To see which used features were introduced early in an APIs lifetime, we can use the API_VERSION table to augment the date collected above with accurate version information per feature; then, for each of the used features, we see which version is the lowest wherein that feature has been introduced.

4.4.2 Results

The overall results for our analysis are summarized in Figures 4.6, 4.7, and 4.8. The first shows a percentage breakdown of usages of API features (left-hand side) and classes (right-hand side); the second and third report the probability distribution of the logarithm of the number of clients per API features, for ‘non-frequent releaser’ APIs and ‘frequent releaser’ APIs, respectively.

Generally, we see that the proportion of used features is never higher than 20% (Figure 4.6) and that the number of clients that use the features has a heavily right skewed distribution, which is slightly flattened by considering the logarithm (Figures 4.7 and 4.8). Moreover, we do not see a special behavior in this context of clients of ‘non-frequent releaser’ APIs vs. clients of ‘frequent releaser’ APIs.

In the following, we present the breakdown of the usage based on the definitions above.

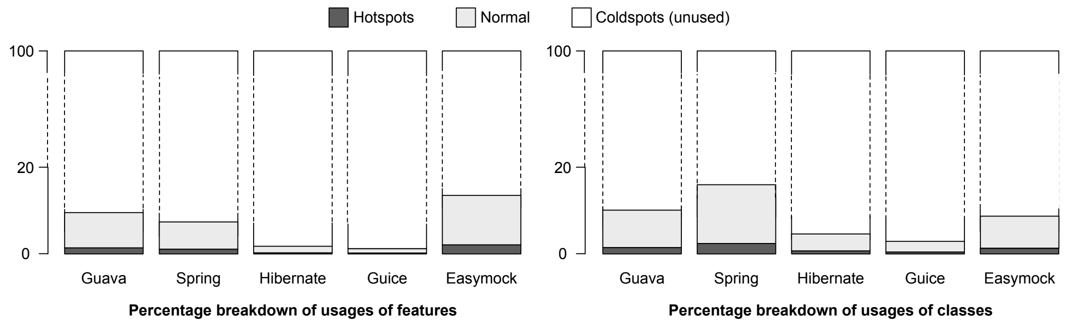

Figure 4.6: Percentage breakdown of usage of features for each of the APIs

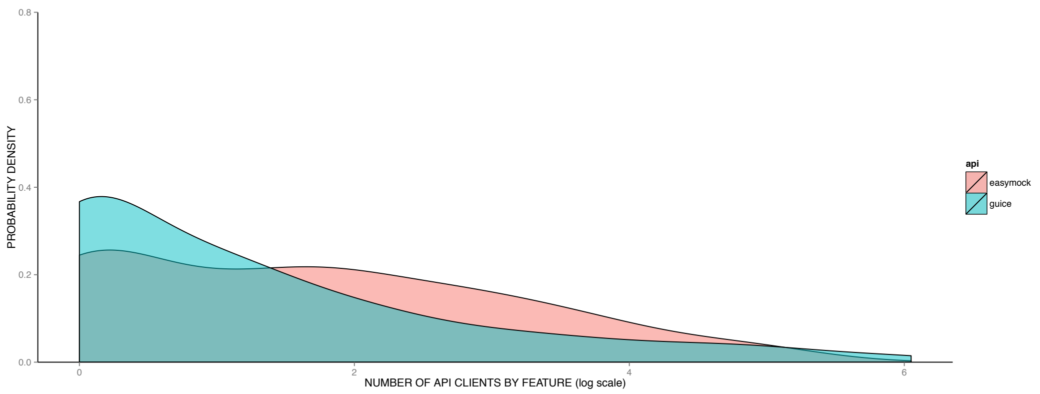

Figure 4.7: Probability distribution of (log) number of clients per API features, by ‘non-frequent releaser’ APIs

Guava. Only 9.6% of the methods in Guava are ever used; in absolute numbers, out of 14,828 unique public methods over 18 Guava releases, only 1,425 methods are ever

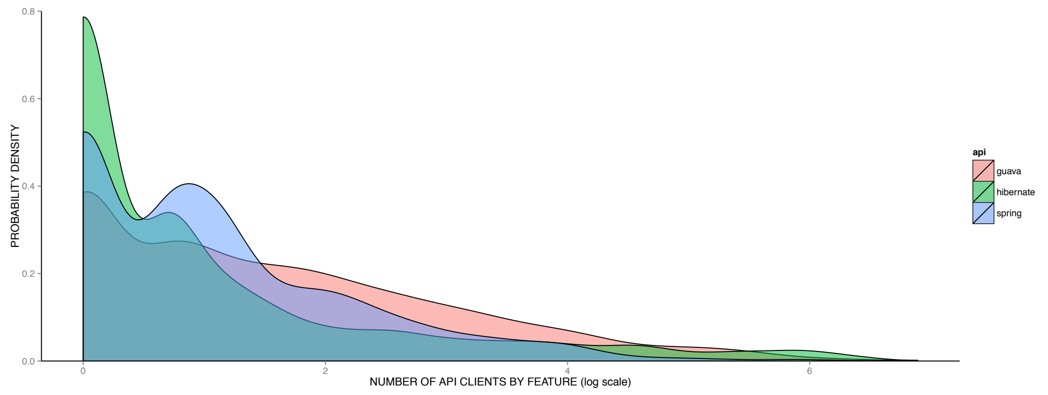

Figure 4.8: Probability distribution of (log) number of clients per API features, by 'frequent releaser' APIs

used. Looking at the used methods, we find that 214 methods can be classified as hotspots. The rest (1,211) are classified as neutral spots. The most popular method from the Guava API is `newArrayList` from the class `com.google.common.collect.Lists` class and it has 986 clients using it.

Guava provides 2,310 unique classes over 18 versions. We see that only 235 (10%) of these are ever used by at least client. Furthermore, only 35 of these classes can be called hotspots in the API. A further 200 classes are classified as neutral. And we can classify a total of 2,075 classes as coldspots as they are never used. The most popular class is used 1,097 times and it is `com.google.common.collect.Lists`.

With Guava we see that 89.4% of the usages by clients of Guava relate to features that have been introduced in version 3 that was released in April 2010. Following which 7% of the usages relate to features that were introduced in version 10 that was released in October 2011.

Spring. Out of the Spring core, context and test projects, we see that 7.4% of the features are used over the 40 releases of the API. A total of 840 features have been used out of the 11,315 features in the system. There are 126 features that can be classified as hotspots. Consequently, there are 714 features classified as neutral. The most popular feature is `addAttribute` from the class `org.springframework.ui.Model` and has been used 968 clients.

The Spring API provides a total of 1,999 unique classes. Out of these there are only 319 classes that are used by any of the clients of the Spring API. We can classify 48 of these classes as hotspot classes and the other 271 can be classified as neutral. We classify 1,680 classes as coldspots as they are never used. The most popular class has 2,417 clients and it is `org.springframework.stereotype.Controller`.

Looking deeper, we see that almost 96% of the features of Spring that are used by clients are those introduced in Spring version 3.0.0 that was released in December 2009.

Hibernate. From the Hibernate core and entitymanager projects we see that only 1.8% of the features are used. 756 out of the 41,948 unique public features provided over 77 versions of Hibernate have been used by clients in GitHub. Of these, 114 features that can be classified as hotspots and a further 642 features can be classified as neutral. The `getCurrentSession` method from the class `org.hibernate.SessionFactory` is the most popular feature, used by 618 clients.

Hibernate is made up of 5,376 unique classes. Out of these only 245 classes are used by clients. We can classify 37 of these classes as hotspots. The rest 208 classes are classified as neutral. We find that Hibernate has 5,131 coldspot classes. The most popular class is `org.hibernate.Session` with 917 clients using it.

In the case of Hibernate over 82% of the features that have been used were introduced in version 3.3.1 released in September 2008 and 17% of the features were introduced in 3.3.0.SP1 released in August 2008.

Guice. Out of the unique 11,625 features presented by Guice, we see that 1.2% (138) of the features are used by the clients of Guice. There are 21 features that are marked as being hotspots, 117 features marked as being neutral, and 11,487 classified as coldspots. The most popular provided by the Guice API is `createInjector` from class `com.google.inject.Guice` and is used by 424 clients.

The Guice API is made up of 2,037 unique classes that provide various features. Out of these only 61 classes are of any interest to clients of the API. We find that 9 of these classes can be classified as hotspots and the other 52 as neutral spots. This leaves a total of 1,976 classes as coldspots. The most popular class provided by Guice is `com.google.inject.Guice` and there are 424 clients that use it.

Close to 96% of the features of Guice that are popularly used by clients were introduced in its first iteration which was released on Maven central in May 2009.

Easymock. There are unique 623 features provided by Easymock, out of which 13.4% (84) are used by clients. This implies that 539 features provided by the API are never by used by any of the clients and are marked as coldspots. 13 features are marked as hotspots, while 71 features are marked as neutral. the The most popular feature is `getDeclaredMethod` from the class `org.easymock.internal.ReflectionUtils` and is used by 151 clients.

Easymock being a small API consists of only 102 unique classes. Out of these only 9 classes are used by clients. Only 1 can be classified as a hotspot class and the other 8 are classified as neutral spots. This leaves 93 classes as coldspots. The most popular class is `org.easymock.EasyMock` and is used by 205 clients.

We observe that 95% of the features that are used from the Easymock API were provided starting version 2.0 which was released in December 2005.

4.4.3 Discussion

We see that for Guava, Spring and Easymock, the percentage of usage of features hovers around the 10% mark. Easymock has the largest percentage of features that are used among the 5 APIs under consideration. This could be down to the fact that Easymock is also the smallest API among the 5. Previous studies such as that by Thummalapenta and Xie [83] have shown that over 15% of an API is used (hotspot) whereas the rest is not (coldspot). However, the APIs that they analyzed are very different to the ones that are here as they are all smaller APIs comparable to the size of Easymock, however none of them are of the size of the other APIs such as Guava and Spring. Also, their mining technique relied on code search engines and not on type resolved invocations.

In the case of Hibernate and Guice we see a much smaller percentage (1.8% and 1.2% respectively) of utilization of features. This is far lower than that of other APIs in this study. We speculate that due to the fact that the most popular features that are being used are also those that were introduced very early in the APIs life (version 3.3.1 in the case of Hibernate and version 1.0 in the case of Guice). These features could be classified as core features of the API. Despite API developers adding new features, there may be a tendency to not deviate from usage of these core features as these may have been the ones that made the API popular in the first place.

This analysis underlines a possibly unexpected low usage of API features in GitHub clients. Further studies, using our dataset, can be designed and carried out to determine which characteristics make certain feature more popular and guide developers to give the same characteristics to less popular features. Moreover, this popularity study can be used, for example, as a basis for developers to decide whether to separate more popular features of their APIs from the rest and provide them as a different, more supported package.

4.5 Study 3: Are popular parts of an API more likely to have bugs?

Large APIs such as Spring and Hibernate provide a lot of features that developers can use. However, the features that they provide are not always bug free and can generally have an adverse impact on a clients' code [57]. The presence of bugs in a version of an API can have a negative impact on the popularity of that version of the API or the popularity of the API itself [66].

As with most software projects, in an attempt to keep the number of bugs that ship with an API release to a minimum, API developers put a large emphasis on testing. To this end a lot of testing tools and metrics [40] have been developed to ensure and measure test quality [43]. The advent of tools such as JUnit [62] and Mockito [32] have made it easier for developers to test their code [8, 49]. These tools have now grown to be very popular [70] and are widely adopted for the purpose of testing Java code.

Despite the development of tools dedicated to testing of Java code, there are still bugs that can be found in the code of APIs. These bugs may be introduced in new features [54] that are published by the API or these bugs may manifest due to changes [51, 89] made in existing functionality. To limit the introduction of bugs there have been a number techniques [16, 72, 81] that have been developed to reduce the number of bugs that are introduced by developers.

Most of the work done till date in the field of bug detection related to APIs has been on the client side where the emphasis is on finding erroneous API usage patterns in client code. One of the earlier techniques developed is that of Wasylkowski *et al.* [88], where they inferred incorrect usage patterns of an API based on popular usage patterns that could be found in the open source world. Another technique was that of Thummalapenta and Xie [85] where they infer exception handling rules from exception handling code that can be found on code search engines. These rules that are inferred are then applied to a client's code to see if there are any errors in the exception handling mechanism. Monperrus *et al.* [67] postulate that bugs are introduced in code when there are some calls to an API that are missed, they check client code against popular usages of the API in question to see if the client code is indeed buggy. Finally, we see the technique developed by Gruska *et al.* [39], where they mine 6,000 real world projects in the Linux ecosystem and compare other client code to the usages that have been mined to detect erroneous patterns.

