Origin, Impact and Cost of Interface Instability

PROEFSCHRIFT

...
Dit proefschrift is goedgekeurd door de
promotor: Prof. dr. A. van Deursen
promotor: Prof. dr. ir. J.M.W. Visser

Samenstelling promotiecommissie:

Rector Magnificus voorzitter
Prof. dr. A. van Deursen Technische Universiteit Delft, promotor
Prof. dr. ir. J.M.W. Visser Radboud Universiteit Nijmegen, promotor

Onafhankelijke leden:

Prof. D. German University of Victoria, Canada
Prof. P. T. Devanbu U.C. Davis, United States
Prof. dr. M. G. J. van den Brand Technische Universiteit Eindhoven
Prof. dr. ir. G. Jongbloed Technische Universiteit Delft
Prof. dr. ir R. van Solingen Technische Universiteit Delft

Copyright © 2015 by S. B. A. Raemaekers

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without the prior permission of the author.

ISBN 978-94-6295-304-8

Typeset by the author with the \LaTeX Documentation System.
Printed by Proefschriftmaken.nl || Uitgeverij BOXPress.
Cover design by Steven Raemaekers.
Author email: stevenraemaekers@gmail.com
To my family
When I started this thesis, I did not know that my mother would not be around to witness my graduation. Without the support of my parents, I would not have been where I am today. My mother has taught me to persevere even when things become difficult, which is something that will be with me the rest of my life.

I would like to thank my wife for listening to my new insights and epiphanies, even though they are hard enough to understand for someone with the appropriate background in software engineering. It is good to have somebody at my side which knows how to relativate things.

I would like to thank Soerin Bipat at the Software Improvement Group for his time and motivation to listen to my endless stories about my next big research idea. I would also like to thank Lodewijk Bergmans at the Software Improvement Group for his valuable ideas and feedback on software cost estimation using compression. Lodewijk, Soerin and me proved to be a great team and our cooperation turned out to be extremely productive. The ideas about compression as software cost estimation method were conceived in brainstorm sessions with Lodewijk and Soerin, and for these moments of insight I am very grateful. I would also like to thank Eric Bouwers for his valuable feedback on my papers. Eric has spent a lot of time proofreading my drafts and this thesis has become better because of his comments.

I would also like to thank Julius Davies at the University of British Columbia for providing me a clone of the Maven repository, which became the basis for this thesis.

I would like to thank Joost Visser at the Software Improvement Group for the opportunity to combine my research with a job at a commercial company, which provided a unique opportunity to gain invaluable skills and to experience commercial practice up close. Due to the unique combination of an academic and commercial perspective, this thesis hopes to bridge the gap between research and practice, although I now know from experience that this can be incredibly hard to do. Nonetheless, I believe the most valuable insights are created in an environment which has the right conditions to perform fundamental research, but at the same time provides ample incentives to deliver practical results.

Finally, I would like to thank Arie van Deursen for his patience and time to read my papers, critically listen to my ideas and provide me valuable feedback. As I learned a lot about software engineering from him, I hope that I was able to return some knowledge to Arie as well. Without him, this thesis would not have existed.

Steven Raemaekers
_Amsterdam, November 2015_
# CONTENTS

## 1 Introduction

1.1 Motivation ............................................. 1
1.2 Research Background ................................. 4
1.3 Research Challenges ................................. 5
1.4 Research Method ..................................... 7
1.5 Thesis Outline ...................................... 8
1.6 Origin of Chapters ................................. 9

## 2 The Maven Dependency Dataset

2.1 Introduction .......................................... 11
2.2 Download Location ................................... 12
2.3 Descriptive Statistics ............................... 13
2.4 Data Schemas ........................................ 13
  2.4.1 MySQL Database................................... 14
  2.4.2 Berkeley DB Database ........................... 15
  2.4.3 Neo4j Database .................................. 15
2.5 Methodology .......................................... 16
2.6 Limitations ........................................... 18
  2.6.1 Skipped Libraries ................................ 18
  2.6.2 Package Prefixes ................................. 19
  2.6.3 Usage Frequencies ............................... 19
  2.6.4 Wrong Snapshot Identifiers .................... 19
2.7 Conclusion ............................................ 19

## 3 Semantic Versioning in Practice

3.1 Introduction .......................................... 21
3.2 Related Work ......................................... 23
3.3 Research Questions ................................. 25
3.4 Method ................................................ 25
  3.4.1 Determining breaking changes .................. 25
  3.4.2 Determining subsequent versions and update types 26
  3.4.3 Detecting changed functionality ................ 26
  3.4.4 Obtaining release intervals and dependencies .. 27
  3.4.5 Obtaining deprecation patterns ................ 28
3.5 Application of Semantic Versioning ............... 28
  3.5.1 Version string patterns ......................... 28
  3.5.2 Breaking and non-breaking changes ............ 29
  3.5.3 Major vs minor vs patch Releases ............ 29
  3.5.4 Median analysis ................................ 31
3.5.5 Breaking changes and errors .............................. 32
3.5.6 The relationship between breaking changes and errors .... 33
3.5.7 Average amount of work with and without breaking changes ... 33

3.6 Semantic Versioning Adherence Over Time ............... 35

3.7 Update Behavior ............................................. 35

3.8 Deprecation Patterns ........................................ 38

3.9 Discussion .................................................... 40
  3.9.1 Signaling interface instability ............................ 40
  3.9.2 Other versioning standards ............................... 41
  3.9.3 Actual usage frequencies ................................. 41
  3.9.4 Release interval and edit script size ................... 42
  3.9.5 Major version 0 releases ............................... 42

3.10 Threats to Validity .......................................... 42
  3.10.1 Release dates ............................................ 42
  3.10.2 Version strings ........................................... 42
  3.10.3 Deprecation tags ........................................ 43
  3.10.4 External validity and generalizability ................. 43

3.11 Conclusion .................................................... 44

4 Measuring Interface Instability ............................... 45
  4.1 Introduction .................................................. 45
  4.2 Motivating example ......................................... 47
  4.3 Problem statement .......................................... 48
  4.4 Related Work ................................................ 49
  4.5 Dataset ....................................................... 49
  4.6 Analyzing Maven Dependencies ............................. 50
  4.7 Metric Ingredients .......................................... 54
    4.7.1 Candidate Variables ................................... 54
    4.7.2 Weighting Measurements Historically ............... 55
    4.7.3 Weighting Measurements via API Usage ............. 55
  4.8 Metric Definitions .......................................... 56
  4.9 Apache Commons Findings ................................ 58
    4.9.1 Apache Commons API Usage ........................... 58
    4.9.2 Metrics ................................................... 59
  4.10 Scenarios .................................................... 60
  4.11 Discussion .................................................. 62
    4.11.1 Used metrics .......................................... 62
    4.11.2 Using historical data ................................. 62
    4.11.3 Estimating rework effort ............................. 63
    4.11.4 Transitive calls and dependencies .................... 63
  4.12 Threats to validity ........................................ 63
    4.12.1 Internal validity ...................................... 63
    4.12.2 Construct validity .................................... 63
    4.12.3 External validity ..................................... 64
  4.13 Conclusion .................................................. 64
5 Impact of Interface Instability 65
  5.1 Introduction .............................................. 65
  5.2 Problem Statement ....................................... 66
    5.2.1 Illustrative example ................................ 66
    5.2.2 Research questions .................................. 67
  5.3 Conceptual Framework and Methodology .................. 67
  5.4 Implementation .......................................... 69
  5.5 Descriptive Statistics ................................... 70
  5.6 Individual Library Results ................................ 71
    5.6.1 The relationship between library properties ........ 71
    5.6.2 Regression results .................................... 72
  5.7 Library Interdependency Modeling Results ............... 74
    5.7.1 Estimating the encapsulation effect ................ 74
    5.7.2 Current encapsulation practice ...................... 75
    5.7.3 Multilevel model for interface instability .......... 75
    5.7.4 Model specification ................................... 76
    5.7.5 Multilevel model explanation ......................... 78
  5.8 Discussion .............................................. 80
    5.8.1 Inferring actionable advice from statistical analysis 80
    5.8.2 Relationship with aspect-oriented programming .... 80
    5.8.3 Model misspecification risk ........................... 81
    5.8.4 Testing multiple hypotheses ......................... 81
  5.9 Threats to Validity ...................................... 82
    5.9.1 Internal validity ...................................... 82
    5.9.2 External validity ...................................... 82
  5.10 Related Work ........................................... 82
  5.11 Conclusion ............................................. 83

6 Change Injection 85
  6.1 Introduction ............................................. 85
  6.2 Problem Statement ....................................... 87
  6.3 Change Injection Algorithm ................................ 88
  6.4 Implementation .......................................... 89
  6.5 Breaking Changes and Errors ............................. 91
    6.5.1 The relationship between breaking changes and errors 92
    6.5.2 Libraries with large impact .......................... 93
  6.6 Breaking Changes and Edit Script Size ................... 94
    6.6.1 Explanation ........................................... 94
    6.6.2 ChangeDistiller ....................................... 96
    6.6.3 Regression result ...................................... 96
  6.7 Dispersion Estimation .................................... 97
    6.7.1 Explanation ........................................... 97
    6.7.2 Explanation of independent variables ............... 98
    6.7.3 Regression results .................................... 99
6.8 Discussion ................................................................. 99
6.8.1 Edit scripts as rework measure ................................. 99
6.8.2 Semantic versioning .............................................. 100
6.8.3 Other applications ................................................ 100
6.9 Threats to Validity .................................................... 100
6.9.1 Error counting ....................................................... 100
6.9.2 Transitive closure of upgrade pairs ......................... 101
6.9.3 Selection of independent variables ......................... 101
6.9.4 Small $R^2$ .......................................................... 101
6.9.5 Generalizability .................................................... 102
6.10 Related Work .......................................................... 102
6.11 Conclusion .............................................................. 103

7 Calculating Software Productivity With Compression 105
7.1 Introduction .............................................................. 105
7.2 Function Point-Based Methods .................................. 107
7.2.1 Example ............................................................... 110
7.3 Compression-Based Method ....................................... 110
7.3.1 Kolmogorov complexity ....................................... 110
7.3.2 Formulas ............................................................. 111
7.3.3 Estimating team size ............................................. 112
7.3.4 Example ............................................................... 113
7.4 Data Collection .......................................................... 114
7.5 Results ................................................................. 115
7.5.1 Compression ratios .............................................. 115
7.5.2 Benchmark churn ............................................... 117
7.5.3 Example scenarios .............................................. 117
7.6 Discussion ............................................................... 119
7.6.1 Estimation in practice ......................................... 120
7.6.2 Issues with benchmark approach ......................... 121
7.6.3 Calculation of compression ratio ......................... 122
7.6.4 Validation ............................................................ 122
7.6.5 Real-world case studies ..................................... 122
7.7 Related work ............................................................ 122
7.8 Conclusion ............................................................... 123