We observe that the field of detection/prediction of bugs in APIs based on the usage of the API itself is largely unexplored. With the aid of the large API usage dataset that we have at our disposal we would like to make a connection between the real world usage of an API artifact and the presence of a bug in that artifact. Intuitively there should be a connection between the parts of the system with the most bugs that are reported and the popularity of these parts in the open source software world. We test our hypothesis by first looking at how well tested the APIs that we at our disposal are. We then try to see if there indeed exists such a correlation between popular parts of the API and buggy parts of the API. Based on our findings we hope to make recommendations on the improvement of testing practices of API.

Structure of this section. In subsection 4.5.1 we describe the research goals of our work. Following which in subsection 4.5.2 we describe some of the results that have been gathered during this work. Future work that would serve and extension to this work or lead to the completion of this work is described in subsection 4.5.3.

4.5.1 Research Goals

We study the characteristics of testing related to APIs to understand whether API developers test their APIs in the best manner possible. There is a large dataset of API usage patterns that is available to us. Our hope is to gain insights into API testing and make recommendations to API developers based on our observations. Intuitively there should be a connection between the parts of the system with the most bugs that are reported and the popularity of these parts in the open source software world. Based on this intuition we try to answer some research questions that are enumerated below:

RQ1: Are tests made for APIs exhaustive? That is do they cover the API artifacts well?

RQ2: Is there a connection between API coverage and popularity of an API artifact? Are popular parts better covered?

RQ3: Is there a connection between bugs in an API and popular parts of the API? Are more bugs reported in popular parts?

RQ4: Are core features of an API the most popular parts?

4.5.2 Results

RQ1: Are tests made for APIs exhaustive? That is do they cover the API artifacts well?

Most mainstream APIs such as Hibernate or Guava have tests that ship along with the source code. Tests are vital for an API as the behavior of the API artifacts that are exposed to the API user have to be well tested before their use so as to prevent too many failures. If an API has far too many bugs, then there is a strong likelihood that such an API will not be adopted by many developers. These tests are generally written by developers during the development of a new feature. The tests have to follow a certain convention and many open source communities require that every new feature introduced is well tested. We examine the tests for each of the APIs to see as to whether the developer guidelines have been following proper testing practices as described by their communities and as to whether the API artifacts are truly well tested.

Test coverage is an effective way to see as to how many statements or methods are covered in testing [33] and whether or not API features are being tested. Coverage is generally expressed in percentage terms. A percentage of 80% implies that 80% of the statements in a codebase are hit during the execution of a test suite. Coverage tools can also give an indication as to whether or not a method is executed during the execution of a test suite.

Java tests are primarily written using the JUnit library. There are a couple of coverage tools such as JCov [5] and Cobertura [2] for Java that ship with the JDK or with Eclipse. These coverage tools can be run as a maven plugin, in standalone mode or as eclipse plugins. They run the test suite and produce a test coverage report in HTML or CSV format.

Using Cobertura we see that for Guava, Spring and Hibernate the coverage of code is in excess of 85%, thus making the tests quite exhaustive and covering large portions of the API. In the case of Guava, it is actually almost 97%, however this might be the result of automated generation of tests. We see that all the publicly exposed artifacts of the API are tested at least once and they all have a statement coverage of 100%.

However, despite such high coverage numbers we see that there are still bugs that reported in these APIs. This leads us to believe that a simple coverage metric as measured by Cobertura may not be sufficient. This metric does not tell us as to how many times a method is invoked from a test. Simply looking at whether a method is tested or not does not tell us as to whether a method is well tested or whether a method is given special focus by testers.

RQ2: Is there a connection between API coverage and popularity of an API artifact? Are popular parts better covered?

According to Begel and Zimmermann [13] one of the top 10 questions asked by developers is “How well does test coverage correspond to actual code usage by our customers?”. Since we have the usage data for five APIs, we try to find a correlation between the coverage of an API artifact and the actual popularity of that artifact.

Looking at just code coverage as before does not suit our purpose here. We need to see as to how many times a method is tested to determine its testing popularity and then compare this to its popularity with clients. For this purpose, we create two new strategies to measure the number of times an API method is tested, results from both are enumerated below:

Dynamic Coverage The first way to evaluate the comprehensive nature of a methods' testing is to measure how many times a method is executed during the execution of a test suite. In order to do this, we had to inject code into every method if the API, the objective of this code would be ping a counter Java Archive(JAR) file on the execution of this method. This JAR file maintains a count of the number of times a method has pinged it, thereby maintaining a count of the number of times a method is invoked during testing. This is a very reliable way to ensure that each method when invoked is counted. We call this mechanism to measure coverage as dynamic coverage at it is necessary that the test suite be executed in order to count the number of executions of a method. We execute this process on the Guava, Spring and Hibernate APIs.

In order to see as to whether a correlation exists between the number of times a method was invoked in testing and the number of times the method is invoked in the real world we carried out a Spearman correlation test [92]. This test revealed that for all the APIs, the correlation was both weak and insignificant. Thus we can conclude that there is no connection between the number of times a method is invoked during the execution of a test suite and the popularity of that method.

The lack of a correlation can be explained by two reasons:

- Counting each and every method that is executed during test suite execution results in private methods to be counted to a large degree. These private methods can never be called as API method calls.

- By counting every single execution of the method, there is an accurate count that is made. However, the popularity of method invocations in real world projects is counted by the number of times it is present in the source code and not the number of times the method is executed. Thus, trying to find a correlation here between these two methods would be impossible as they are essentially two different measures.

Static coverage Since the dynamic coverage method had a number of significant drawbacks, we decided to measure the number of times a method is invoked in a test suite by using a static method. A static method of counting does not involve running the tests. To get a static count, we would have to accurately count the number of times a method is invoked in each test case of a test suite. Here we define accuracy as counting of a method invocation only if we are sure that the type of the method invocation can be resolved. For the purpose of acquiring this count we apply the same type resolution techniques as fine-GRAPE a description of which can be found in section 3. The static method results in a more appropriate count of the number of times a method is invoked from the tests. Just as before we perform out static coverage analysis on the Guava, Spring and Hibernate APIs.

To see if whether there is a connection between the static count of invocation from tests and the popularity of a method we conducted a Spearman correlation test. This test shows us that the correlation between the popularity of a method and the static coverage metric of a method for all three APIs have a strong and significant correlation. This goes to show that the number of times a method is invoked in the real world is directly related to the number of times this method is invoked during testing.

We find that there is indeed a connection between the popularity of an API artifact and the number of times it is tested by API developers. By showing that such a connection does indeed exist, we can conclude that API testers appear have an indication as to what parts of the API will be the most popular and accordingly test those methods the most.

RQ3: Is there a connection between bugs in an API and popular parts of the API? Are more bugs reported in popular parts?

Based on the correlation that is seen between the popular parts and the better tested parts of the API we try to see if more bugs are present/reported in the popular parts. From the earlier research question we conclude that the more popular parts are better tested when it comes to the test coverage metric. However, test coverage says nothing about the quality of the tests, thus despite the method and statement coverage that is seen there is still a chance that bug reports originate from these parts. Also, since certain parts of an API are more used it is logical that most bug reports filed would be related only to those parts as real world developers might be using these parts in many different ways that may result in the discovery of a bug.

To identify if there are bugs in a certain API we can mine the issue trackers [46] used to maintain the workflow of the development team of the API. Issue trackers are used by large open source projects to facilitate the effective reporting of issues that are encountered during the usage of the project [14]. These issues can be bug reports, refactoring requests, feature requests or requests for improvements. Here we focus on the bug reports that have been made.

There are a number of issue trackers that open source projects can use, these are among others: JIRA [6], GitHub issue tracker [4] and BugZilla [1]. Projects such as Spring and Hibernate use JIRA as their issue tracker and others such as Guice, Guava and Easymock use the GitHub issue tracker. For the purpose of identifying the location of a bug we would have to make a connection between a bug report and the methods in the source code that change as a result of a bug fix commit.

There are a couple of well-known ways [80, 24] to connect bugs to commits that fix the bug in the code. We find that when trying to make a connection between a bug and a source code artifact, the simplest way would be to leverage the GitHub issue tracker. The GitHub issue tracker contains information as to which commit fixes a bug and the GitHub API can be leveraged to get a diff of the source code to see what part of the code actually changed and thus isolate the bug that was present in the code. However, projects such as Guava and Guice do not utilize the issue tracker in the manner that it was meant to be used. These projects are Google based projects and Google houses its own internal bug tracking system, and the bug fixes are only mirrored out to GitHub.

Since the GitHub based projects are non-starters, we focus on Spring and Hibernate both of which use JIRA as their issue tracker. The problem with JIRA though, is that there is no real connection with the source code. The issue on JIRA does not mention as to which commit fixed a certain bug. However, the developers for Spring and Hibernate use a commit convention wherein they mention the issue ID that is being fixed in a certain commit. By going through the entire git history of a project we can isolate all commit messages that mention an issue ID.

Once all the issue IDs that have been mentioned in a commit are collected, we check if the issue pertains to a bug or not. This can be done by querying the JIRA Rest API. This helps isolate the commits that fix bugs, and the commits before these commits that are said to bug inducing.

Now that all commits that fix bugs have been identified, we proceed to identify all the files that changed per commit. With the file list in hand, we retrieve a previous version of the same file and then do a deep AST diff between the two versions of the file. This AST diff checks if a method has changed between two different versions of the file. If the method has changed, then we mark that method as the one for which a bug report was made. Since these methods pertain to APIs and we would like to make connection with popularity of these methods, we do not include private methods in our analysis.

We try to find a correlation between the buggy methods and the popularity of the same. However, in the first instance we see that for both Hibernate and Spring the methods that are buggy are mostly those that have never been used by any of the open source clients that are in our API usage dataset. For the methods that we do see in our dataset, it is observed that they figure on the low end of spectrum of popularity and can be classified as unpopular methods. This leads us to conclude that the bugs are found in the unpopular parts of the API and not in the popular parts.

RQ4: Are core features of an API the most popular parts?

In the previous research question we see that the unpopular parts were the ones with more bug reports and bugs in general. We try to investigate as to why the popular parts are then the more stable parts of the API. We theorize that the most popular features used by developers of an API are those that have been present in the API since the introduction of that particular API. This could result in the fact that these features formed a core set of features of the API are well tested thus resulting in fewer bug reports. We try answering this question by looking at the usage of each API artifact and see as to in which version this feature was first introduced.

As in chapter 4, we try to analyze as to whether the features that are used the most by clients are also core features of the API. We see that in the case of both Spring and Hibernate (the other APIs are not considered here as there is no bug information available for them) the features that are popular are introduced in early versions, in the case of Spring 96% of the features were introduced in version 3.0.0 and in the case of hibernate 80% of the features were introduced in version 3.3.1. In the case of Spring we cannot claim that major version 1 or 2 introduces these features because these versions both predate GitHub and Maven central and hence do no figure in our usage dataset. In the case of Hibernate, there was a major revision that took place between major version 2 and major version 3. This leads to us marking the introduction of the popular features in version 3.3.1.

On the whole we see that popular features can be classified as core features of an API. This may be one of the causes as to why we see no bugs in these parts of the API, as they are by far the most tested parts of the API and are the features that made the API popular in the first place.

4.5.3 Future work

We have seen in the results that have been collected that there is no correlation between the popular parts of the API and the bugs that are present in the API. We have sought to explain this by saying that the root-cause behind this might be the fact that the features that are popular are also the core features of the API. We would like to further explain the observed phenomenon by carrying out some more analysis as is mentioned below.

One theory that explains the results of RQ3 could be that these bugs are reported by developers who are internal to the APIs themselves. The implication of this is that developers of the API itself report bugs that they encounter during development and report these to the bug tracker. However, we have as of yet to develop a plan to identify the internal developers for the APIs under consideration, as there is no publicly available list of the same.

Another theory that has been put forward is that the data that has been gathered in our API usage dataset from GitHub may not accurately reflect the actual usage of API artifacts. There is a possibility that the features that we mark as unpopular may actually have been developed for some specific use case that was requested by a company or corporate developer. We do not have access to this data as it has not been made open source. But we can make use of another datasource that is present on Maven central. All large projects/APIs that refer to the Sping and Hibernate APIs are listed on the Maven central site (there are approximately 3,000 projects for each API). This is would form a dataset that is different to the one we currently have. Any analysis performed on this dataset might show a different result.

4.6 Limitations

Mining API usages on such a large scale and to this degree of accuracy is not a trivial task. We report consequent limitations to our dataset.

Master branch. To analyze as many projects as possible on GitHub, we needed to checkout the correct/latest version of the project on GitHub. GitHub uses git as a versioning system which employs branches, thus making the task of automatically checking out the right version of the client challenging. We consider that the latest version of a given project would be labeled as the 'master' branch. Although this is a common convention [20], by restricting ourself to only the master branch there is a non-negligible chance that some projects are dropped.

Inner and Internal classes. The method we use to collect all data about the features provided by the APIs, identifies all classes and methods in the API that are publicly accessible and can be used by a client of the API. These can include inner public classes and their respective methods. Or it can also consist of internal classes that are used by the features of the API itself but not meant for public consumption. The addition of these classes and methods to our dataset can inflate our count of classes and methods per API. If a more representative count is desired, it would be necessary to create a crawler for the API documentation of each API that is hosted online.

Maven (central) We target only projects based on a specific build automation tool on GitHub, *i.e.*, Maven. This results in data from just a subset of Java projects on GitHub and not all the projects. This may in particular affect the representativeness of the sample of projects. We try to mitigate this effect by considering one of the most popular building tools in Java: Maven. Moreover, the API release dates that we consider in our dataset correspond to the dates in which the API were published on Maven central, rather than the dates in which the API were official released on their websites. This could have an impact on the computed lag time.

GitHub. Even though GitHub is a very popular repository for open source software projects, this sole focus on GitHub leads to the oversight of projects that are on other open source platforms such as Sourceforge and Bitbucket. Moreover, no studies have yet ensured the representativeness of GitHub projects with respect to industrial ones; on the contrary, as also recently documented by Kalliamvakou *et al.* [50], projects on GitHub are all open source and many of the projects may be developed by hobbyists. This may result in developers not conforming to standard professional software maintenance practices and, in turn, to abnormal API update behavior.

Hibernate. In the case of Hibernate, we could not retrieve data for version 2 or 1. This is due to the fact that neither of these versions were ever released on the maven central repository. This may have an impact on both of the case studies as the usage results can get skewed towards version 3 of the API.

4.7 Conclusion

We presented three studies that serve as an example for future users. The first study analyzes how much clients migrate to new versions of APIs. Besides confirming that clients tend not to update their APIs, this study highlights an interesting distinction between clients of APIs that frequently release new version and those that do not. For the former, the lag time is significantly lower. We deem this finding to deserve further research as it could potentially help API developers decide which release policy to adopt, depending on their objectives.

In the second study, we analyze which proportion of the features of the considered API is used by the clients. Results show that a considerably small portion of an API is actually used by clients in practice. We suspect that this may be a result of clients only using features that an API was originally known for as opposed to migrating to new features that have been provided by the API.

In the third study we have seen as to how well tested APIs are. All the APIs that we have considered have a high percentage of test coverage. When we measure both the dynamic and static coverage metrics defined by us, we see that the API is still quite well covered and in the case of the static metric there is a correlation between the well tested-ness of methods and the popularity of that method. Our original hypothesis for this work was that the more popular parts would be more bug infested. However, during the course of our investigation we see that this hypothesis is invalidated by some of our findings. We try to explain the reason behind this inverse phenomenon that is observed. To get a more accurate picture we will need to perform additional analysis.

We hope to have given an indication of the kind of studies that can be performed using our dataset and have exposed the versatility of this dataset.

Chapter 5

API Deprecation

5.1 Foreword

This chapter consists of content that has been included in the paper titled "How Do Developers React to API Deprecation? Case Study of Five Java APIs and their Clients on Github". On this work I was the first author and I was responsible for aggregating all the results for the entire paper. This paper is the result of a collaboration between Dr. Alberto Bacchelli, Dr. Romain Robbes and myself.

5.2 Introduction

An Application Programming Interface (API) is a definition of functionalities provided by a library or framework that is made available to an application developer. APIs promote the reuse of existing software systems [47]. In his landmark essay "No Silver Bullet" [17], Brooks argued that reuse of existing software was one of the most promising attacks on the essence of the complexity of programming:

> The most radical possible solution for constructing software is not to construct it at all.

Revisiting the essay three decades later [34], Brooks found that indeed, reuse continues to be the most promising attack on essential complexity. APIs enable this: to cite a single example, we found at least $15,000$ users of the Spring API.

However, reuse comes with the cost of dependency on other components. This is not an issue when said components are stable. But evidence shows that APIs are not always stable: The Java standard API for instance has an extensive deprecated API [1]. Studies of other APIs, such as Dig and Johnson's [28] find that API breaking changes are common.

In light of this, it is important to understand how developers use APIs. If there are several studies on the evolution of APIs, there are still few studies on how clients actually use them, and how they react—or not—to API changes.

[1]see `http://docs.oracle.com/javase/8/docs/api/deprecated-list.html`

Part of this is due to the challenge of gathering enough data on the clients of APIs. In recent years, the situation has improved as there are many online platforms that offer distributed revision and source code management systems, such as Sourceforge and GitHub. The open source projects on these platforms can be mined for API usage related data. If gathering the data is possible, this still leaves the challenge of processing large amounts of data. Most of these platforms house open source projects as they require their customers to make their projects open source.

To our knowledge, the largest study of the impact of deprecation on API clients is the 2012 study [76] which investigated the popularity of deprecated methods in the Squeak [56] and Pharo [15] software ecosystems. This study mined more than 2,600 Smalltalk [37] projects hosted on the SqueakSource platform. Based on the information gathered we looked at whether the popularity of deprecated methods either increased, decreased or remained as is after their deprecation.

We conduct a non-exact replication [48] of the previous study by Robbes *et al.* [76]. We study the reactions of the clients of 5 different APIs on client systems implemented in the statically-typed Java language, as opposed to the dynamically-typed Smalltalk. In addition, our dataset contains accurate API version information (extracted from Maven), and is much larger, as we investigate tens of thousands of Java projects hosted on GitHub. This study compares, contrasts and complements our previous findings on the reactions to API deprecations.

Structure of this chapter. Section 5.3 presents the related work on studies of APIs. Section 5.4 details the similarities and differences between this study and the previous deprecation study. Section 5.5 presents the the results of this study. We close the papers with an extended discussion of the results (Section 5.6), before concluding (Section 4.7).

5.3 Related Work

5.3.1 Studies of API Evolution

Several studies of API evolution have been performed, at smaller or larger scales.

Dig and Johnson studied and classified the API breaking changes in 4 APIs [27]; they did not investigate their impact on clients. They found that 80% of the changes were due to refactorings.

Cossette and Walker [23] studied five Java APIs in order to evaluate how API evolution recommenders would perform on these cases. They found that all recommenders handle a subset of the cases, but that none of them could handle all the cases they referenced.

The Android APIs have been extensively studied. McDonnell *et al.* [63] investigate stability and adoption of the Android API on 10 systems; the API changes are derived from Android documentation. They found that the API is evolving quickly, and that clients have troubles catching up with the evolution. Linares-Vsquez *et al.* also study the changes in Android, but from the perspective of questions and answers on Stack Overflow [58], not API clients directly.

Bavota *et al.* [12] study how changes in the APIs of mobile apps (responsible for defects if not reacted upon) correlate with user ratings: succesful applications depended on less change-prone APIs. This is one of the few large-scale studies, with more than 5,000 applications.

Wang *et al.* [87] study the specific case of the evolution of 11 REST APIs. Instead of analyzing API clients, they also collect questions and answers from Stack Overflow that concern the changing API elements. Espinha *et al.* [31] study 43 mobile applications depending on web APIs, and how they respond to web API evolution.

Raemaekers *et al.* measured the stability of software libraries [74] and analyzed the usage of the libraries by 140 industrial systems that used Maven in order to calibrate the model.

Finally, the work by Robbes *et al.* [76] and Hora *et al.* [45] are large-scale studies of API clients in the Pharo ecosystem. Our first study focused on API deprecations, while the second one focused on API changes that were not marked as deprecations beforehand.

5.3.2 Supporting API evolution

Many approaches have been developed to support API evolution and reduce the efforts of client developers. Chow and Notkin [22] present an approach where the API developers annotate changed methods with replacement rules that will be used to update client systems. Henkel and Diwan [42] propose CatchUp!, a tool that uses an IDE to capture and replay refactorings related to the API evolution. Dig *et al.* [29] propose a refactoring-aware version control system for the same purposes.

Dagenais and Robillard observe the framework's evolution to make API change recommendations [26], while Schfer *et al.* observe the client's evolution [79]. Wu *et al.* present a hybrid approach [90] that includes textual similarity.

Nguyen *et al.* [69] propose a tool (LibSync) that uses graph-based techniques to help developers migrate from one framework version to another.

Finally, Holmes and Walker notify developer of external changes in order to focus their attention on these events [44].

Some studies compute rules by comparing two versions of one system. Kim *et al.* [53] automatically infer rules from structural changes. The rules are computed from changes at or above the level of method signatures, *i.e.*, the body of the method is not analyzed. Kim *et al.* [52] propose a tool (LSDiff) to support computing differences between two system versions. In such study, the authors take into account the body of the method to infer rules, improving their previous work [53] where only method signatures were analyzed. Each version is represented with predicates that capture structural differences. Based on the predicates, the tool infers systematic structural differences. In this process, the tool takes as input the client system, a set of systems already migrated to the new framework as well as the old and new version of the framework in focus. Using the learned adaptation patterns, the tool recommends locations and update operations for adapting due to API evolution.

Dig and Johnson [27] help developers to better understand the requirements for migration tools. For example, they have found that 80% of the changes that break client systems are refactorings.

Some studies address the problem of discovering the mapping of APIs between different platforms that separately evolved. For example, Zhong *et al.* [93] target the mapping between Java and C# APIs while Gokhale *et al.* [36] present the mapping between JavaME and Android APIs.

5.4 Research Goals

API developers do not have a thorough understanding of the impact that the deprecation of a method could have on the clients. For instance, they are unsure about whether they are deprecating a popular method or an unpopular one. It is also unknown as to whether clients who use APIs adapt to a change at all, or if they end up not upgrading their API version. Having detailed knowledge of the actual usage of the API and the potential impact of an API change could significantly influence the API's policy with respect to deprecation, helping developers making more informed decisions.

5.4.1 Differences with the study by Robbes *et al.*

This paper is a non-exact replication of our previous study of the Squeak and Pharo ecosystems [76]. We analyzed projects hosted on the Squeaksource platform, that used the Monticello versioning system. The dataset contained 7 years of evolution of more than 2,600 systems, which collectively had over 3,000 contributors. We identified 577 deprecated methods and 186 deprecated classes in this dataset. If its results were very informative, our previous study had several shortcomings that this follow-up study addresses:

Specific dataset. The study was based on a rather specific dataset, the Squeak and Pharo ecosystems found on Squeaksource. In this followup we investigate the same phenomenon on a mainstream toolset (Java projects on GitHub).

Dynamically typed language. Since there is a lack of explicit type information in Smalltalk, there is no way of actually knowing if a specific class is referenced and whether the method invocation found is actually from that referenced class. This does not present an issue when it comes to method invocations on methods that have unique names in the ecosystem. However, in the case of methods that have common names such as `toString` or `name` or `item`, this can lead to some imprecise results. In the previous study, we resorted to manual analysis of the reactions to an API change, but had to discard cases which were too noisy. In this study, Java's static type system addresses this issue without the need for a tedious, and conservative manual analysis.

"Small" dataset. The set of systems we investigated in the previous study is relatively small, compared to other ecosystems. This study investigates the impact of deprecation in a dataset of approximately 25,000 projects (nearly an order of magnitude more than the previous work). Additionally, these 25,000 projects are all clients of at least one of the APIs, which use Maven to manage their dependencies. This allows us to address the two following limitations of the previous study.

Granularity. Due to the limited amount of data in the previous study, we conducted it at the granularity of the API element. This is a shortcoming as different APIs may have different policies regarding deprecation. In this work, we specifically target 5 different APIs, and our main unit of study is the API. This allows us to compare and contrast individual APIs.

Implicit versions. Explicit library dependencies are rarely mentioned in Smalltalk, and there are several ways to specify them, often programmatically and not declaratively. In addition, Smalltalk does not use import statements as Java does. This makes it hard to detect dependencies between projects (heuristics are needed [60]) and to analyze the impact of deprecated methods on client. In contrast, Maven projects specify their dependencies explicitely and declaratively. This allows us to determine which version of the API a project depends on, and hence answer additional questions, such as whether projects freeze their dependencies instead of upgrading.