8 Software Growth Speed 125
8.1 Introduction .............................................................. 125
8.2 Dataset .................................................................. 126
8.3 Software Growth ..................................................... 126
8.3.1 Absolute Growth per day .................................... 126
8.3.2 Percentual growth per year ................................. 129
8.4 Discussion ............................................................... 130
8.5 Conclusion ............................................................... 131
9 A Software Growth Model 133
  9.1 Introduction ........................................... 133
  9.2 Theory and Hypotheses ................................. 135
  9.3 Dataset and Data Collection ............................ 137
  9.4 Growth Types ........................................... 138
    9.4.1 Kolmogorov complexity ............................ 139
    9.4.2 Compression ratio .................................. 139
    9.4.3 Normalized Compression Distance .................. 140
    9.4.4 Other applications of compression ................ 141
  9.5 Conceptual Model ....................................... 142
    9.5.1 Relationships ....................................... 142
  9.6 Measurement Variables .................................. 143
    9.6.1 Maintainability rating ............................. 143
    9.6.2 Duplication ......................................... 143
    9.6.3 Average cyclomatic complexity .................... 144
    9.6.4 Work done .......................................... 145
    9.6.5 Number of new methods and ΔLOC .................. 145
  9.7 Summary Statistics ..................................... 145
    9.7.1 Correlations ........................................ 146
  9.8 Individual Relationships ............................... 146
    9.8.1 System size and system growth ..................... 146
    9.8.2 System size and work done ......................... 149
    9.8.3 Maintainability and work done ........................ 150
    9.8.4 Maintainability and NCD ............................ 151
    9.8.5 NCD and desirable growth .......................... 152
    9.8.6 Compression ratio and undesirable growth ........ 154
  9.9 Factor Analysis ......................................... 154
  9.10 Discussion ............................................ 156
    9.10.1 Impact for practitioners ........................... 157
  9.11 Threats to Validity .................................... 158
  9.12 Conclusion ............................................ 158

10 Conclusion .................................................. 161
  10.1 Contributions .......................................... 161
  10.2 Answers to Research Questions ........................ 162
  10.3 Impact and validity .................................... 165
  10.4 Lessons for Practitioners .............................. 167
  10.5 Future Work ........................................... 168
    10.5.1 Developer behavior ................................ 168
    10.5.2 Change injection ................................... 168
    10.5.3 Applying compression ............................... 168
1.1. **Motivation**

Third-party libraries are used frequently in modern software development. There is a large chance that the websites that you visit on a daily basis make use of such libraries. These libraries execute a large number of tasks in the background of the system that most people take for granted: an e-commerce website may use open-source logging software or an open-source database system. The page may be created using an open-source content management system. The connection with the bank to handle transactions may be secured using a secure communications library.

Without these libraries, all of these common tasks would have to be reimplemented every time, which would make any software project a daunting endeavour. Imagine that each time a new website would be developed, people would have to recreate a logging framework, a database system, a content management system and a secure communications library from scratch! Instead, there are free libraries available that provide this functionality and do their job in silence and do not expect to receive any credit for doing so. Most website visitors and software users are unaware that they typically use tens of software libraries on every click of a button. These software libraries are the topic of this thesis.

The popularity of open-source software libraries is due to a number of reasons. First, a decreased total development effort is expected since code does not have to be created from scratch. Since other developers have already invented the best way to implement secure communications, for instance, a lot of time can be saved by reusing existing code. Furthermore, specific knowledge and expertise of library developers can be employed, which may not normally be available within a certain company or institution. There is no need to hire external experts: simply include a third-party library which implements the desired functionality and the job is done.

\[1\text{In this thesis, we use the terms “third-party library”, “open-source library” and “free library” interchangeably, since the investigated libraries are all third-party, open source and free.}\]
Another advantage of third-party libraries is that they often have been tested and have been used extensively by other software development teams, which means that the evolution of these libraries has been guided by direct feedback from a developer community. It can be expected that the more actively software libraries are used in practice, the faster a library will evolve and the sooner bugs will be fixed by the community.

Finally, and perhaps most importantly, open-source software libraries are very often free to use. Sometimes, proprietary libraries are also available to perform the same tasks. But it is hard to compete with free.

However, despite all these advantages, the usage of third-party libraries also creates a number of risks to the users of these libraries. For instance, including third-party libraries can expose a software system to security weaknesses present in these libraries. A good example is the Heartbleed bug\textsuperscript{23}, which was recently found in a popular secure communications library called OpenSSL. Despite the fact that several experts with a background in security protocols worked on this library for years and that the library is used by millions of users on a daily basis, the Heartbleed bug was found only recently. Imagine that a banking website, on which millions of people log in daily to take care of their finances, uses a library with such a severe security leak. Passwords and money could have been stolen without leaving a trace.

Figure 1.1: As the Trojans have learned long ago, incorporating externally created artifacts carries a certain risk. Painting: “The Procession of the Trojan Horse in Troy” by Giovanni Domenico Tiepolo, 1760.

An active community behind a software library also does not guarantee that specific desirable features will be included in its next release. For instance, assume that an e-commerce website uses a database library which lacks a certain query feature. Unless the developers of the website implement the feature themselves in the library, they are dependent on the developers of the library to implement the feature. A disadvantage of open-source software development is that it can take a lot of time before a certain feature gets implemented, because developers working on these projects often have

\textsuperscript{2}http://en.wikipedia.org/wiki/Heartbleed
\textsuperscript{3}http://www.heartbleed.com
daytime jobs and they develop a library as a hobby. But even when the feature can be implemented immediately, this still does not guarantee that library developers are willing to include it. The library developers may need to be convinced that the proposed contribution benefits the community as a whole.

The evolution of a software library does not only constitute new functionality, but also changes in existing functionality. It is a fact of daily software engineering practice that software requirements keep changing, and so must software if it does not want to become obsolete. This can manifest itself in changes in the implementation of a library, such as bug fixes or performance improvements. These changes are relatively innocent because they are not expected to change the external behavior of the system in a radical way.

However, when changes are required that change the interface or implementation of a library in a significant way, library developers have to make a trade-off. On the one hand, they want to update their software to include new features and changes to prevent it from becoming obsolete, but on the other hand, they are faced with the consumers
of their interfaces which expect these interfaces to be constant. In theory, interfaces should be designed properly at the start of a new library, and should be kept constant for the entire lifetime of the library.

In practice, however, the interface and implementation of a library are both dynamic and keep changing to meet the changing needs of the end users of the library. This often means that a new version of a library is released which is backward incompatible with its previous version, and library users have to update their code to use the newer version of the library.

Going further than visible changes in library interfaces are changes in the functionality that a library provides. These changes may not lead directly to problems in client systems using them, but can nevertheless have a large impact on these systems. It may be relatively straightforward to adapt a system to use a changed interface, but functional changes can be more subtle and harder to detect.

Nonetheless, libraries and software systems in general should be capable of evolving in such way that they can keep up with changing requirements. This ability depends to a large degree on the flexibility and quality of the design of a software system. Some software systems are better suited to adapt to changing requirements than others. Eventually, a software system should be able to achieve sustainable long-term growth, which means that changes to a system should be made in such way that they do not degrade existing architecture or make changing the system harder in the future. This will lead to lower maintenance costs in the long run.

1.2. Research Background

Interface instability and its impact on developers using that interface has received research attention from several different researchers. Until now, the research in the area of interface instability has focused mainly on two areas. First, change-proneness of classes and interfaces and properties of code which tend to change frequently has been investigated extensively using a wide range of methods. For instance, Arisholm et al.[8] investigate dynamic coupling metrics and the relationship with change-prone classes. Tsantalis et al.[107] aim to determine probabilistically whether a class will need to be changed in a given object-oriented design. Often, this type of research is not concerned with changing interfaces per-se but rather with changes in an entire class or component. In this thesis, we are specifically interested in changes in interfaces and the impact these changes have on client systems.

The actual degree to which a real-world dataset contains backward-incompatible changes has only been investigated anecdotally in literature[29, 35, 38, 78, 104]. This thesis tries to fills the gap between academic research and practice in the sense that it investigates a real-world engineering problem with a structural empirical research approach using a large dataset of software libraries that are used on a daily basis in a large number of software projects.

Another area of active research is the automatic refactoring of client systems in order to update to a newer version of a class or interface. An example is an algorithm that finds out how code was adapted after a method has been removed from an interface[31]. These updates can then automatically be applied when updating to a new version of a library [11, 30, 31, 36, 54, 63, 116]. In contrast to the research performed in this area,
our work is more of an empirical nature and does not have the goal to suggest automatic refatorings to developers or to create a tool that can do this. Instead, the objective of our research is to investigate and understand real-world developer behavior by mining a repository of software libraries. The results of our investigation can be used by developers to gain an understanding of their own development practices and to deepen their insight into the underlying processes. With these insights, software can be built that is of higher quality and is cheaper to maintain in the long run.

This thesis is not the first to use the Maven repository in a research context. Two other works that also use the same repository as this thesis and also use a similar empirical approach are [34] and [85].

In the area of software cost and productivity estimations, many different methods and calculation models have been suggested [62]. These models often contain a benchmark to which the productivity and costs of a software project can be compared. Often, these methods are based on the size of a project as measured by the number of lines of code or number of function points it contains. Methods such as COCOMO [19] and the benchmark of Capers Jones are used extensively in practice to estimate costs and the rebuild value of software. We provide an alternative to these estimation methods which does not rely on counting function points or lines of code and can be applied automatically on source code.

In this thesis we use compression as a method to determine the quality of software and to determine the amount of functional growth of a software system between two versions. Although our method can be used to calculate the cost of implementing any amount of functionality similar to other cost estimation methods, this is not our direct goal. Instead, we use compression to quantify the amount of “desirable” and “undesirable” growth in software and we use it to investigate the underlying processes that ultimately cause interface instability. Compression has not been applied for this goal in software engineering research before, although it has been applied to a wide variety of fields such as DNA analysis[74] and music comparison[112].