5.4.2 Research Questions

As this is a partial replication work we try to keep as much as possible the same research questions as the original work. Given our additional information, we also add three research question (marked with "new"), and alter the order and—in some cases—the methodology we use to answer the research questions. This leads to some differences in the formulation and execution of the research questions. The research questions this work investigates are:

- **API Versions.** RQ0a. (new) What API versions do clients use? RQ0b. (new) Do clients upgrade their dependencies?
- **Adaptations.** RQ1. Do all the projects adapt to API changes?
- **Magnitude.** RQ2a. How large are the reactions to API changes? RQ2b. (new) How large are the potential reactions to API changes?
- **Duration.** RQ3a. How long does it take for projects to notice an API change? RQ3b. How long does it take for projects to adapt to an API change?
- **Frequency.** RQ4. How often do deprecated API methods cause ripple effects in the ecosystem?
- **Consistency.** RQ5. Do the projects adapt to an API change in similar ways?
- **Documentation** RQ6. How helpful was the deprecation message, if any?

5.5 Results

In this section we answer the research questions that were detailed in section 5.4.

5.5.1 API Versions

RQ0a. What API versions do clients use?

Guava. There are 3,013 Guava clients on GitHub. We find 18 different major versions of the API, and 39 unique API versions, showing consequent fragmentation. The most popular version used in the latest revision of all the projects is version 14, with 868 usages, or 28% of the clients. This is even though there are 4 more recent releases, accounting for

520 usages (17% of the clients). Further a large proportion of the projects have chosen to stay at older revisions of the API (versions 1 to 13: 1606 clients or 53%). There are 19 unspecified versions (which Maven resolves to the latest version).

Spring. Spring has a total of 15,003 clients. We find 4 major versions, with version 3 having the bulk of the clients (13,480 or 90%). The more recent version 4 has only 972 clients, or 6.5%, while the older versions 1 and 2 have 356 clients or 2.5%. There are 189 unspecified or "latest" versions (1.3%). Drilling down on version 3 we find that both versions 3.1 and 3.2 have more than 5,300 clients each, while version 3.0 has 2,778. We find 75 individual versions.

Hibernate. There are 6,038 Hibernate clients, with 92 individual versions. We only find major versions 3 (2490 clients, 41.2%) and 4 (3,498 clients, 58%), with 50 unspecified or "latest" versions (0.8%). In major version 3, version 3.9 has the majority of usages (1560, more than 60% of version 3 users); in version 4, minor versions 4.1 (1696, 48.5% of version 4 users) and 4.2 (1158, 33%) have a large proportion of usages, despite version 4.3 being the most recent.

Easymock. We find a total 649 clients of Easymock. There are 3 major versions and 16 individual versions. In this case the latest version, version 3, is the most popular (443 clients, 68.2%), with earlier versions accounting for 31.2% of all usages, and 3 unspecified versions (0.5%). In version 3, minor versions 3.0 and 3.1 are the most popular (90% of usages), despite version 3.2 being more recent (10%).

Guice. There are 654 projects that refer to the Google Guice API; they are spread across 4 major versions, and 11 total versions. The most popular version is version 3 with 548 clients (83.8%). Earlier versions have 86 clients (15.7%), while the later version 4 has only 15 clients (2.3%), with 5 unspecified versions (0.1%). Version 2 was released in May 2009, while version 3 was released in 2011. This shows that most projects do not transition to a newer version of the API and are content sticking to an older one.

Takeaway. There is a large number of different versions of the APIs that are used, and a large amount of fragmentation between the versions. Further, the vast majority of projects use older versions of the APIs.

RQ0b. Do clients upgrade their dependencies? Seeing that so many projects depend on older versions of the APIs, we look at whether these projects ever updated their dependencies or if they "froze" their dependencies—that is, if they never updated their API version. If projects update we measure how long they took to do so (time between the release of the new version of the API in Maven central, and when the project's POM file is updated). We count the few projects with blank versions as upgrading since Maven automatically gives the latest version in that case.

Guava. Out of the 3,013 Guava clients, 2,403 projects freeze their version (79.8%). The remaining 610 projects (20.2%) update at least once their API versions. The median client that upgrades does so within 72 days of the release of a new API (a little over two months).

Spring. The situation is the opposite in the case of Spring: only 3,891 projects froze their API version and never update it (25.9%). The remaining 11,112 projects do update their versions (74.1%). It takes 69 days for the median client to upgrade to a newer version after its release.

Hibernate. 3,584 out of 6,038 clients freeze their version (59.3%), while 2454 projects change it (40.6%). The median client takes 62.5 days to upgrade the version.

Easymock. Only 63 projects change their version (9.7%), while the remaining 586 projects (90.3%) froze it. In addition, the clients of Easymock that do upgrade, take a very long time to do so—the median is 272 days or 9 months.

Guice The situation for Guice is even more drastic: 605 of 654 projects freeze their version (92.5%), while only 49 change it (7.5%). Further, the clients of Guice are very slow to move to a new version. The median number of days to upgrade is 909, or 2 and a half years.

Takeaway. From this we can see that version usage and adaptation behavior heavily vary among APIs. If the clients of APIs such as Spring do change their versions for the most part, 80% or more of the clients of 3 other APIs (Guava, Easymock and Guice) do not appear to do so, while 60% of Hibernate clients stay with the same API versions. Update time varies considerably—we will come back to this in RQ3a.

5.5.2 Adaptations to API changes

RQ1. Do all the projects adapt to API changes?

As we have seen, many projects do not adopt new versions of APIs. In this research question, we focus on the projects that use deprecated method invocations, and on whether and how they react to these deprecations. We first count all the projects that use calls marked as deprecated in the API version they use (Affected Projects in Figure 5.1). We also count the number of projects that use deprecated calls in the latest API version separately (Potentially Affected Projects, a superset of Affected Projects). Projects which are definitely not affected by deprecation are shown on top.

We then focus our analysis on the subset of affected projects that update their API version during their lifetime. This is because projects that do not update their API version do not have a strong incentive to fix a deprecation warning, since the method is still functional in their version. Table 5.1 shows how we categorize projects according to their reactions.

Further, we also classify the type of reaction (Table 5.5.2). As projects may react to each deprecation in a particular way, we perform this classification per API element, and then compute for each project the proportion of each category of reactions. We report the median and the interquartile range (IQR) of these proportions on all the projects and include a boxplot as well.

Guava. We find that there is a total of 774 (25.7%) projects which have deprecated method invocations or annotations. Out of these, 245 projects change their API version. A large percentage (161, or 65%) of these projects do react to the deprecations; the rest of the projects never fix any of the deprecated calls, even though they upgraded the API (84 or 34%). In the end, a minority of projects (56, 22%) fix all the deprecation warnings, including the ones they added along the way. If all projects were to upgrade to the latest version of guava, 917 projects would have deprecated calls (30.3%).

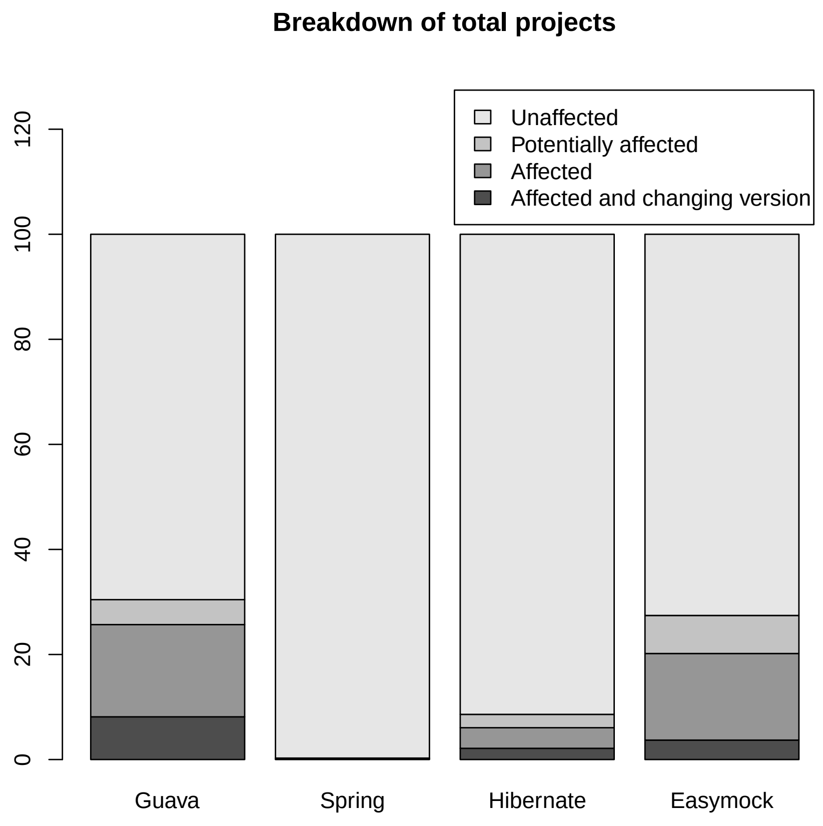

Figure 5.1: Deprecation status of clients of each API

The most common strategy as can be seen in figure 5.2, was to delete the deprecated API elements: 159 out of 161 projects did so at least once. Projects used this strategy often (median 55%; IQR 25—100%). The next most common strategy was to leave API elements, (median 20%, IQR 0—50%). Replacements and Rollbacks were less common: 16 (out of 161) projects fixed some deprecations by rolling back to a previous version of the API (median and IQR 0%), while 42 projects replaced some deprecated API elements with non-deprecated ones. A few outliers replaced the majority or up to the entirety of the deprecations in this way; however, the vast majority of projects did not do so (median: 0%, IQR 0—9%).

Spring. Surprisingly, out of 15,003 clients, only 36 projects contain any deprecated call of some sort (0.24%). Of those, 31 changed their versions, with 10 (32%) reacting to some deprecations by removing them, while the remaining 21 did no such thing (68%). A minority of projects did finally fix all the deprecation warnings they encountered (7, 22.6%). If all projects were to upgrade to the latest version, only 41 would have deprecated

Category	Description
Affected	Projects using a deprecated entity.
Version change	Projects that change API version during their lifetime
Reacting	Projects that remove at least one usage of the deprecated entity
Fixes	Project that removed all deprecated entities in their latest version
Not Reacting	projects not removing any dependency to the deprecated entity
Counter-reacting	Projects adding more usages of the deprecated entities

Table 5.1: Categorization of affected projects

Category	Description
Deletion	Deletion of the deprecated entity from the class of a project; either removed completely or replaced with an ad-hoc solution
Replacement	Deprecated entity is replaced with a non deprecated entity from the API
Rollback Version	Project with deprecated entity rolls back to a previous version where the entity is not yet deprecated
No reaction	The project does not do anything

Table 5.2: Categorization of reactions to API deprecation

calls (0.27%).

In Spring, reactions are split between deleting the entities (median 100%, IQR 81—100%), and leaving them (median: 0%, IQR: 0—19%). There were no replacements or rollbacks. This is reflected in the boxplot in figure 5.3.

Hibernate. 366 of 6,048 projects (6%) use deprecated functionality, out of which 129 changed their version. A minority of these reacted (40 projects, 31%), and a smaller but important minority (25 projects, 19.4%) solved all deprecation issues. There were 89 projects that did not react (69%). If all projects were to update to the latest version of Hibernate, 520

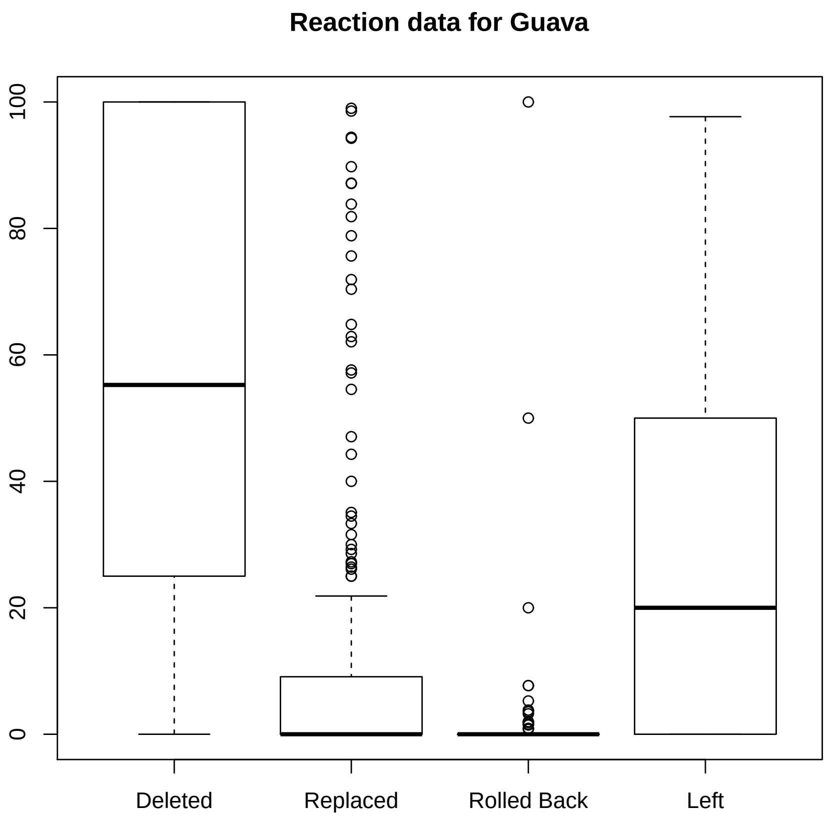

Figure 5.2: Boxplot for each reaction type for Guava

of them would have deprecated calls or annotations to resolve (8.6%).

In Hibernate, all four reactions are present as is seen in figure 5.4, although deletions (median 67%, IQR: 33—100%) and leaving the API elements predominate (median: 0%, IQR (0—45%). Replacements are occasional, concerning 11 projects (median: 0%, IQR: 0—14.5%), and one project used a rollback.

Easymock. Out of the 649 clients of Easymock, 131 use deprecated functionality (20.2%). Of those, 24 changed versions, and a majority of those (17, 71%) reacted to the deprecations, while 4 projects (16.6%) fixed all their deprecation issues. The remaining 7 projects did not react (29 %). Were all projects to upgrade to the latest version of Easymock, 178 projects would use deprecated functionality (27.4%).

In Easymock, the most common adaptation was to delete the API elements as is reflected in figure 5.5, this occurs in all of the reacting projects (median 50%, IQR: 20—75%). Replacements (median: 0, IQR: 0—51%), and leaving the elements (median: 15.5%, IQR 0.8—34%) are occasional. There were no rollbacks.

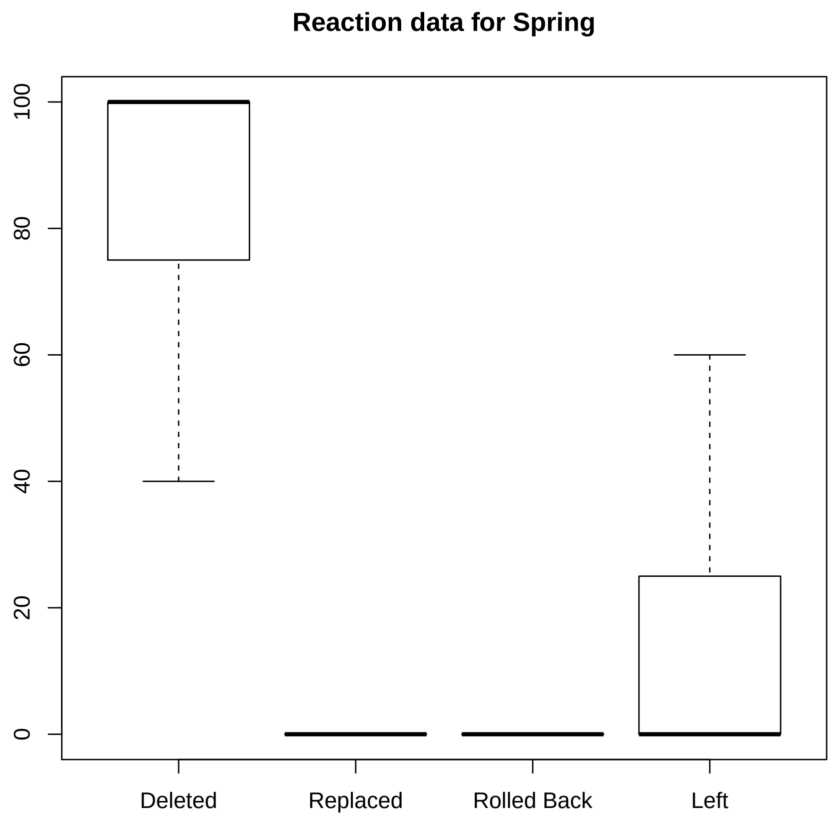

Figure 5.3: Boxplot for each reaction type for Spring

Guice. We analyzed all the Guice projects and looked for usage of a deprecated annotation or method, however we find that none of the projects have used either. This is because of the fact that Guice does not have many methods or annotations that have been deprecated. In fact, Guice follows a very aggressive deprecation policy: methods are removed from the API without being deprecated previously. We observed this behavior in the Pharo ecosystem as well, and studied it separately [45]. In our next research questions, we thus do not analyze Guice, as the deprecations are not explicitly marked.

Counter-reactions. We see that a vast majority of projects (95 to 100%) insert calls to deprecated API elements. This concerns even the ones that end up migrating all their deprecated API elements later on.

Takeaway. A minority of projects are exposed to deprecation: this varies from 20% or more for Easymock and Guava, to less than 10% for Hibernate, and barely any for Spring. On the other hand, a minority of these projects consistently adapts to API deprecations. Most of the projects either adapt partially, do not attempt to adapt at all, or even increase

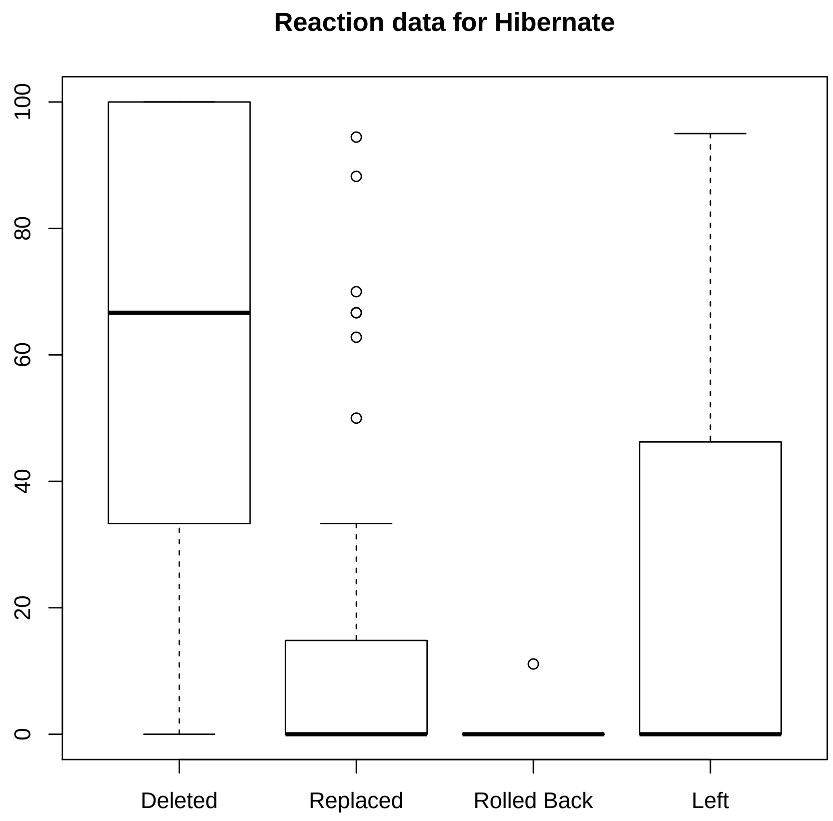

Figure 5.4: Boxplot for each reaction type for Hibernate

their usage of deprecated functionality, regardless of their deprecated status.

5.5.3 Magnitude of the reactions to API changes

Previous work by Robbes *et al.* [76] measured the reactions of individual API changes in terms of commits and developers affected. Having exact API dependency information, in this study we measure API evolution on a per-API basis, rather than per-API element. It is hence more interesting to measure the magnitude of the changes necessary between two API versions in terms of the number of methods calls that need to be updated between two versions. Another measure of the difficulty of the task is the number of different deprecated methods one has to react to: it is easier to adapt to 10 usages of the same deprecated method than it is to react to 10 usages of 10 different deprecated methods.

RQ2a. How large are the reactions to API changes? We start by measuring the magnitude of the actual reactions of projects that do react to API changes. We focus on the upper half of

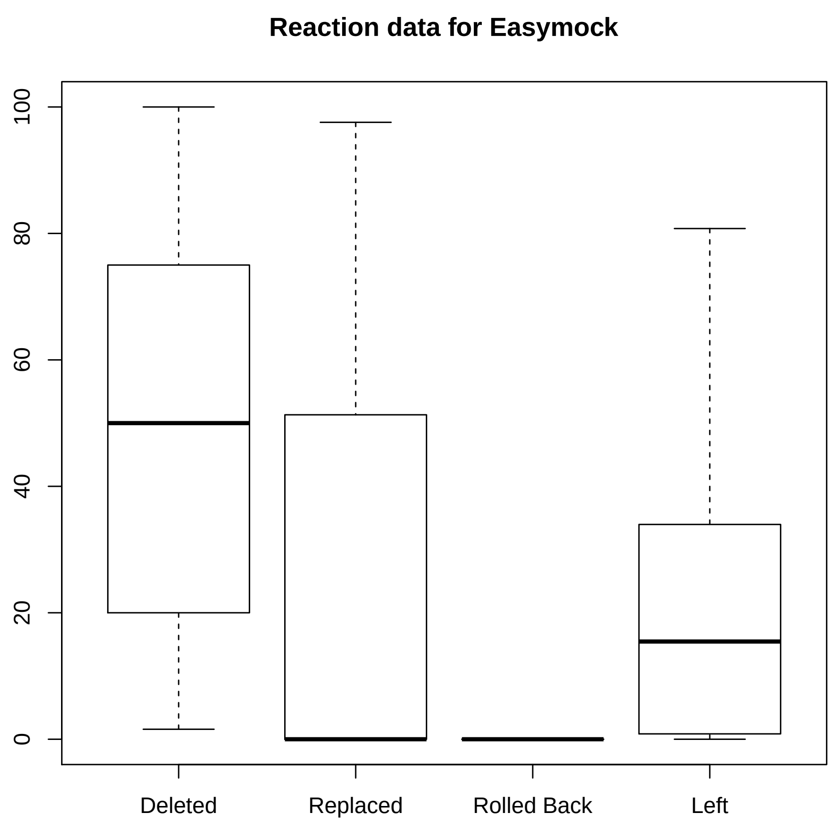

Figure 5.5: Boxplot for each reaction type for Easymock

the distribution (median, upper quartile, 95th percentile, and maximum), in order to assess the critical cases; we expect the effort needed in the bottom half to be low.

Guava. The extent of the adaptations were rather low for the majority of projects: the median number of adaptations was 3, and the upper quartile was 8. Outliers had considerably more work: the 95th percentile was 127, and the maximum 283. In terms of unique methods, we see a median of 1 and a 3rd quartile of 2, pointing again at small adaptations; however, the 95th percentile was 5 unique methods, and the maximum was 10.

Spring. In the few projects that adapt to deprecations in Spring, the effort was larger, with a median of 31 an upper quartile of 54, a 95th percentile of 104 and a maximum of 131. The projects also had to adapt to a variety of deprecated methods (median: 17, upper quartile: 21, 95th percentile and maximum: 27).

Hibernate. Adaptations were also important (median 5, 3rd quartile 20, 95th percentile 41, maximum 59). Unique methods were lower for most projects (median 1), but higher in the upper range (3rd quartile: 16, 95th percentile 27, maximum 40).

Easymock. Finally, Easymock users had also important adaptations (median: 11, 3rd quartile: 21, 95th percentile: 109, maximum: 109), however adaptations were rather systematic (median: 1, 3rd quartile: 2. 95th percentile and maximum: 3).

Takeaway. The magnitude of the adaptation varies: if most projects have little effort, outliers invest more heavily; this may explain the reluctance of some projects to update.

RQ2b. How large are the potential reactions to API changes? Since a large portion of project do not react, we wondered how much work was accumulating should they wish to update their dependencies. We thus counted the number of updates that a project would need to perform in order to migrate their code base to be compliant with the latest version of the API (i.e., removing all deprecation warnings).

Guava. We find that the number of methods to updates varies considerably. Out of the 917 projects that would need changes to be updated to the latest version of Guava, the median number of changes is only 12. However, the upper quartile is 42, the 95th percentile is 319, and the maximum is 8,568 deprecated methods to update. The figures for unique methods are lower, with a median at 1, an upper quartile of 2, a 95th percentile at 7, and a maximum of 44.

Spring. The same numbers for Spring (41 projects) are much lower: the median number of updates is only 3, with the upper quartile at 4, the 95th percentile at 51, and the maximum is 205. In addition, unique methods are very low with both the median and upper quartile at 1, and the 95th percentile at two methods, even if the maximum is 55.