1.3. Research Challenges

The main goal of this thesis is to investigate the origin, impact, and cost of interface instability in software engineering practice. We start by investigating the stability of software interfaces in practice by investigating backward incompatible changes in software libraries. The result of this investigation will provide a basis for later chapters. The software community itself has created guidelines and rules for updating library versions and signaling to library users whether a new library version contains backward incompatible changes. These rules provide an opportunity to test the degree to which backward compatibility is taken into account by the open-source community. The goal of our investigation is not to obtain a way to measure interface instability, but to investigate current practice. The answer to the second research question will provide a way to quantify our findings. The first research question is thus as follows:

RQ1: How stable are software interfaces in practice?

http://www.namcookanalytics.com
After we have obtained an understanding of the stability of interfaces in practice, we want to quantify interface and implementation instability over multiple versions of a library. Since the metrics used to answer the first research question are not detailed enough for our later analyses, we define our own set of metrics. These metrics provide a single number (a “rating”) of a library and shows how a library scores in terms of the number of removed methods from public interfaces, the amount of change in existing methods, the amount of new methods added to each release and the ratio of change in old and new methods. Together, these metrics provide an overview of the interface stability and implementation effort of a library during its release history. The second research question is thus as follows:

**RQ2: How can we measure interface instability?**

We have now determined the stability of software interfaces in practice and we have defined metrics to measure this instability. Next, we are interested in the actual impact of interface instability. This impact manifests itself through *ripple effects*, which is code that needs to be updated because of changes in other code, in this case interfaces. We use our previously defined metrics to measure the amount of ripple effects coming from changed interfaces. Additionally, we also measure the impact of interface instability by automatically injecting changes in interfaces in the old version of a library to find out how many compilation errors need to be fixed to apply this update. The third research question is as follows:

**RQ3: What is the impact of interface instability?**

After we have determined the impact of interface instability in terms of ripple effects and compilation errors, we also investigate what can be done to mitigate these ripple effects. In particular, how effective is *encapsulation*, a widely known and applied software design principle, to mitigate these effects? The next research question is therefore as follows:

**RQ4: How can interface instability be mitigated?**

To translate our findings to practice, we introduce a method that enables to estimate the amount of time or money it takes to update software after an interface has changed. Although the amount of compilation errors and the metrics we have introduced can give an overview of the amount of expected changes, an amount of time or a monetary amount are easier to translate to business practice than our other metrics. The use of compression enables us to calculate the functional size of a change without relying on the number of lines of code of that change. This approach also makes it possible to compare software written in different programming languages. This leads us to the fifth research question:

**RQ5: How can we calculate the costs of interface instability?**
Ultimately, changes in interfaces are just one manifestation of changing software systems, which are themselves manifestations of changing requirements. To position our work in a broader context, we also investigate the growth of software libraries and industrial software systems in general, irrespective of changes in interfaces. This leads to a structural model of changes in software which relates the concepts of software growth, size, work done and quality to each other. These concepts are expected to be the most important factors that play a role in the evolution of software, and as a result, interface instability. The final research question is therefore as follows:

**RQ6:** What factors influence software growth in libraries?

Together, these research questions can help to understand interface instability and its impact on client systems using these interfaces. In the next section, we describe the research method that was used during the creation of this thesis.

1.4. **Research Method**

This thesis investigates current software engineering practice by empirically investigating two large repositories of software libraries and industrial software systems. These datasets provide a rich environment to test our hypotheses. The first dataset is the repository of the Maven build system\(^5\), a large collection of open-source Java libraries. When Maven is used as a build system in a software project, dependencies as specified in the build file of the project are automatically downloaded from the central repository\(^6\). This ensures that all required dependencies of a software project are available in order to successfully compile the system on any machine on which the source code is placed.

The availability of subsequent versions of the same software system (snapshots) in both datasets makes it possible to study changes in these systems over time. The Maven repository is used frequently by a wide variety of software engineers on a daily basis; we assume that many software engineering practices of interest, be it good or bad, are reflected in the source code present in this repository.

The second dataset that we use consists of industrial software systems of which source code is available at the Software Improvement Group (SIG)\(^7\). The repository contains a large number of snapshots of different systems written in different languages, which provides an opportunity to perform cross-language analyses. This repository is used in later chapters in this thesis is also expected to reflect a wide variety of software engineering practices because the source code belongs to a large number of different companies.

These two datasets are a prerequisite to be able to answer research questions of an empirical nature. In fact, our research questions were inspired by the actual practice of software development in the open-source community and software development industry. Often, research ideas for this thesis were conceived by browsing through the repositories or using the repositories in practice. The primary tool of choice in this thesis is statistics, which we use extensively given the empirical nature of our research questions.

---

\(^5\) [http://maven.apache.org](http://maven.apache.org)

\(^6\) [http://search.maven.org](http://search.maven.org)

\(^7\) [http://www.sig.eu](http://www.sig.eu)
When investigating our datasets, we made the assumption that all phenomena of interest can be analyzed by performing measurements on source code. In our view, these phenomena can also include “soft” properties such as code quality and process effectiveness. We assume that all factors of interest that influence software development practice will impact the most important result of that practice, namely the source code. If a phenomenon of interest does not lead to measurable effects on source code, the effect is out of scope of this thesis, or when an hypothesized effect is not found, we assume that it disproves our original hypothesis about the effect. As we will indeed see in this thesis, a lot of phenomena can be measured on source code alone.

This assumption leads to the following research approach, which contains a number of steps which were taken in the large majority of the performed investigations. First, an idea was formed, inspired from research papers or from discussions with colleagues at SIG or the TU Delft. Next, small preliminary analyses were run to perform a quick check on the idea. If found interesting enough, software was written to extract all required variables from the repositories. This software was then run on a selection of appropriate source code using the supercomputer at the TU Delft or servers available at the SIG. This resulted in a table containing cases and variables of interest. On this table, statistical calculations were performed such as correlations, regressions or other specific statistical models. The results were often discussed with other researchers or consultants at the SIG. Based on these results and the discussions, a paper was written.

1.5. Thesis Outline

We start by presenting the Maven Dependency Dataset, the repository on which our analyses are based, in Chapter 2. We explain the repository, the methodology to obtain our metrics from it and the database containing the results.

In Chapter 3, we answer RQ1 by testing to what degree the principles as stated by the open-source community itself are adhered to in practice. Next, in Chapter 4, we answer RQ2 by introducing a set of metrics to measure interface instability. In Chapter 5, to answer RQ3 and RQ4, we investigate the impact of interface instability by investigating ripple effects of interface changes and we also investigate encapsulation as a way to mitigate ripple effects. In Chapter 6, we provide an alternative answer to RQ2 by presenting a change injection method to determine the effect of interface changes on client systems. In Chapter 7, we answer RQ5 by presenting a method to calculate software productivity and comparisons between programming languages by applying compression to source code. Chapter 8 investigates the growth speed of industrial software systems and open-source libraries and serves as the basis for the growth model presented in Chapter 9. In Chapter 9, we answer RQ6 by investigating a model that connects the concepts of software growth, size, work done and quality to each other.

We come back to answer our main research questions in Chapter 10. Addendum A contains installation instructions for the Maven Dependency Dataset, which can be downloaded from the link found in Chapter 2.

8http://www.tudelft.nl
1.6. ORIGIN OF CHAPTERS

Most chapters in this thesis are based on peer-reviewed publications, some chapters are based on papers that have been submitted for review. To keep each chapter self-contained, the introduction and related work sections of each chapter can contain some redundancy. The author of this thesis is the main author of all publications.

- Chapter 2 is based on a short paper titled *The Maven Repository Dataset of Metrics, Changes and Dependencies* which appeared in the 9th Working Conference on Mining Software Repositories (MSR 2012).
- Chapter 3 is based on a paper titled *Semantic Versioning versus Breaking Changes: A Study of the Maven Repository* which appeared in the 14th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2014). An extended journal edition of this paper combined with the contents of Chapter 6 has been submitted for review.
- Chapter 4 is based on a paper titled *Measuring Software Library Stability Through Historical Version Analysis* which appeared in the 28th IEEE International Conference on Software Maintenance (ICSM 2012).
- Chapter 6 is based on an extended journal edition combined with Chapter 3 and is submitted for review.
- Chapter 7 is submitted for review to the industry track of the International Conference on Software Maintenance and Evolution (ICSME) 2015.
- Chapter 8 will be submitted for review to PeerJ Computer Science.
- Chapter 9 is submitted for review to PeerJ Computer Science.

The author has also contributed to the following publications, which are not included in this thesis:

- *An Analysis of Dependence on Third-party Libraries in Open Source and Proprietary Systems, SQM 2012*. The ideas in this short paper formed the basis of this thesis but the paper was not directly used in this thesis.
- *Refactoring Fat Interfaces Using a Genetic Algorithm, ICSME 2014*. The Maven Dependency Dataset served as the basis for the analysis in this paper but the results were not used in this thesis.
- *Final height in survivors of childhood cancer compared with Height Standard Deviation Scores at diagnosis, Annals of Oncology, 2012*

9 [https://peerj.com/computer-science/](https://peerj.com/computer-science/)
Maven is a build system for software projects which also offers the possibility to include dependencies on third-party software libraries in a project. These libraries are stored in a central location, in a Maven repository. The central Maven repository contains over 100,000 open-source Java libraries. We present the Maven Dependency Dataset, which is based on a snapshot of the Maven repository, and contains metrics, changes and dependencies of 148,253 software libraries. Metrics and changes have been calculated at the level of individual methods, classes and packages of multiple library versions. A complete call graph is also presented which includes call, inheritance, containment and historical relationships between all units of the entire repository. In this chapter, we describe our dataset and the methodology used to obtain it. We present different conceptual views of the Maven Dependency Dataset and we also describe limitations and data quality issues that researchers using this data should be aware of.¹

2.1. Introduction

Maven² is a popular build system for software projects. It is typically used for Java projects, but it can be used with other programming languages as well. Each project using Maven needs to define a project build file (pom.xml), in which several project properties can be specified, such as a name of the project, modules that the project contains and dependencies on third-party libraries. When compiling a Maven project, specified dependencies are downloaded from a Maven repository, which is Maven Central³ by default. These dependencies are automatically included and linked to the compiled binary files of the project. This resolves problems with missing dependencies that are frequently en-

¹ Parts of this chapter have been published as “The Maven Repository Dataset of Metrics, Changes and Dependencies”, Mining Software Repositories 2012 [91].
² https://maven.apache.org
³ http://search.maven.org
countered when developers must manually include library dependencies to compile a project.

The analyses in this thesis are performed on a snapshot of the Maven repository, dated July 30, 2011\(^4\). Based on this file, we have created the Maven Dependency Dataset (MDD), in which we have extracted several metrics from this snapshot and have made them available for download by other researchers. Metrics and dependencies are calculated on individual binary jar files and source jar files. Changes between snapshots are calculated on pairs of subsequent library versions.