Hibernate. For the 521 projects affected by Hibernate's deprecations, we find that the median would be 15 updates, the upper quartile 35, the 95th percentile 216, and the maximum a whopping 17,471. Most projects use a single deprecated method (75th percentile is 1, 95th is 2; however the maximum is 140 distinct methods).

Easymock. For the 178 projects affected by Easymock's deprecations, we find high values: the median is at 55, the upper quartile at 254 method calls to update, and the 95th percentile at 1,120. The maximum is 4,464. However, the number of unique methods is low (3rd quartile at 1, 95th percentile at 5, and maximum at 7).

Takeaway. If the majority of projects would not need to invest a large effort to upgrade to the latest version, a significant minority of projects, would need to update a large quantities of methods. This can explain their reluctance to do so. However, this situation, if left unchecked—as is the case now—can and does grow out of control. If there is a silver lining, it is that the number of unique methods to update is generally low, hence the adaptations can be systematic. Outliers would run in troubles, with a large number of unique methods to adapt to.

5.5.4 Duration of the reactions

RQ3a. How long does it take for projects to notice an API change?

In this section, we focus on the amount of time developers take to react to an API deprecation once they notice it. We assume developers notice the deprecation upon updating their API version. We quickly saw in RQ0 that projects take a long time before upgrading

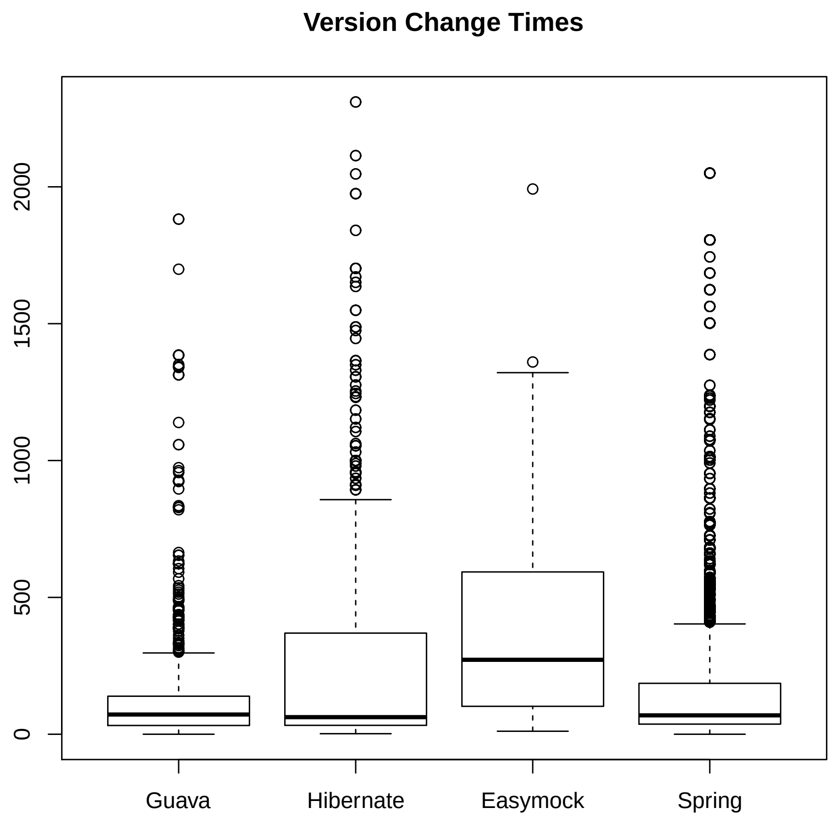

Figure 5.6: Time taken by projects to update their API version

their API dependencies after the API was released. We revisit this here in more details (Figure 5.6).

In the case of Guava, Hibernate, and Spring, we can see that the projects that do upgrade do so quickly: the lower quartile is at one month for these 3 APIs, while the median is at two months. There are still slower clients, especially in the case of Hibernate where the third quartile is at one year. On the other hand, Easymock users upgrade much more slowly, with a first quartile of 2 months, a median of 9 months and a third quartile of nearly two years. In all the cases, we see that outliers can take a very long time to upgrade, in the range of multiple years.

RQ3b. How long does it take for projects to adapt to an API change?

Reaction time is overall much quicker (see Figure 5.7). We notice that in all projects, the median reaction time is low: it is even 0 days for Guava, Hibernate, and Spring, while for Easymock it is 25 days. A reaction time of 0 days means that a deprecated method call start is fixed on the same day that the API version was updated. We see that barring

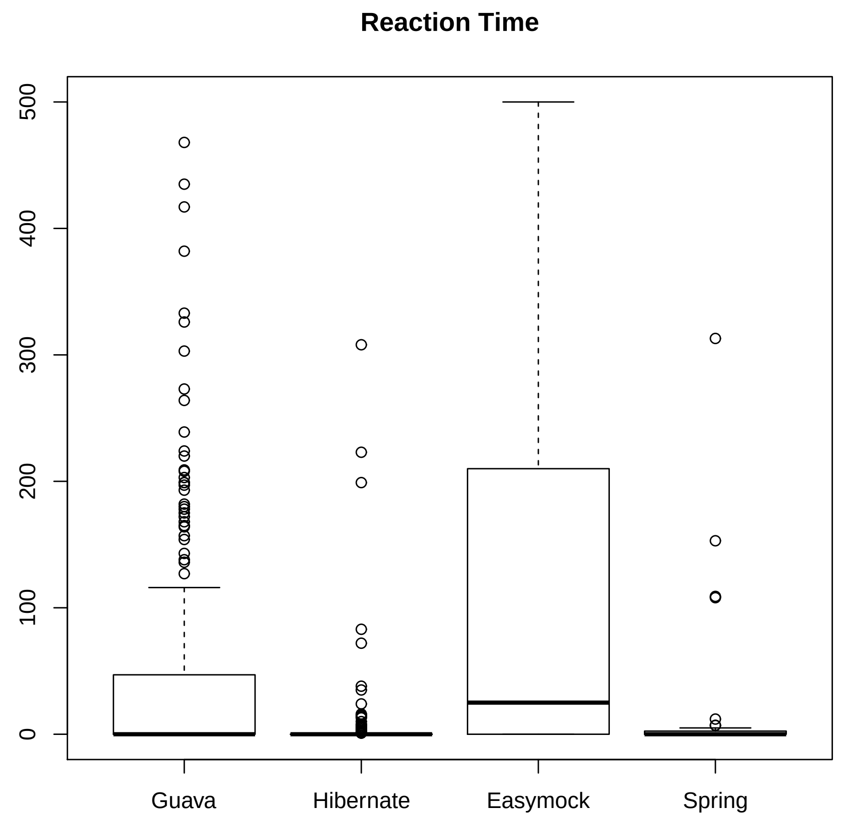

Figure 5.7: Time taken by projects to react to an API deprecation once it is noticed.

outliers, reaction times in Hibernate and Spring are uniformly fast (the third quartiles being at 0 and 2.5 days). Reaction times are however longer for Guava and Easymock, with an upper quartile of 47 for Guava, and 200 days for Easymock.

Outliers have a very long reaction time, up to 500 days for Guava and Easymock, and around 300 days for Hibernate and Spring (despite their much faster reaction times for most projects).

Takeaway. When projects react to API deprecation, they tend to do it quickly, barring outliers. They do however take much longer to notice the deprecation in the first place, as they infrequently update their dependencies.

5.5.5 Frequency of reactions

RQ4. How often do deprecated API methods cause ripple effects in the ecosystem?

We assess how often deprecating a method ends up having an impact on clients in GitHub.

Guava has 1479 methods that have been deprecated. Out of these only 104 (7%) relate to methods that have been used by clients of Guava on GitHub, and for which clients reacted to the deprecation.

Spring deprecated 1320 methods over all its versions. Out of these deprecated entities we find that 149 (11.3%) affect projects on GitHub.

Hibernate deprecated 7591 methods over its lifetime. Out of these deprecated entities we find that 487 affect any clients of Hibernate (6.4%).

Easymock. There are 124 deprecated methods over all the versions of Easymock. Out of these, we find that 16 have an impact (12.9%)

Takeaway. APIs are not shy in deprecating methods, with more than 1,000 deprecations for Guava, Spring, or Hibernate. Fortunately, the proportion of deprecated methods that cause reactions is rather low, hovering slightly above or below 10% in all 4 of the APIs.

5.5.6 Consistency of reactions

RQ5. Do the projects adapt to an API change in similar ways?

The approach by Schfer *et al.* [79] recommends API changes by analyzing the evolution of other API clients to discover evolution rules. The approach was evaluated on Eclipse, JHotDraw, and Struts. We assess whether such an approach would work on the clients of our APIs.

There is no definite way of identifying if a new call made to the API is a replacement for the original deprecated call, so we use a heuristic: We analyze the co-change relationships in each class file across all the projects. If we find a commit where a client removes a usage of a deprecated method `add(String)` and also adds a reference to `add(String, Integer)`, this new method invocation is a possible replacement for the original deprecated entity. We compute the frequencies of these co-change relationships to find whether API clients react uniformly to a deprecation.

In the case of Guava we find 23 API replacements. In 17% of the cases there is a systematic transition *i.e.*, there is only one way in which a deprecated method is replaced by clients that make a transition. The median of the frequency distribution for the highest ranked candidate is 33.3% (IQR: 20%—76.4%).

When reacting, Spring clients mostly delete deprecated entities instead of replacing them. Thus we have no information on whether all the projects react similarly.

For Hibernate, we find only 4 distinct methods where replacements were made. None of them has a frequency of 100%, the maximum frequency is 75%, the others being 22.6, 17, and 16.7%.

Easymock has no systematic transitions either: there are only 3 distinct methods for which the frequency of the co-change relationships is calculated: the highest is 34%, while the other two are 16.6%.

Takeaway. Since API replacements are rather uncommon in our dataset, with the exception of Guava. Thus we find that an approach such as the one of Schfer *et al.* could conceptually work, but in limited cases.

5.5.7 Documentation

Finally, we look at whether API developers have provided documentation that would aid a client in transitioning away from a deprecated entity.

When an API entity is deprecated the API provider should recommend a replacement. The Javadoc tool handles this with the `@deprecated` tag and a developer can optionally provide a link to a replacement.

RQ6. How helpful was the deprecation message, if any?

For this research question, we investigated the deprecation messages. We first determine whether the deprecation message provides a potential replacement for the deprecated API (e.g., "Use Resources.asByteSource(URL) instead."), or whether the method is no longer needed (e.g. "instances of FluentIterable don't need to be converted to FluentIterable "). We also report on additional insights gathered from the exception messages and the APIs.

In the case of Guava, we investigated all 104 deprecated methods that had an impact on clients. For Easymock, we look at all 16 deprecated methods that had impact on clients. For Spring and Hibernate, we inspect the deprecation messages of a sample of methods (100 each) that an impact on the clients. A first pattern that we see is that methods with similar signatures are often deprecated with the same message, leading to repetitions in the deprecation messages.

We find that the overwhelming majority of deprecation messages recommend a replacement. In Guava, 100 out of 105 messages either recommend a replacement (61 messages), or state the method is no longer needed and hence can be safely deleted (39 messages); only 5 deprecated methods do not have a message. In the case of Spring, all the messages provide a replacement (88 messages) or state that the method is no longer needed (12 messages). For Hibernate, all the messages provide a replacement. Finally, for Easymock, 15 of 16 messages provide a replacement; only one does not.

Takeaway. API maintainers of popular APIs make an effort to provide their clients with alternatives when they deprecate an API, even if clients do not always take advantage of this.

We also observed interesting characteristics of each APIs:

Guava is the API with the most diverse deprecation messages. Most messages that state a method is no longer needed are rather cryptic ("no need to use this"). On the other hand, several messages have more precise rationales, such as stating that functionality is being redistributed to other classes. Others provides several alternative recommendations and detailed instructions (e.g., "use MapMaker.softValues() to create a memory-sensitive map, or MapMaker.weakKeys() to create a map that doesn't hold strong references to the keys"), and one method provides as many as four alternatives, although this is because the deprecated method does not have exact equivalents. Guava also specifies in the deprecation message when entities will be removed (e.g., “This method is scheduled for removal in Guava 16.0”, or even “This method is scheduled for deletion in June 2013.”).

Spring is consistent in specifying in which version of the API the methods was deprecated. On the other hand, most of the messages do not specify any rationale for the decision, except JDK version testing methods that are no longer needed since Spring does not run in early JDK versions anymore.

In Hibernate, most deprecation messages do not have a rationale. The only exceptions are messages explaining the advantages of the recommended alternative database connection access compared to the deprecated one.