The goal of this dataset is to facilitate replicable large-scale research on software releases, versions and evolving dependencies at the level of packages, classes and methods. MDD contains code metrics, dependencies, breaking changes between library versions and a complete call graph of the entire Maven repository. This makes it possible to answer a wide range of software evolution-related research questions, such as the following:

- Can we predict when code changes will occur?
- Can we estimate the impact of these changes?
- How fast do libraries adapt to changes in dependencies?
- What patterns can we observe in changes of methods, packages and classes?
- What code properties are associated with a high adaptation and survival rate of library versions?

MDD facilitates answering these and other research questions and we therefore invite other researchers to explore our dataset and use it in innovative ways.

We enriched each Maven artifact with a set of evolution-related metrics to answer research questions about software evolution and maintenance. The size of the dataset and the fact that a large number of different development teams have been releasing artifacts over a large timespan makes it a valuable source for data analysis and hypotheses testing in the field of software evolution. Collected data includes size information (e.g. LOC, number of methods), evolution information (e.g. number of removed methods per release, breaking changes per release) and a complete call graph of the entire repository, containing four different types of dependencies: containment, historical, call and extension/inheritance. We describe these types of dependencies in more detail in Section 2.4.3.

This chapter is structured as follows. In Section 2.2, the permanent download location of our dataset can be found. In Section 2.3, descriptive statistics are presented. Section 2.4 presents the data schemas of databases in our dataset. In Section 2.5, our data collection approach is outlined. In Section 2.6, data quality issues and limitations of our dataset are discussed.

### 2.2. Download Location

The Maven Dependency Dataset can be downloaded from the following location:

\(^4\)Downloaded from [http://juliusdavies.ca/2013/j.emse/bertillonage/maven.tar.gz](http://juliusdavies.ca/2013/j.emse/bertillonage/maven.tar.gz), kindly made available by Julius Davies from the University of British Columbia.
A detailed per-column description of this dataset and instructions how to install it can be found in Addendum A.

2.3. DESCRIPTIVE STATISTICS

Descriptive statistics of the dataset can be found in Figure 2.1. As can be seen in the upper table, the dataset contains a total of 148,253 jar files. When uploading a library to the central repository, library developers can upload binary, source and javadoc jars. Note that not all library versions are uploaded with corresponding source and javadoc jars: only 101,413 of 148,253 libraries (68.4%) have source code available and only 78,766 libraries (53.1%) have javadoc available.

The second part of Figure 2.1 gives information on the size of libraries. It shows that the 75th percentile of number of lines of code is at 2,200, indicating that most libraries in the repository are relatively small. There are 22,111 artifacts (projects) in the repository, with on average 6.7 versions per artifact.

<table>
<thead>
<tr>
<th>Number of binary jar files</th>
<th>148,253</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of source jar files</td>
<td>101,413</td>
</tr>
<tr>
<td>Number of javadoc jar files</td>
<td>78,766</td>
</tr>
<tr>
<td>Unresolved jar references*</td>
<td>3,319</td>
</tr>
<tr>
<td>Total SLOC</td>
<td>350,571,247</td>
</tr>
<tr>
<td>Number of classes</td>
<td>4,174,150</td>
</tr>
<tr>
<td>Number of methods</td>
<td>37,406,546</td>
</tr>
</tbody>
</table>

Table 2.1: Descriptive statistics for libraries in the Maven repository. loc = lines of code, m/j = number of methods per jar, c/j = number of classes per jar, d/j = number of dependencies per jar, v/a = number of versions per artifact, a/g, number of artifacts per groupId. *Libraries sometimes refer to artifacts or versions that are not present in our snapshot.

2.4. DATA SCHEMAS

For performance reasons we used three different types of database formats: a MySQL database, a Berkeley DB database and a Neo4j graph database. The graph database is most suitable to query graph-like structures such as call graphs. The Berkeley DB database is an on-disk key-value store which can look up metrics very quickly. We give a conceptual model of each of these databases in this section.
2.4.1. MySQL Database

The data schema of the MySQL database is presented in Figure 2.2. As can be seen in this figure, it consists of the following tables:

- **files**
  - fileId (PK): integer
  - fullName: string
  - groupId: string
  - artifactId: string
  - version: string
  - reservedNodeId: integer
  - snapshotId: integer
  - hasSource: boolean
  - PageRank: float
  - Betweenness: float
  - Hubbiness: float
  - Authoritativeness: float
  - WRM: float
  - CEM: float
  - RCNO: float
  - PNM: float
  - nrUnits: integer
  - nrNewUnits: integer
  - nrOldUnits: integer
  - nrRemovedUnits: integer
  - deltaUn: float
  - deltaUo: float
  - hws: float
  - maintainability: float
  - CRS: float
  - RL: float
  - updated: datetime
  - enabled: boolean
  - packagePrefix: string

- **stats**
  - vol, dup, us, ui, mc, cb, ci: float
  - mem: integer
  - nc: integer
  - np: integer
  - loc: integer

- **changes**
  - changeId (PK): integer
  - changeType (FK): integer
  - fileIdv1 (FK): integer
  - fileIdv2 (FK): integer
  - packageUnitIdv1: Long
  - packageUnitIdv2: Long
  - methodUnitIdv1: Long
  - methodUnitIdv2: Long
  - classUnitIdv1: Long
  - classUnitIdv2: Long
  - fieldUnitIdv1: Long
  - fieldUnitIdtv2: Long
  - unitTypes
    - unitTypeId (PK): integer
    - parentType (FK): integer
    - description: string
  - unitTypes
    - unitTypeId: integer
    - description: string
    - breaking: boolean

- **units**
  - unitId (PK): Long
  - name: string
  - unitType (FK): integer
  - parentId (FK): integer
  - fileId (FK): integer
  - LOC: integer
  - McCabe: integer
  - nrParams: integer
  - usageCount: integer

- **deps**
  - callId (PK): integer
  - fromField (FK): integer
  - toField (FK): integer
  - isolation: float

- **depTypes**
  - depTypeId: integer
  - description: string

Table 2.2: The MySQL database schema. Some tables are present in the other database formats and are presented here to give an overview of the interconnection between the datasets. Foreign keys are drawn in the schema but have been removed from the database due to performance reasons; however, foreign key identifiers still match with primary key identifiers.

**files** The files table contains information on all library versions. Metrics such as the number of methods (nrUnits), the number of methods compared to the next version (nrNewUnits) and other metrics are stored in this table. Libraries that are referenced by other libraries but which were not found in our dataset are entered in this table without a fullName. The files table also contains the values of the stability metrics which we introduce in Chapter 4. For a more detailed description of the files table, see Addendum A.

**stats** The stats stats table stores metrics such as LOC, McCabe, number of methods and number of classes for each library version. It also contains SIG star ratings, which are further described in [53].

**units** This table is not stored in MySQL but it is shown here to demonstrate that there exist (conceptual) foreign key relationships between the MySQL, Neo4j and Berkeley DB databases. Units can be complete files, packages, classes or methods, which are all stored in this table. Each unit belongs to a certain file and has a fully qualified name.
(the name field). Metric values such as the McCabe, LOC and parameter count are also stored in this table.

changes Different types of changes between library versions are stored in this table. Changes can be breaking, meaning that source code has to be recompiled if using a dependency that introduces such a change. Non-breaking changes are less severe and do not require recompilation. Unit identifiers are looked up in Berkeley DB and are stored in this table, if found. In either case, names of the affected package, class, method or field are also stored for each change.

deps This table contains all library dependencies as present in the build configuration file of a project. When a library depends on another library, a <dependency> section is present in the pom.xml file of the project specifying the exact groupId, artifactId and version of the library it depends on. Also stored in this table is an isolation rating, specifying the percentage of files that does not import the dependency and is essentially a measure of encapsulation of a dependency in a system. This table only contains library dependencies; all other dependency types are stored in the Neo4j database.

Supporting tables such as changeTypes, unitTypes and depTypes are reference tables that give additional information on properties of changes, units and dependencies, respectively. For a complete description of all columns in the MySQL database and instructions on how to query the Berkeley DB database, see the online addendum.

2.4.2. Berkeley DB Database
To make fast lookup of single methods, classes and packages possible, a Berkeley DB database was created. This database contains information on 36,695,764 different methods, classes and packages. Indices on unique unit identifiers, fully qualified name, fileId, unit type, groupId, artifactId and versions have been created to facilitate searching on any of those fields. The unique unit identifiers match the identifiers as used in the Neo4j call graph. The fileId index refers to the fileId column in the MySQL database. Unit type is a number denoting the type of the unit: 1 = jar file, 2 = package, 3 = java file, 4 = class, 5 = method. The script getunits.sh in the replication package is the main interface to query the Berkeley DB database directly and can be used to obtain information on single methods, classes or packages or to obtain a list of units based on a combination of values for any of the mentioned indices.

2.4.3. Neo4j Database
The Neo4j database is a graph database that contains connections between entities on different levels. A conceptual model is shown in Figure 2.3 and an example is shown in Figure 2.4.

Table 2.3: A conceptual model of units in the Neo4j database. (v) = version.
As can be seen in Figure 2.3, there exists several entities and connections between these entities in this database. The entities are jar files, packages, classes and methods. Each of these entities is denoted with a version (v), indicating that a single entity is always a snapshot in time. Each entity is connected to its own type through a “next version” connection, meaning that an entity points to the next version of itself through an edge in Neo4j. The jar, package and class entities point to the package, class and method entities through a “contains” connection, meaning that a package is contained within a jar file, a class is contained within a package and a method is contained within a class. Also, each jar file is connected with possibly multiple other jar files through a “depends on” relationship, meaning that a jar file can have dependencies on other jar files as specified in the pom.xml file. A package can be a subpackage of another package, and a class can extend another class. In our graph database, interfaces are treated the same way as classes. Finally, a method can call another method, as denoted by the “calls” relationship.

With such a graph, it becomes possible to query a large graph of connected software libraries on a meta-level. For instance, the following queries can be naturally answered with this graph database:

• Return all methods which are not present in the next version of the same library;
• Return all methods which call other methods that are in libraries of which the version number changed in the next release;
• Return all jar files with at least 10 packages of which at least 2 methods were added over 3 different versions;

The structure of the graph database is naturally suited to answer questions like these. We encourage other researchers to formulate their own research questions based on this graph format. The database can be queried using the Cypher query language\(^5\).

Figure 2.4 shows an example of two versions of a jar file, 1.1 and 1.2, which contains a package with a single class in it.

2.5. Methodology

The DAS-3 supercomputer\(^6\) was used to process all jar files. The supercomputer consists of 68 dual-node 2.4 GHz computing nodes with 4 GB memory each. The system runs on ClusterVisionOS 2.1, which is based on Scientific Linux 4.3. The system has a central head node which contains the database and distributes commands to the computing nodes. The database was filled in multiple runs; each run took approximately one week. Since tasks can be easily parallellized across a large number of machines, a speedup of approximately 60 times was achieved. Without the supercomputer, total running time of all analyses was estimated to be more than four years. Custom software was developed to obtain all data. Eventually, this software consisted of approximately 10,000 LOC of Java and 3,000 LOC of bash, Python and R scripts.

Figure 2.5 shows the steps that were taken to obtain all data. The numbers in the figure correspond to the following steps:

\(^5\)http://docs.neo4j.org/chunked/stable/cypher-query-lang.html
\(^6\)http://www.cs.vu.nl/das3
2.5. Methodology

(1) First, source code was processed using the supercomputer. The SAT was adapted to run in parallel in multiple machines and was used to obtain metrics and call graphs from source code.

(2) The SAT writes call graph and metric information to a MySQL database for each different artifact. We do not save all databases completely but we extract interesting information from this database and put it in a separate MySQL database as described in Section 2.4.1.

To detect changes between library versions, we use an adapted version of Clirr\(^7\). This tool checks for breaking changes between each two subsequent versions of binary jar files. A breaking change is any change in the next version of a binary jar file which causes compilation errors in systems using it. These changes are also referred to as binary incompatibilities and require users of those libraries to adjust and recompile their code. There exist several types of breaking changes; examples are public method and class removals. The Eclipse Wiki has more information on binary (in)compatibilities in Java\(^8\) and the Java Language Specification\(^9\) contains formal definitions and explanations.

(3) Metrics on more than 200 million methods, classes and packages were collected. To make fast lookup possible, we stored this information in Berkeley DB as described in Section 2.4.2. We created several keys to obtain information on units, such as fully qualified names, unique identifiers and library names. This enables fast retrieval of units that satisfy certain selection criteria.

(4) We use the obtained call graph information to build a graph of methods, classes, packages and jar files. These units are connected through one of four different relationship types: method call, inheritance, historical and containment. This schema was described in Section 2.4.3. This call graph is not restricted to a single version of a library but

---

\(^7\)http://clirr.sourceforge.net
\(^8\)http://wiki.eclipse.org/Evolving_Java-based_APIs
Table 2.5: An overview of our data collection approach.

connects methods, classes and packages from all versions of all libraries in the Maven repository. Mathematically, the graph is a collection of tuples connecting two unit identifiers, annotated with one of the four relationship types.

(5) To reduce the size of the Neo4j database, only unique unit identifiers and connections between units are stored. These identifiers are unitIds stored in the Berkeley DB database. Neo4j makes fast querying of graph structures possible and also enables the usage of specialized graph queries which relational databases cannot handle (for instance, arbitrarily deep transitive queries). Also, graph traversals can be performed which start at a specific node and visit related nodes to obtain specific information.

(6) The Neo4j graph can be used to query a specific library or a specific method and to investigate changes through time. The graph can also be used to visualize connected units.

(7) The information from Berkeley DB can also be used directly, for instance to obtain a list of all methods present in a certain version of a library or to get information on a specific method in a specific library.

2.6. Limitations
Since the dataset is based on a snapshot of the Maven repository, updates to this repository after the snapshot date are not taken into account into this dataset. Furthermore, users of this dataset should be aware of the following limitations and data quality issues:

2.6.1. Skipped Libraries
For several reasons, not all libraries have been analyzed:

- Source jars are not available for specific library versions;
- Source jars sometimes contain other languages than Java, contain only test code, property files or binary class files;
2.7. Conclusion

We presented MDD, the Maven Dependency Dataset, which contains metrics on 148,253 Java libraries. We presented conceptual schemas of three different databases. First, we

- Some source jars are corrupted.

We assume that these missing libraries are randomly distributed over the entire set of libraries, and that they do not introduce a bias in our dataset.

2.6.2. Package Prefixes

Due to the large size of the dataset it is impossible to manually check data quality. This is also true for package prefixes, which are stored in the files table and which were used to calculate the isolation rating, which is described in Chapter 5, as stored in the deps table [90]. One problem is that some libraries use multiple package prefixes. For example, com.thoughtworks.selenium and org.openqa.selenium occur in the same library version and denote the same base package of the same library. To mark an import statement in another library as third-party, both strings have to be recognized. Furthermore, some libraries do not have a common package prefix but use several different notations, making automated detection more difficult. We expect that there does not exist a bias in systems that have missing package prefixes.

2.6.3. Usage Frequencies

Our dataset also includes usage frequencies of methods, which we have used determine the expected impact of changes [90]. We calculated these usage frequencies on binary dumps of disassembled class files. This means that the calls present in binary class files can be different from the calls present in source code. This becomes visible, for instance, with calls to StringBuilder.append, which is the most frequently called method in the Maven repository. This, however, is caused by the fact that the Java compiler replaces string concatenation using “+” with calls to StringBuilder.

Another issue is whether the usage frequencies of libraries by other libraries are representative for the usage frequencies of libraries by actual systems. We do not possess the same data about non-library systems on such a large scale, and therefore we restrict our analysis to inter-library usage.

2.6.4. Wrong Snapshot Identifiers

A final data problem is the automatic labeling of snapshot numbers as stored in the snapshotId column of the files table. If there are three libraries with the same groupId and artifactId, and the version numbers are 1.2.2, 1.2.3, and 1.3.0, we expect them to get the subsequent version numbers 1, 2 and 3. These numbers are used throughout this entire thesis when a comparison between two snapshots is performed. The Maven indexing software itself contains an algorithm to order releases based on version strings. Manual inspection shows that this software makes mistakes in a very small number of cases. Due to the large scale of the repository, it was impossible to manually check all numberings and version strings. We expect the impact of these errors to be negligible given the large scale of the repository.

2.7. Conclusion

We presented MDD, the Maven Dependency Dataset, which contains metrics on 148,253 Java libraries. We presented conceptual schemas of three different databases. First, we
presented a relational database which contains information on individual files and dependencies as well as breaking changes in these files. Next, we presented a key-value database containing information on individual methods, classes and packages. Finally, we presented a graph database which contains all connections between methods, classes and packages of the entire Maven repository. We described our methodology to obtain our data and we discussed data quality issues present in our dataset.
According to semantic versioning principles, the version string of each software release should have the form “MAJOR.MINOR.PATCH”, where there are strict rules regarding the incrementation of the major, minor and patch version numbers. The major version number should be incremented when incompatible API changes are made, the minor version number should be incremented when backward-compatible functionality is added and the patch version should be incremented when backward-compatible bug fixes are made. In this chapter, we investigate to what degree semantic versioning rules are adhered to in practice. We investigate backward incompatible changes, release intervals, the number of functional changes and migration patterns in the Maven repository. We find that the adherence to semantic versioning principles in the Maven repository is low, but slowly increases over time. Major releases tend to be released faster than minor or patch releases, and developers tend to update to major releases of dependencies faster than minor or patch releases. Finally, deprecation patterns as suggested by semantic versioning are not adhered to in practice.\(^1\)

### 3.1. Introduction

For users of software libraries or application programming interfaces (APIs), backward compatibility is a desirable trait. Without backward compatibility, library users will face increased risk and cost when upgrading their dependencies. In spite of these costs and risks, library upgrades may be desirable or even necessary, for example if the newer version contains required additional functionality or critical security fixes. To conduct the upgrade, the library user will need to know whether there are incompatibilities, and, if so, which ones.

---

\(^1\)Parts of this chapter have been published as “Semantic Versioning versus Breaking Changes: A Study of the Maven Repository”, SCAM 2014 \(^93\).
Determining whether there are incompatibilities, however, is hard to do for the library user (it is, in fact, undecidable in general). Therefore, it is the library creator’s responsibility to indicate the level of compatibility of a library update. One way to inform library users about incompatibilities is through version numbers. As an example, semantic versioning\(^2\) (semver) suggests a versioning scheme in which three digit version numbers MAJOR.MINOR.PATCH have the following semantics:

**MAJOR:** This number should be incremented when incompatible API changes are made;

**MINOR:** This number should be incremented when functionality is added in a backward-compatible manner;

**PATCH:** This number should be incremented when backward-compatible bug fixes are made.

These principles were formulated in 2010 by (GitHub\(^3\) founder) Tom Preston-Werner, and GitHub actively promotes semver and encourages all 10,000,000 projects hosted by GitHub to adopt it. Similarly, the Maven Central repository, the repository used to collect dependencies that are specified using the build tool Maven, strongly recommends following semver when releasing new library versions\(^4\).

Semantic versioning principles have also been embraced in the Javascript community. An example of a Javascript project that explicitly announced to follow semver is jQuery, which state that “the team has tried to walk the line between maintaining compatibility with code from the past versus supporting the best web development practices of the present”\(^5\). Another example is NPM (Node Package Manager)\(^6\), a build tool for Javascript similar to Maven, which requires users to follow semver when submitting a new version of a library\(^7\).

In the .NET community, NuGet\(^8\) is a build tool and software repository for libraries that does not enforce or recommend versioning guidelines. It automatically includes the latest version of dependencies in software projects. This leads to problems when these releases contain breaking changes\(^9\).

An example of a software project which demonstrates that including breaking changes in non-major releases causes problems for software developers is JUnit. In its 4.12-beta-1 release, JUnit introduced breaking changes as compared to its previous release. In version 4.12-beta-2, these breaking changes have been reversed after complaints of library users\(^10\).

Although not all developers of the projects mentioned before may be aware of the semantic versioning standard or other official rules regarding incrementing major, minor or patch versions, a lot of library users implicitly assume that non-major releases should

---

\(^2\) [http://semver.org](http://semver.org)

\(^3\) [https://github.com](https://github.com)

\(^4\) [http://central.sonatype.org/pages/requirements.html](http://central.sonatype.org/pages/requirements.html)


\(^6\) [http://www.npmjs.com](http://www.npmjs.com)

\(^7\) [https://docs.npmjs.com/getting-started/semantic-versioning](https://docs.npmjs.com/getting-started/semantic-versioning)

\(^8\) [https://www.nuget.org](https://www.nuget.org)

\(^9\) [http://blog.nuget.org/20141010/nuget-is-broken.html](http://blog.nuget.org/20141010/nuget-is-broken.html)

\(^10\) [https://groups.yahoo.com/neo/groups/junit/conversations/topics/24572](https://groups.yahoo.com/neo/groups/junit/conversations/topics/24572)
not include breaking changes. As argued in the semantic versioning specification, “these rules are based on but not necessarily limited to pre-existing widespread common practices in use in both closed and open-source software.”