In Easymock, 15 of the 16 deprecated methods are instance creation methods, whose deprecation message directs the reader to using a Builder pattern instead of these methods. The last deprecation message is the only one with a rationale, and is also the most problematic: the method is incompatible with Java version 7 since it’s more conservative compiler does not accept it; no replacement is given.

5.6 Discussion

5.6.1 Comparison to the Smalltalk studies

Reactions. We find in both studies that many projects do not react to the API deprecations, for a variety of reason. In this study we find that many projects do not update their API version, which we occasionally saw in the SmallTalk study [76]. We find more counter-reactions in this study; this is because a Java deprecation gives a compile-time warning that can be ignored, while in SmallTalk some deprecations may give a run-time error.

Magnitude. We found that some API changes can have a large impact in both studies. In this study we also quantify the potential impact of deprecation if projects would upgrade their API version, and find that in some cases the impact would be very high.

Time to update. In both studies, we find that projects that update take some time to notice the deprecation. Both studies also find that the adaptation time is usually short (same-day), but can occasionally be longer if some obsolete method calls are not discovered right away.

Frequency. Our study of API deprecation in SmallTalk found that a 14% of deprecated methods caused reactions in the ecosystem. This is in line with what we observe in this study (around 10%)

Systematic replacements. The SmallTalk study found that in a large number of cases, there are systematic replacements to a deprecated API element that most projects end up using. We find that this is not the case here, as replacements are not that common. This may be because we analyze a handful of specific APIs; the situation may be different with other APIs.

Deprecation messages. Contrary to the SmallTalk study (where half of the deprecation messages were useful), we find that the vast majority of deprecation messages point towards a replacement. This may be due to the high profile of the APIs we analyze.

Guice. We find that Guice does not use deprecation and instead immediately removes obsolete methods. Our second SmallTalk study [45] found that this behavior also occurs in SmallTalk.

5.6.2 Variance in deprecation marking

We find that every API behaves differently. There are two ways in which a developer can be notified of a feature's deprecation. The first is by using a compiler directive and the second is to mark it in the Javadoc. Javadoc also allows developers to use the `@link` annotation to indicate in the documentation as to which entity replaces the deprecated entity. Both these schemes are generally used in conjunction with each other as one is a compiler directive whereas the other is to inform a developer. In this section we look at the difference in the schemes used by the various APIs that fall under this study.

We observe that Guava, Spring, Hibernate and Easymock all use the compiler directive to mark deprecated entities in their APIs. These APIs are also well documented and thus use the Javadoc technique as well. However, as we have seen earlier there are differences in whether they link to a replacement or not. Despite the presence of both techniques we found some anomalies it comes to Guice and Hibernate.

In the case of Hibernate it appears that they only use the source code annotations starting version 4.0.0 onwards. Thus for all the previous (3.3.0 - 3.6.10), this directive has not been used instead the deprecation was marked only in the Javadoc. This makes it hard for us to detect deprecated entities in earlier versions of Hibernate as our deprecation detection technique depends on the presence on the compiler directive. We have seen that even in the present day version 3.6.0 is the most popular version that is used. The combination of these two fact results in the loss of a lot of data for the purpose of this analysis. This is the main reason as to why we see a small number of projects that are affected by a deprecated entity despite the dataset containing a large number of projects.

Guice is a young API with only three released versions and two beta version for the fourth installment. Due to this there are very few methods and annotations that are deprecated in the first three versions. On inspection of the documentation of each version we found that there was a very small number of deprecated entities, furthermore these deprecated entities do not show up in our dataset. We tried to make the connection between the documented deprecated methods and annotations, but it appears that in the released JARs of the different version of the APIs the class files containing the deprecated entity did not exist. As these classes had been removed, we could not mark the deprecated features in the dataset. Due to this, there was no data to be inferred related to the usage of deprecated features of Guice.

5.6.3 Difference in deprecation strategies

The decisions to deprecate a feature in APIs can be made based on a variety of reasons. One could be that the feature is obsolete and there is a better way to reach he same end goal. Another would be because the feature is rarely used and requires effort to maintain thus it could be marked as deprecated. Or the feature might have a flaw in it so it is replaced by a new feature thus making the old one obsolete. The reasons behind deprecating a feature and the feature itself that is deprecated can have impact on the clients of the APIs as we see in this section.

Out of all the APIs we see that the clients of Guava are the ones that are most affected by deprecation. This is reflected in the number of projects that end up using deprecated entities. Out of the 3,014 clients of guava 774 of them are affected by deprecated entities. This would imply that a lot of features that are popularly used by Guava clients are deprecated by its developers. This is reflected in the fact that almost one tenth of the method invocations that are captured in our dataset are marked as deprecated at the time of usage.

The clients of Easymock are similarly affected as the clients of Guava. Here we see that 131 out of the 649 projects are affected by deprecation. In terms of number of classes and methods that are present in the API, Easymock is the smallest of the APIs that is under consideration. Thus on deprecation of even one feature a large number of clients could be affected by the change.

Despite having information on the deprecation of features Hibernate version 4 onwards, we see that there is a surprisingly large number of projects that are affected by deprecation. This has to do with the fact that there are 3548 projects out of the 6038 projects that use Hibernate that migrate or use version 4. Out of these we see that almost one tenth (336) projects are affected by deprecation. This could be down to the fact among all the APIs studied, Hibernate has by far the largest number of features that are deprecated. The Hibernate developers have deprecated 7591 features in version 4 alone as opposed to Guava and Spring who have deprecated 1,479 and 1,320 entitles during their entire lifetime.

We see that Spring has the lowest number of projects that are affected by deprecation. Also, out of the larger projects such as Guava and Hibernate it has the least number of deprecated features over 37 different releases. This leads to a really small number of clients that are affected by deprecation of features. This could indicate that the developers of Spring are doing a good job when it comes to deprecating features as only a minimal number of clients are affected.

We postulate that less aggressive deprecation policies are probably best. However, we would need to investigate this phenomenon further with the aid of data for a lot more APIs to make the results generalizable.

5.6.4 Additional remarks

The additional information we collect on API versions leads us to the following observations.

If it ain't broke, don't fix it. We were surprised that so many projects did not update their API versions. Those that do often are not in a hurry, as we see for Easymock or Guice. Developers also routinely leave deprecated method calls in their code base despite the warnings, and even often add new calls. This is in spite of all the APIs providing precise instructions on which replacements to use. As such the effort to upgrade to a new version piles up. We think that a qualitative study of the reasons of these choices would be very instructive.

Impact of deprecation messages. We also wonder if the deprecation messages that Guava has, which explicitly state when the method will be remove, could act as a double-edged sword: part of the clients could be incentivized to upgrade quickly, while others may be discouraged and not update the API or roll back. In the case of Easymock, the particular method that has no alternative may be a roadblock to upgrade.

5.6.5 Threats to validity

Our most important threat to validity is that we do not detect deprecation that is only specified by Javadoc tags. We checked each API (especially Spring as it has few deprecations), and find that this is an issue for Hibernate before version 4, but not for the other APIs. As a result, we underestimate the impact of API deprecation for a fraction of Hibernate clients. We considered crawling the online Javadoc of Hibernate to recover these tags, but we found that Javadoc per revision of a version was missing (e.g. version 3.1.9).

We only analyze 5 APIs, and notice significant variation between them. As such the results are not representative of all APIs, and a more comprehensive study could be conducted.

The use of projects from GitHub has several threats, as documented by Kalliamvakou *et al.* [50]. The projects are all open-source, and some may be personal projects where maintenance may not be a priority. The projects may be inactive; however, we only select projects that are marked as “active” in GHTorrent. Also, to overcome any bias that may exist in a GitHub based dataset we could download other major clients (*e.g.*, other APIs) of APIs as specified on their Maven central pages. Other threats are that GitHub projects may be toy projects or not projects at all (still from [50]); we think this is unlikely, as we only select projects that use Maven: these are by definition Java projects, and, by using Maven, show that they adhere to a minimum of software engineering practices.

Finally, we only look at the master branch of the projects. We assume that projects follow the git convention that the master branch is the latest working copy of the code [20]. However, we may be missing reactions to API deprecations that have not yet been merged in the main branch.

In RQ5, we use a heuristic to detect possible API replacements; the heuristic might not work in all cases.

For RQ6, we investigated deprecation messages of API elements that caused reactions. It is possible that deprecation messages of API elements that did not cause reaction would be different (e.g., there may not be a message). We checked this by inspecting the messages of all the deprecated methods of Easymock, and did not see a difference between categories.

5.7 Conclusion

In this chapter we have presented an empirical study on the effect of deprecation of API artifacts on API clients. This non-exact replication of a previous study on Smalltalk projects found a broad agreement in its results, with however a few differences. We described a mechanism by way of which we identified popular APIs and their usages by clients in a large corpus of source code. Using this data, we analyzed the usage of deprecated artifacts over the history of all the clients of 5 popular APIs. Our main findings are:

1. Few projects ever upgrade their API dependencies; most projects pick an API version and use it for the entirety of their lifetime.

2. Of the projects that are affected by API deprecation, a minority reacts. Most keep using the deprecated calls, and even add new calls to deprecated functionality.

3. A minority of projects replaces deprecated entities with a replacement from the API; most choose to delete the entity all together. This is in-spite of there being very good API documentation that should aid the transition.

4. The effort involved to react to deprecations can be large; however, the time to react is usually short on the deprecation is noticed. We see that projects that do not react accumulate technical debt in the form of deprecated API calls, should they need to upgrade their API in the future.

5. We find very different API deprecation policies among APIs; we hypothesize that less aggressive deprecation policies are more successful.

These findings call for further investigation, in particular on the low percentage of projects that update their API versions, and that react to API changes. Qualitative studies on the reasons of this could bring insight on whether and how to address this situation.

Chapter 6

Conclusions and Future Work

This chapter gives an overview of the project's contributions. After this overview, we will present some future avenues. Finally, we conclude this thesis with an overview of the most important takeaways from this thesis.

6.1 Contributions

In this thesis our main contribution is the approach fine-GRAPE. This approach allows for the creation of large API usage datasets. We enumerate a number of steps that can be followed to collect data from large scale open source platforms such as GitHub and Sourceforge. The advantage of targeting such platforms is that there is a large amount of data that is available. Our approach only deals with Java based projects that use a specific build system *i.e.*, maven. However, we do not feel that is a drawback as Java is one of the most popular programming languages and Maven is one of the more popular build tools that is used. The most important aspect of our approach which sets it apart from previous approaches is that we collect only type-checked API usages, which helps keeping the number of false positive API usages to a minimum.

Based on the approach that we have created we create an SQL based database that contains API usages collected for five (Spring, Hibernate, Guava, Guice and Easymock) mainstream Java APIs. This database contains data that has been collected from $20,263$ projects which comprises of a grand total of $1,482,726$ method invocations and $85,098$ annotation usages have been collected. This data forms the base upon which we perform four different analyses.

The first analysis that we perform is to analyze the upgrade behavior of clients using APIs. Typically, APIs release new versions of their APIs that either contain new functionality, or refactor existing features or remove certain functionality or deprecate some features. In the case of SOAP and REST APIs, API developers can force the clients of their APIs to upgrade to a newer version. However, in the case of more traditional APIs we see that this is not the case. There generally appears to be a large lag when it comes to clients upgrading the version of the API that they use. Here we are able to distinguish between two types of APIs. APIs that release frequently seem to encourage their clients to upgrade more frequently. This is also reflected by the fact for these APIs the more popular versions are more recent releases of the API. On the other hand, we see that for APIs that have an infrequent release pattern it appears that clients are far less likely to upgrade to a new version of the API and thus suffer from a much larger lag time.

Our second analysis deals with the percentage usage of an API. The APIs that we have selected in our dataset are all large APIs with the exception of Easymock. They all provide a multitude of features to their clients. It is natural to assume that not a large proportion of the API is typically used by clients of the API. This assumption is confirmed by our investigation into number of features used. We see that in the case of three APIs (Spring, Guava and Easymock) the percentage of the API used is in the 10% to 15% bracket. However, for the Guice and Hibernate API the usage percentage is below 2%. As we dive deeper into the numbers we see that the most popular features being used are also the ones that have been introduced early in the APIs life cycle. This fact may explain why newer features that are introduced by API developers are not adopted by users.

In the third analysis we look at the relation between bugs in APIs and the popular features of the API. We postulate that there should be more bugs/bug reports related to the more popular parts of an API as opposed to the unpopular parts of the API. However, we observe the opposite to be true. We see that most parts of the API are quite well tested, and this includes the popular parts of the API. And we observe that the unpopular parts despite being well tested are more prone to bugs. This is especially surprising as we see a very low rate of usage in these parts. We hypothesize that the bug reports related these parts may be submitted by internal developers or that these features that are unpopular on open source platforms such as GitHub might be more popular with corporate clients whose data is unavailable to us.