Similarly, Microsoft .NET suggests the following distinction between major and minor releases

**Major**: “A higher version number might indicate a major rewrite of a product where backward compatibility cannot be assumed.”

**Minor**: “If the name and major version number on two assemblies are the same, but the minor version number is different, this indicates significant enhancement with the intention of backward compatibility.”

But how common are these practices in reality, in open-source Java libraries? Are breaking changes just harmless, or do they actually hurt by causing rework? Do breaking changes mostly occur in major releases, or do they occur in minor releases as well? Furthermore, for the breaking changes that do occur, to what extent are they signalled through, e.g., *deprecation tags*? Does the presence of breaking changes affect the time (delay) between library version release and actual adoption of the new release in clients?

In this chapter, we seek to answer questions like these. For this, we use the full seven years of historical data present in the Maven Dependency Dataset.

As an approximation of the (undecidable) notion of backward compatibility, we use *binary compatibility* as defined in the Java language specification. This is an underestimation, since binary incompatibilities are certainly breaking, but there are likely to be different (semantic) incompatibilities as well. As a measurement for the amount of changed functionality in a release, we will use the *edit script size* between two subsequent releases. Equipped with this, we will study versioning practices in the Maven dataset, and contrast them with the idealized guidelines as expressed in the *semver* specification. Even though we do not expect that all developers that submit code to the Maven repository are aware of the guidelines of *semver*, we still expect that most developers are aware that most other developers perceive a difference in changing a patch, minor or major version number in the next release.

This chapter is structured as follows. We start out, in Section 3.2, by sketching related work in the area of binary incompatibilities and change impact analysis. In Section 3.3, we formulate the research questions we seek to answer. Then, in Section 3.4, we describe our approach to answer these questions, and how we measure, e.g., breaking changes, changed functionality, and deprecation. In Section 3.5–3.8 we present our analysis in full detail. We discuss the wider implications and the threats to the validity of our findings in Sections 3.9 and 3.10, after which we conclude the chapter in Section 3.11.

### 3.2. RELATED WORK

To the best of our knowledge, our work is the first systematic study of versioning principles in a large collection of Java libraries. However, several case studies on backward compatible and incompatible changes in public interfaces as appearing in these libraries

have been performed \[29, 35, 38, 78, 104\]. For instance, Cossette et al. \[29\] investigate binary incompatibilities introduced in five different libraries and aim to detect the correct adaptations to upgrade to the newer version of the library. Similarly, Dig et al. \[38\] investigate binary incompatibilities in five other libraries and conclude that most of the backward incompatible API changes are behavior-preserving refactorings. Dietrich et al. \[35\] have performed an empirical study into evolution problems caused by library upgrades. They manually detect different kinds of source and binary incompatibilities, and conclude that although incompatibility issues do occur in practice, the selected set of issues does not appear very often.

Another area of active research is to automatically detect refactorings based on changes in public interfaces \[11, 30, 31, 36, 54, 63, 116\]. The idea behind these approaches is that these refactorings can automatically be “replayed” to update to a newer version of a library. This way, an adaptation layer between the old and the new version of the library can automatically be created, thus shielding the system using that library from backward incompatible changes.

While our work investigates backward incompatibilities for given version string changes, Bauml et al. \[13\] take the opposite approach, in the sense that they propose a method to generate version number changes based on changes in OSGi\[12\] bundles. A comparable approach in the Maven repository would be to create a plugin that automatically determines the correct subsequent version number based on backward incompatibilities and the amount of new functionality present in the new release as compared to the previous one.

The Maven repository has been used in other work as well. Davies et al. \[34\] use the same dataset to investigate the provenance of a software library, for instance, if the source code was copied from another library. They deploy several different techniques to uniquely identify a library, and find out its history, much like a crime scene containing a fingerprint. Ossher et al. \[85\] also use the Maven repository to reconstruct a repository structure with directories and version based on a collection of libraries of which the groupId, artifactId and version are not known.

Issues with backward incompatibilities can also be found in web interfaces. Romano et al. \[97\] investigate changes in the context of service oriented architectures, in which a web interface is considered to be a contract between subscribers and providers. These interfaces are shown to suffer from the same type of problems as investigated in this chapter, which leads to rework on the side of the subscribers of these interfaces. The authors propose a tool that compares subsequent versions of these web interfaces to automatically extract changes.

Developer reactions to API deprecations has been investigated for the Smalltalk language and ecosystem by Robbes et al. \[95\]. They have investigated a set of more than 2,600 distinct Smalltalk systems which contained 577 deprecated methods and 186 deprecated classes, and found that API changes caused by deprecation can have a large impact on developers using that API.

\[12\] http://www.osgi.org
3.3. **Research Questions**

The overall goal of this chapter is to understand to what degree developers of software libraries use versioning conventions in the development of these libraries, and what the impact of unstable interfaces is on clients using these libraries. We investigate instability of interfaces through the number of compilation errors caused by breaking changes and the dispersion of these errors through libraries using the changed interfaces.

Even though not all developers might be aware of the semver standard, we still regard semver as a formalization of principles that are considered to be best practices, even before the manifesto was released in 2010. As mentioned before, the prime example of such a best practice is not to include breaking changes in non-major releases.

By investigating semver, we can find out to what degree the best practices as encoded in this standard are actually adhered to in practice. This way, we can find out if developers actually mean to give a signal, for instance, that a library contains only backward-compatible bug fixes when releasing a new patch version, or that a library introduces a substantial number of backward-incompatible changes to its public interface when releasing a new major version.

Similar to versioning guidelines, deprecation tags provide library developers with a means to signal whether a certain method can safely be used or whether it may be removed in a next version. We also investigate the use of deprecation tags in our dataset and find out how they are applied.

To achieve our overall goal, we seek to answer the following research questions:

- **RQ1**: How are semantic versioning principles applied in practice in the Maven repository, in terms of binary (in)compatible changes and new functionality?
- **RQ2**: Has the adherence to semantic versioning principles increased over time?
- **RQ3**: How are dependencies to newer versions updated, and what are factors causing systems not to include the latest versions of dependencies?
- **RQ4**: How are deprecation tags applied to methods in the Maven repository?

In the next section, we discuss our research method.

3.4. **Method**

3.4.1. **Determining Breaking Changes**

Determining full backward compatibility amounts to determining equivalence of functions, which in general is undecidable. Instead of such semantic compatibility, we will rely on binary incompatibilities.

Binary incompatible changes, in this chapter also called breaking changes, are formally defined by the Java Language specification as follows: “a change to a type is binary compatible with (equivalently, does not break binary compatibility with) pre-existing binaries if pre-existing binaries that previously linked without error will continue to link without error”. We will use the following working definition: breaking changes are any changes to a library interface that require recompilation of systems using the changed
functionality. Examples of breaking changes are method removals and changes in return types\textsuperscript{13}.

To detect breaking changes between each subsequent pair of library versions, we use Clirr. Clirr is a tool that takes two jar files as input and returns a list of changes in the public API. Clirr is capable of detecting 43 API changes in total, of which 23 are considered breaking and 20 are considered non-breaking. Clirr does not detect all breaking changes that exist, but it does detect the most common ones. We executed Clirr on the complete set of all subsequent versions of releases in the Maven repository.

Whenever Clirr finds a binary incompatibility between two releases, those releases are certainly not compatible. However, if Clirr fails to find a binary incompatibility, the releases can still be semantically incompatible. As such, our reports on e.g., the percentage of releases introducing breaking changes is an underestimation: The actual situation may be worse, but not better.

### 3.4.2. Determining Subsequent Versions and Update Types

In the Maven repository, each library version (a single jar file) is uniquely identified by its groupId, artifactId, and version, for instance “junit”, “junit” and “4.8.1”. To determine subsequent version pairs, we sort all versions with the same groupId and artifactId based on their version string. We used the Maven Artifact API\textsuperscript{14} to compare version strings with each other, taking into account the proper sorting given the major, minor, patch and prerelease in a given version string. For each subsequent pair of releases from this sorted list, the release type is determined according to the change in version number. For instance, a change in version number from “1.1.1” to “1.2.0” was marked as a minor release. We do not check whether version numbers are incremented properly, i.e. if there are no gaps in version numbers. The result is that each pair of subsequent versions is marked as either a major, a minor or a patch update.

Since semver applies only to version numbers containing a major, minor and patch version number, we only investigate pairs of library versions which are both structured according to the format “MAJOR.MINOR.PATCH” or “MAJOR.MINOR”. In the latter case, we assume an implicit patch version number of 0.

Semantic versioning also permits prereleases, such as 1.2.3-beta1 or (as commonly used in a maven setting) 1.2.3-SNAPSHOT. We exclude prereleases from our analysis since semver does not provide any rules regarding breaking changes or new functionality in these release types.

### 3.4.3. Detecting Changed Functionality

To estimate the amount of changed functionality between releases, we calculate the edit script size between each pair of subsequent versions. We do so by calculating differences between abstract syntax trees (ASTs) of the two versions. The edit script size represents the total number of statements that needs to be inserted, deleted, updated or moved to convert the first version of the library into the second. We use the static code anal-

\textsuperscript{13}For an overview of different types of binary incompatibilities and a detailed explanation, see \url{http://wiki.eclipse.org/Evolving_Java-based_APIs}

\textsuperscript{14}\url{http://maven.apache.org/ref/3.1.1/maven-artifact}
ysis tool ChangeDistiller\textsuperscript{15} to calculate edit scripts between library versions. For more information on ChangeDistiller, we refer to [41, 46].

Figure 3.1 shows an example of two pieces of code and the steps as determined by ChangeDistiller to convert the first version of the method into the second one. ChangeDistiller detects that the statement \texttt{int x = 1;} (line 2) is updated with a new value of 2. Also, it detects that the \texttt{if}-statement on line 4 of version 1 is deleted, and the statement \texttt{x--} (line 5) is moved. Altogether, the size of the edit script to convert the first version into the second is 5: one update, two delete, one insert and one move operation.

\begin{table}
\centering
\begin{tabular}{|c|c|}
\hline
\texttt{public void m1() {} } & \texttt{m1() } \\
\hline
\texttt{int x = 1;} & \texttt{int x = 2;} \\
\hline
\texttt{while (true) {}} & \texttt{while (true) {}} \\
\hline
\texttt{if (x > 0)} & \texttt{if (x > 0)} \\
\hline
\texttt{x--;} & \texttt{x--;} \\
\hline
\end{tabular}
\caption{An example of the calculation of an edit script between two version of a method. The resulting edit script has size of 5: one update, two delete, one insert and one move operation.}
\end{table}

We use edit script script as representation of changed functionality instead of changes in lines of code for the following reasons:

1. It closely resembles the actual rework developers have performed between two releases;
2. It is not sensitive to changes in layout, whitespace, and comments;
3. It can be obtained automatically, which is a requirement given the large size of the repository.

3.4.4. Obtaining release intervals and dependencies

To calculate release intervals, we collect upload dates for each jar file in the Maven Central Repository. Unfortunately, not for all libraries a valid upload date is available. Ultimately, for 129,183 out of 144,934 (89.1\%) libraries we could identify a valid release date. To identify moments of dependency updates, we combine the upload dates with updates to dependencies as stored in \texttt{maven\textbackslash{}pom.xml} build files.

\textsuperscript{15}https://bitbucket.org/sealuzh/tools-changedistiller
Table 3.2: Version string patterns and frequencies of occurrence in the Maven repository.

<table>
<thead>
<tr>
<th>#</th>
<th>Pattern</th>
<th>Example</th>
<th>#Single</th>
<th>#Pairs</th>
<th>Incl.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MAJOR.MINOR</td>
<td>2.0</td>
<td>20,680</td>
<td>11,559</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>MAJOR.MINOR.PATCH</td>
<td>2.0.1</td>
<td>65,515</td>
<td>50,020</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>#1 or #2 with nonnum. chars</td>
<td>2.0.D1</td>
<td>3,269</td>
<td>2,150</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>MAJOR.MINOR-prerelease</td>
<td>2.0-beta1</td>
<td>16,115</td>
<td>10,756</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>MAJOR.MINOR.PATCH-pre.</td>
<td>2.0.1-beta1</td>
<td>12,874</td>
<td>8,939</td>
<td>no</td>
</tr>
<tr>
<td>6</td>
<td>Other versioning scheme</td>
<td>2.0.1.5.4</td>
<td>10,930</td>
<td>8,307</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
<td>129,138</td>
<td>91,731</td>
<td></td>
</tr>
</tbody>
</table>

3.4.5. Obtaining Deprecation Patterns

For API developers, the Java language offers the possibility to warn about future incompatibilities by means of the "@Deprecated" annotation.\(^\text{16}\) The \texttt{semver} standard states that public methods should be annotated with the deprecated tag before they are removed.

We detect deprecated methods in the following way. We extract the source code from source jar files for each library and, for performance reasons, textually search for occurrences of the string "@Deprecated" first. Only when at least one deprecated tag is found, we parse the complete source code of the library using the JDT (Java Development Tools) Core library\(^\text{17}\).

Using JDT, we create an abstract syntax tree for each source file, and apply a visitor to find out which methods have deprecation tags. Next versions of the same method are connected using method header (name and parameters) matching. Combining this information with the update types from Section 3.4.2 makes it possible to distinguish between different types of deprecation patterns.

3.5. Application of Semantic Versioning

We first investigate different version string patterns that can be found in our repository. After this, we determine how many major, minor and patch releases actually occur in our dataset, and differences between these update types in terms of release cycle and average number of breaking changes.

3.5.1. Version String Patterns

Table 3.2 shows the six most common version string patterns that occur in the Maven repository. For each pattern, the table shows the number of libraries with version strings that match that pattern (#Single) and the number of subsequent versions that both follow the same pattern (#Pairs) – we will use the latter to identify breaking changes between subsequent releases.

The first three versioning schemes correspond to actual \texttt{semver} releases, whereas the remaining ones correspond to \texttt{prereleases}. Since prereleases can be more tolerant in terms of breaking changes (\texttt{semver} does not state what the relationship between prereleases and non-prereleases in terms of breaking changes and new functionality is)\(^\text{18}\) we

\(^\text{16}\)\url{http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/deprecation/deprecation.html}
\(^\text{17}\)\url{http://www.eclipse.org/jdt/core}
\(^\text{18}\)Pre-releases in maven correspond to -SNAPSHOT releases, which should not be distributed via Maven's
exclude prereleases from our analysis.

The table shows that the majority of the version strings (69.3%) is formatted according to the first two schemes, and 22.3% of the version strings contains a prerelease label (patterns 4 and 5). The difference between the single and the pair frequency is due to two reasons: (1) the second version string of an update can follow a different pattern than the first; and (2) a large number of libraries only has a single release (6,442 out of 22,205 libraries, 29%).

3.5.2. BREAKING AND NON-BREAKING CHANGES

Table 3.3 shows the top 20 breaking changes and top 10 non-breaking changes in the Maven repository as detected by Clirr. The breaking changes in these table are obtained from the 126,070 potential updates from the first version of a library (denoted as $L_y$) to a next version (denoted as $L_{y+1}$). The most frequently occurring breaking change is the method removal, with 177,480 occurrences. A method removal is considered to be a breaking change because the removal of a method leads to compilation errors in all places where this method is used. The most frequently occurring non-breaking change as detected by Clirr is the method addition, with 518,690 occurrences.

Table 3.4 shows the number of major, minor and patch releases containing at least one breaking change. The table shows that 35.8% of major releases contains at least one breaking change. We also see that 35.7% of minor releases and 23.8% of patch releases contain at least one breaking change. This is in sharp contrast to the best practice that minor and patch releases should be backward compatible. The overall number of releases that contain at least one breaking change is 30.0%.

The table shows that there does not exist a large difference between the percentage of major and minor releases that contain breaking changes. This indicates that best practices such as encoded in semver are not adhered to in practice with respect to breaking changes. The total number of updates in Table 3.4 (80,589) differs from the total number of pairs in Table 3.2 (91,731) because of missing or corrupt jar files, which have a correct version string but cannot be analyzed by Clirr.

3.5.3. MAJOR VS MINOR VS PATCH RELEASES

To understand the adherence of semantic versioning principles for major, minor, and patch releases, Table 3.5 shows the average number of breaking changes, non-breaking changes, edit script size and number of days for the different release types. Each release is compared to its immediate previous release, regardless of the release type of this previous release.

As the table shows, on average there are 58 breaking changes in a major release. Minor and patch releases introduce fewer breaking changes (around half as many as the major releases), but 27 and 30 on average is still a substantial number (and clearly not 0 as semantic versioning requires). The differences between the three update types are significant with $F = 7.31$ and $p = 0$, tested with a nonparametric Kruskall-Wallis test, since the data is not normally distributed\(^{19}\).

---

\(^{19}\)Even if the data is not normally distributed, we still summarize the data with a mean and standard deviation.
### Breaking changes

<table>
<thead>
<tr>
<th>#</th>
<th>Change type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Method has been removed</td>
<td>177,480</td>
</tr>
<tr>
<td>2</td>
<td>Class has been removed</td>
<td>168,743</td>
</tr>
<tr>
<td>3</td>
<td>Field has been removed</td>
<td>126,334</td>
</tr>
<tr>
<td>4</td>
<td>Parameter type change</td>
<td>69,335</td>
</tr>
<tr>
<td>5</td>
<td>Method return type change</td>
<td>54,742</td>
</tr>
<tr>
<td>6</td>
<td>Interface has been removed</td>
<td>46,852</td>
</tr>
<tr>
<td>7</td>
<td>Number of arguments changed</td>
<td>42,286</td>
</tr>
<tr>
<td>8</td>
<td>Method added to interface</td>
<td>28,833</td>
</tr>
<tr>
<td>9</td>
<td>Field type change</td>
<td>27,306</td>
</tr>
<tr>
<td>10</td>
<td>Field removed, previously constant</td>
<td>12,979</td>
</tr>
<tr>
<td>11</td>
<td>Removed from the list of superclasses</td>
<td>9,429</td>
</tr>
<tr>
<td>12</td>
<td>Field is now final</td>
<td>9,351</td>
</tr>
<tr>
<td>13</td>
<td>Accessibility of method has been decreased</td>
<td>6,520</td>
</tr>
<tr>
<td>14</td>
<td>Accessibility of field has been weakened</td>
<td>6,381</td>
</tr>
<tr>
<td>15</td>
<td>Method is now final</td>
<td>5,641</td>
</tr>
<tr>
<td>16</td>
<td>Abstract method has been weakened</td>
<td>2,532</td>
</tr>
<tr>
<td>17</td>
<td>Added final modifier</td>
<td>1,260</td>
</tr>
<tr>
<td>18</td>
<td>Field is now static</td>
<td>726</td>
</tr>
<tr>
<td>19</td>
<td>Added abstract modifier</td>
<td>564</td>
</tr>
<tr>
<td>20</td>
<td>Field is now non-static</td>
<td>509</td>
</tr>
</tbody>
</table>

### Non-breaking changes

<table>
<thead>
<tr>
<th>#</th>
<th>Change type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Method has been added</td>
<td>518,690</td>
</tr>
<tr>
<td>2</td>
<td>Class has been added</td>
<td>216,117</td>
</tr>
<tr>
<td>3</td>
<td>Field has been added</td>
<td>206,851</td>
</tr>
<tr>
<td>4</td>
<td>Interface has been added</td>
<td>32,569</td>
</tr>
<tr>
<td>5</td>
<td>Method removed, inherited still exists</td>
<td>25,170</td>
</tr>
<tr>
<td>6</td>
<td>Field accessibility increased</td>
<td>24,954</td>
</tr>
<tr>
<td>7</td>
<td>Value of compile-time constant changed</td>
<td>16,768</td>
</tr>
<tr>
<td>8</td>
<td>Method accessibility increased</td>
<td>14,630</td>
</tr>
<tr>
<td>9</td>
<td>Addition to list of superclasses</td>
<td>13,497</td>
</tr>
<tr>
<td>10</td>
<td>Method no longer final</td>
<td>9,202</td>
</tr>
</tbody>
</table>

Table 3.3: The most common breaking and non-breaking changes in the Maven repository as detected by Clirr.