Finally, we also look at the impact of deprecation of API artifacts on its clients. The study we perform is a non-exact replication of a previous study done in the world of Smalltalk. Deprecation of an API artifact usually implies that the artifact in question has been marked for removal in the future. We see that there are very few clients that are affected by deprecation as only a few ever migrate to a new version of the API. Out of the ones that are affected, we see that even smaller proportion react to the deprecated entity. There is a large number that counter react *i.e.*, they introduce new calls to a deprecated entity as opposed to using the suggested replacement. We see that out of the clients that react, most of them do so late and they react by deleting the deprecated call as opposed to replacing it with recommended functionality. This is in-spite of their being very good documentation provided by the API developers at the time of Deprecation.

6.2 Future work

Here we propose a number of other research fields wherein the dataset we have created can be helpful in.

First, the evolution of the features of the API can be studied. An analysis of the evolution can give an indication as to what has made the API popular. This can be used to design and carry out studies on understanding what precisely makes ascertain API more popular than other APIs that offer a similar service. Moreover, API evolution information gives an indication as to exactly at what point of time the API became popular, thus it can be studied in coordination with other events occurring to the project.

Second, a large set of API usage examples is a solid base for recommendation systems. One of the most effective ways to learn about an API is by seeing samples [77] of the code in actual use. By having a set of accurate API usages at ones' disposal, this task can be simplified and useful recommendations can be made to the developer; similarly to what has been done, for example, with Stack Overflow posts [71].

6.3 Conclusion

We have presented our approach to mine API usage from OSS platforms. Using fine-GRAPE we created a rich and detailed dataset that allows researchers and developers alike to get insights into trends related to APIs. A conscious attempt has been made to harvest all the API usage accurately. A total of $20,263$ projects and accumulated a grand total of $1,482,726$ method invocations and $85,098$ annotation usages related to five APIs have been mined.

We also presented four analyses that we performed based on the dataset that had been created. And based on the four studies we can draw the following major conclusions:

- We see that there is a tendency for clients of an API to not upgrade to the latest version of the API on a regular basis. This can result is a large lag time for these clients behind the latest version.
- There is a difference between the upgrade behavior of clients for APIs that release frequently versus the APIs that release infrequently. This is something that will require further investigation as it could potentially help API developers decide a release policy.
- Out of all the features that are provided by an API, there is a very small portion that is actually used by clients. We also see that these features are the ones that were originally introduced in the API and not those that have been added since.
- There is no connection between the popularity of an API feature and the bugs in that part of the API. This is a surprising result as it is counter-intuitive.
- Not a lot of API clients appear to care about deprecated API entities in the Java world. Not many notice the deprecation in the first place, and out of the ones that do there are even fewer that react to it.
- We find that there are similarities between the reaction to deprecation between the SmallTalk and the Java ecosystems. This is surprising as SmallTalk is a dynamically typed language and Java is statically typed.

We have proposed a couple of future studies that can be carried out using our current dataset. We also provide indications as to what kind of future work can be performed to the various analyses that we have performed on our dataset. Overall, it is our hope that that the approach and the database that we have created will inspire a lot more work in the field of APIs.

Bibliography

[1] Bugzilla. https://www.bugzilla.org/. accessed on 13 November 2015.

[2] Cobertura. http://cobertura.github.io/cobertura/. accessed on 13 November 2015.

[3] Codase. http://www.codase.com/. accessed on 15 November 2015.

[4] Github issue tracker. https://guides.github.com/features/issues/. accessed on 13 November 2015.

[5] Jcov. https://wiki.openjdk.java.net/display/CodeTools/jcov. accessed on 13 November 2015.

[6] Jira. https://www.atlassian.com/software/jira/. accessed on 13 November 2015.

[7] Tiobe index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. accessed on 16 October 2015.

[8] Sujoy Acharya. *Mastering Unit Testing Using Mockito and JUnit*. Packt Publishing Ltd, 2014.

[9] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[10] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682. ACM, 2006.

[11] Earl T Barr, Christian Bird, Peter C Rigby, Abram Hindle, Daniel M German, and Premkumar Devanbu. Cohesive and isolated development with branches. In *Fundamental Approaches to Software Engineering*, pages 316–331. Springer, 2012.

[12] Gabriele Bavota, Mario Linares-Vasquez, Carlos Eduardo Bernal-Cardenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *Software Engineering, IEEE Transactions on*, 41(4):384–407, 2015.

[13] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 12–23, New York, NY, USA, 2014. ACM.

[14] Tegawende F Bissyande, Daniel Lo, Lingxiao Jiang, Laurent Reveillere, John Klein, and Yves Le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 188–197. IEEE, 2013.

[15] Andrew P Black, Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. *Pharo by example*. Lulu. com, 2010.

[16] Alexander Breckel. Error mining: bug detection through comparison with large code databases. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 175–178. IEEE Press, 2012.

[17] Frederick P Brooks. No silver bullet. *Software state-of-the-art*, pages 14–29, 1975.

[18] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[19] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[20] Scott Chacon. *Pro git*. Apress, 2009.

[21] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for java using free-form queries. In *Fundamental Approaches to Software Engineering*, pages 385–400. Springer, 2009.

[22] Kingsum Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *International Conference on Software Maintenance*, pages 359–368, 1996.

[23] Bradley E Cossette and Robert J Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *International Symposium on the Foundations of Software Engineering*, page 55. ACM, 2012.

[24] Davor Čubranic and Gail C Murphy. Hipikat: Recommending pertinent software development artifacts. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 408–418. IEEE, 2003.

[25] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *ACM Sigplan Notices*, volume 43, pages 313–328. ACM, 2008.

[26] Barthelemy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *International Conference on Software engineering*, pages 481–490, 2008.

[27] Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107, 2006.

[28] Danny Dig and Ralph E. Johnson. The role of refactorings in api evolution. In *ICSM 2005: Proceedings of the 21st International Conference on Software Maintenance*, pages 389–398, 2005.

[29] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 427–436. IEEE, 2007.

[30] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of CoSET 2000*.

[31] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web api fragility: How robust is your mobile application? In *Proceedings of the 2nd International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 12–21. IEEE, 2015.

[32] Szczepan Faber. Mockito: Simpler and better mocking.

[33] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[34] Steven D Fraser, Frederick P Brooks Jr, Martin Fowler, Ricardo Lopez, Aki Namioka, Linda Northrop, David Lorge Parnas, and David Thomas. No silver bullet reloaded: retrospective on essence and accidents of software engineering. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 1026–1030. ACM, 2007.

[35] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

[36] Aniruddha Gokhale, Vinod Ganapathy, and Yogesh Padmanaban. Inferring likely mappings between apis. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 82–91. IEEE, 2013.

[37] Adele J Goldberg. Smalltalk-80: the interactive programming environment. 1984.

[38] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean GHTorrent: Github data on demand. In *Proceedings of MSR 2014*, pages 384–387.

[39] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6,000 Projects: Lightweight Cross-Project Anomaly Detection. *Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10*, pages 119–129, 2010.

[40] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897–910, 2005.

[41] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th International conference on Software engineering*, pages 274–283. ACM, 2005.

[42] Johannes Henkel and Amer Diwan. Catchup!: Capturing and replaying refactorings to support API evolution. In *International Conference on Software Engineering*, pages 274–283, 2005.

[43] Daniel Hoffman and Paul Strooper. Tools and techniques for java api testing. In *Software Engineering Conference, 2000. Proceedings. 2000 Australian*, pages 235–245. IEEE, 2000.

[44] Reid Holmes and Robert J Walker. Customized awareness: recommending relevant external change events. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 465–474. ACM, 2010.

[45] Andre Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Marco Tlio Valente. How do developers react to api evolution? the pharo ecosystem case. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, page in press, 2015.

[46] Jeffrey N Johnson and Paul F Dubois. Issue tracking. *Computing in Science and Engineering*, 5(6):71–77, 2003.

[47] Ralph E Johnson and Brian Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988.

[48] Natalia Juristo Juzgado and Sira Vegas. The role of non-exact replications in software engineering experiments. *Empirical Software Engineering*, 16(3):295–324, 2011.

[49] Tomek Kaczanowski. *Practical Unit Testing with JUnit and Mockito*. Tomasz Kaczanowski, 2013.

[50] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101. ACM, 2014.

[51] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160. ACM, 2011.

[52] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 309–319. IEEE Computer Society, 2009.

[53] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. *ICSE*, 7:333–343, 2007.

[54] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196, 2008.

[55] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 306–315. ACM, 2005.

[56] Adrian Lienhard and Lukas Renggli. Squeaksource-smart monticello repository. *European Smalltalk User Group Innovation Technology Award, August*, 2005.

[57] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 477–487. ACM, 2013.

[58] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. How do api changes trigger stack overflow discussions? a study on the android sdk. In *proceedings of the 22nd International Conference on Program Comprehension*, pages 83–94. ACM, 2014.

[59] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 296–305. ACM, 2005.

[60] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *ASE'10: Proceedings of the 25th IEEE/ACM international conference on Automated Software Engineering*, ASE '10, pages 309–312, 2010.

[61] David Mandelin and Doug Kimelman. Jungloid Mining : Helping to Navigate the API Jungle . *Synthesis*, pages 48–61, 2006.

[62] Vincent Massol and Ted Husted. *Junit in action*. Manning Publications Co., 2003.

[63] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of API stability and adoption in the android ecosystem. In *International Conference on Software Maintenance*, pages 70–79. IEEE, 2013.

[64] Tyler McDonnell, Bonnie Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *29th IEEE International Conference on Software Maintenance*, ICSM 2013, pages 70–79. IEEE, 2013.

[65] Amir Michail. Data mining library reuse patterns in user-selected applications. In *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, pages 24–33. IEEE, 1999.

[66] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. Mining api popularity. In *Testing–Practice and Research Techniques*, pages 173–180. Springer, 2010.

[67] Martin Monperrus, Marcel Bruch, and Mira Mezini. Detecting missing method calls in object-oriented software. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6183 LNCS:2–25, 2010.

[68] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can i use this method? In *Proceedings of the 37th International conference on Software engineering*, page in press, 2015.

[69] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to API usage adaptation. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 302–321, 2010.

[70] David Parsons. Unit testing with junit. In *Foundational Java*, pages 219–244. Springer, 2012.

[71] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: Stack overflow in the ide. In *Proceedings of ICSE 2013 (35th International Conference on Software Engineering, Tool Demo Track)*, pages 1295–1298. IEEE CS Press, 2013.

[72] Michael Pradel and Thomas R Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 288–298. IEEE, 2012.

[73] Ophir Primat. Github's 10,000 most popular java projects – here are the top libraries they use. http://blog.takipi.com/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/, nov 2013.

[74] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 378–387. IEEE, 2012.

[75] Steven P Reiss. Semantics-based code search. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 243–253. IEEE, 2009.

[76] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 56. ACM, 2012.

[77] Martin P Robillard and Robert DeLine. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[78] Anand Sawant and Alberto Bacchelli. API Usage Databases. `http://dx.doi.org/10.6084/m9.figshare.1320591`, 2015.

[79] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *International Conference on Software engineering*, pages 471–480, 2008.

[80] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[81] Daniela Steidl and Nils Göde. Feature-based detection of bugs in clones. In *Proceedings of the 7th International Workshop on Software Clones*, pages 76–82. IEEE Press, 2013.

[82] Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, Brad Myers, et al. Improving api documentation using api usage information. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, pages 119–126. IEEE, 2009.

[83] Suresh Thummalapenta and Tao Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. *ASE 2008 - 23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings*, pages 327–336, 2008.

[84] Suresh Thummalapenta and Tao Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 283–294. IEEE Computer Society, 2009.

[85] Suresh Thummalapenta and Tao Xie. Mining Exception-Handling Rules as Conditional Association Rules. *International Conference on Software Engineering (ICSE)*, 2009.

[86] Lars Vogel. Eclipse JDT-Abstract Syntax Tree (AST) and the java model-tutorial. http://www.vogella.com/tutorials/EclipseJDT/article.html.

[87] Shaohua Wang, Iman Keivanloo, and Ying Zou. How do developers react to restful api evolution? In *Service-Oriented Computing*, pages 245–259. Springer, 2014.

[88] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 07:35–44, 2007.

[89] Chadd Williams and Jaime Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 32–36. ACM, 2008.

[90] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 325–334. ACM, 2010.

[91] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57. ACM, 2006.

[92] George Udny Yule. *An introduction to the theory of statistics*. C. Griffin, limited, 1919.

[93] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining api mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 195–204. ACM, 2010.