<table>
<thead>
<tr>
<th>Update type</th>
<th>Contains at least 1 breaking change</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
<td>%</td>
<td>No</td>
</tr>
<tr>
<td><strong>Major</strong></td>
<td>4,268</td>
<td>35.8%</td>
<td>7,624</td>
</tr>
<tr>
<td><strong>Minor</strong></td>
<td>10,690</td>
<td>35.7%</td>
<td>19,267</td>
</tr>
<tr>
<td><strong>Patch</strong></td>
<td>9,239</td>
<td>23.8%</td>
<td>29,501</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>24,197</td>
<td>30.0%</td>
<td>56,392</td>
</tr>
</tbody>
</table>

Table 3.4: The number of major, minor and patch releases that contain breaking changes.
3.5. Application of Semantic Versioning

3.5. A PPLICATION OF S EMANTIC V ERSIONING

Breaking

<table>
<thead>
<tr>
<th>Type</th>
<th>μ</th>
<th>σ²</th>
<th>μ</th>
<th>σ²</th>
<th>μ</th>
<th>σ²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major</td>
<td>58.3</td>
<td>337.3</td>
<td>90.7</td>
<td>582.1</td>
<td>50.0</td>
<td>173.0</td>
</tr>
<tr>
<td>Minor</td>
<td>27.4</td>
<td>284.7</td>
<td>52.2</td>
<td>255.5</td>
<td>52.7</td>
<td>190.5</td>
</tr>
<tr>
<td>Patch</td>
<td>30.1</td>
<td>204.6</td>
<td>42.8</td>
<td>217.8</td>
<td>22.7</td>
<td>106.5</td>
</tr>
<tr>
<td>Total</td>
<td>32.0</td>
<td>264.3</td>
<td>52.2</td>
<td>293.3</td>
<td>37.2</td>
<td>152.3</td>
</tr>
</tbody>
</table>

Table 3.5: Analysis of the number of breaking and non-breaking changes, edit script size, and release intervals of major, minor, and patch releases.

In terms of size, major releases are somewhat smaller than minor releases (average edit script size of 50 and 52, respectively), with patch releases substantially smaller (22), with \( F = 117.49 \) and \( p = 0 \). This provides support for the rule in semver stating that patch releases should contain only bug fixes, which overall would lead to smaller edit script sizes than new functionality.

With respect to release intervals, these are on average 2 (for major and patch releases) to 2.5 months (for minor releases), with \( F = 115.47 \) and \( p = 0 \). It is interesting to see that minor, and not major updates take the longest time to release.

Care must be taken when interpreting the mean for skewed data. All data in this table follows a strong power law, in which the most observations are closer to 0 and there are a relative small amount of large outliers. Nonetheless, a larger mean indicates that there are more large outliers present in the data.

3.5.4. MEDIAN ANALYSIS

To find out how the number of days since the previous release relates to the update type of the release, we perform a quantile regression that shows the median number of days that an update in each category approximately takes. Since the data is highly skewed, we perform a bootstrap to resample from the skewed distributions, which provides normal distributions. To further prevent the influence of extreme outliers, we estimate the median number of days instead of the average number of days per group.

Table 3.6 shows the result of the analysis. Practically, the table shows us that major releases are released at a median number of days of 42. Minor releases are released at a median number of days of \( 42 + 10 = 52 \), and patch releases take a median of \( 42 - 3 = 39 \) days to be released.

<table>
<thead>
<tr>
<th>Release type</th>
<th>Median coeff.</th>
<th>Bootstr. std. error</th>
<th>p-value</th>
<th>95% C.I.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minor</td>
<td>10</td>
<td>1.319</td>
<td>0.000</td>
<td>7.416 - 12.584</td>
</tr>
<tr>
<td>Patch</td>
<td>-3</td>
<td>1.353</td>
<td>0.027</td>
<td>-5.652 - -0.348</td>
</tr>
<tr>
<td>constant (major)</td>
<td>42</td>
<td>1.128</td>
<td>0.000</td>
<td>39.50 - 44.50</td>
</tr>
</tbody>
</table>

Table 3.6: ANOVA analysis to compare the number of breaking changes and the churn in major, minor and patch releases.

This shows that minor releases tend to take longer to be released than major releases.
An ANOVA analysis based on averages \( n = 58763, \ F = 0 \) gives 79 days for major, 84 days for minor and 61 days for patch releases, also showing that minor releases tend to take longer on average to be released than major releases. A possible explanation is that a major release contains less rework that takes a large development effort but instead mainly contains changes to the interface instead of rework effort in the entire library, which would take more time. An alternative explanation is that development on major releases started on a separate branch earlier than the update dates in our data shows.

### 3.5.5. Breaking Changes and Errors

To determine the actual impact of breaking changes in releases, we investigate the number of breaking changes and the relationship with compilation errors in this section.

Table 3.7 shows overview statistics for the 10 different types of breaking changes detected by applying Algorithm 1 to the entire Maven repository.

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Frequency</th>
<th>#Errors</th>
<th>#E/F</th>
<th>#sys</th>
<th>#uniq</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MR</td>
<td>177,480</td>
<td>1,524,498</td>
<td>8.59</td>
<td>8,328</td>
<td>960</td>
</tr>
<tr>
<td>2</td>
<td>CR</td>
<td>168,743</td>
<td>1,645,518</td>
<td>9.75</td>
<td>3,983</td>
<td>505</td>
</tr>
<tr>
<td>3</td>
<td>FR</td>
<td>126,334</td>
<td>4,143,723</td>
<td>32.80</td>
<td>8,028</td>
<td>960</td>
</tr>
<tr>
<td>4</td>
<td>PTC</td>
<td>69,335</td>
<td>956,314</td>
<td>13.79</td>
<td>5,357</td>
<td>547</td>
</tr>
<tr>
<td>5</td>
<td>RTC</td>
<td>54,742</td>
<td>288,939</td>
<td>5.28</td>
<td>4,478</td>
<td>433</td>
</tr>
<tr>
<td>6</td>
<td>IR</td>
<td>46,852</td>
<td>95,250</td>
<td>2.03</td>
<td>1,657</td>
<td>130</td>
</tr>
<tr>
<td>7</td>
<td>NPC</td>
<td>42,286</td>
<td>533,741</td>
<td>12.62</td>
<td>5,071</td>
<td>713</td>
</tr>
<tr>
<td>8</td>
<td>MAI</td>
<td>28,833</td>
<td>126,427</td>
<td>4.38</td>
<td>4,746</td>
<td>562</td>
</tr>
<tr>
<td>9</td>
<td>FTC</td>
<td>27,306</td>
<td>1,233,095</td>
<td>45.16</td>
<td>4,324</td>
<td>485</td>
</tr>
<tr>
<td>10</td>
<td>CFR</td>
<td>12,979</td>
<td>677,234</td>
<td>52.18</td>
<td>3,354</td>
<td>317</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>595,158</td>
<td>11,139,014</td>
<td>18.72</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.7: The types of changes detected. Frequency = the number of times this change type occurred in an update, #Errors = The number of errors this update type caused in all \( S_x \), #E/F = the average number of errors per breaking change, #sys = The number of distinct \( S_x \) that contain errors because of this update, #uniq = The number of different updates of \( L_y \) that contain this change.

The table shows the number of breaking changes and the number of compilation errors these changes cause. For instance, class removals occur 168,743 times and cause a total of 1,645,518 compilation errors when applying the algorithm to the entire repository. The most frequently occurring breaking change is the method removal, occurring 177,480 times in the repository and causing 1,645,518 compilation errors in total. For method removals, there are 3,983 unique jar files that contain compilation errors caused by breaking changes in 505 unique jar files. Another type of frequently occurring breaking change is the class removal, which appears 126,334 times in our dataset and causes 1,645,518 errors.

The average number of errors per breaking change is also shown in Table 3.7. It shows that a constant field removal (CFR) has the highest average number of errors per change: 52.18. Furthermore, field type changes (45.16), field removals (32.8) and parameter type changes (13.79) cause a relatively large number of compilation errors as compared to other change types. On average, a breaking change causes 18.72 errors.

Applying all possible library updates and collecting all compilation errors gives a total of 595,158 breaking changes of the 10 most occurring change types and a total of
3.5. Application of Semantic Versioning

11,139,014 compilation errors because of these changes.

3.5.6. The Relationship Between Breaking Changes and Errors

To further investigate the relationship between breaking changes and the number of errors caused by these changes, we calculate the correlation between these properties. The Spearman rank correlation between the number of breaking changes in $\Delta L_{y,y+1}$ and the number of errors in $S_x$ caused by these changes is 0.65 ($p = 0$), indicating a significant positive relationship between breaking changes and compilation errors caused by these changes, as expected.

To investigate further how many errors each breaking change introduces, we perform the following regression analysis:

$$\ln(NE)_i = \beta_1 \ln(NBC)_i + \varepsilon_i$$

with $NE$ being the number of errors in $S_x$ and $NBC$ being the number of breaking changes in $\Delta L_{y,y+1}$. We do not estimate a constant since each error must be caused by a breaking change. Both $NE$ and $NBC$ are log-transformed because the data is lognormally distributed. The results can be found in Table 3.8. The model is highly significant with a $p$-value of 0 and an adjusted $R^2$ of 88.79%. The estimated slope coefficient of NBC is 1.683, indicating that if the number of breaking changes increases by 1%, the number of errors is expected to increase by 1.683%.

![Table 3.8: Regression analysis to estimate the relationship between breaking changes and errors.](image)

3.5.7. Average Amount of Work With and Without Breaking Changes

To further investigate the relationship between edit script size and breaking changes in libraries, we calculate the mean edit script size per method for library updates with and without breaking changes. We use the 3,260 library updates which contain breaking changes as described in Section 3.4, but due to missing data, only 2,106 systems can be used in this analysis. We denote the average edit script size in this set as $\langle \mu_{bc} \rangle$, which we compare to the average edit script size in the entire Maven repository regardless of breaking changes, denoted as $\langle \mu_{maven} \rangle$. The edit script size is divided by the number of methods in $L_{y+1}$ to correct for the effect of library size. We compare these means to find out if the amount of work in library updates with breaking changes is comparable to the amount of work performed in general.

There are three possibilities:
1. $\mu_{bc} < \mu_{maven}$: A library update containing breaking changes contains less work as compared the work done in the average library release. This may be caused by the fact that fixing breaking changes requires rework in the library itself, which may interfere with other work performed in that update.

2. $\mu_{bc} \approx \mu_{maven}$: The average amount of work done in library updates which include breaking changes is not significantly different from work done in releases in general.

3. $\mu_{bc} > \mu_{maven}$: A developer performs more work in a library update that contains breaking changes than in library releases in general: breaking changes are more frequently introduced in bigger updates.

To compare the means between these two groups, we perform an ANOVA analysis, of which the results are shown in Table 3.9.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$\mu$</th>
<th>$\sigma$</th>
<th>freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,106 systems with breaking changes</td>
<td>0.657</td>
<td>4.055</td>
<td>2,106</td>
</tr>
<tr>
<td>Entire Maven repository</td>
<td>0.376</td>
<td>3.500</td>
<td>24,565</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SS</th>
<th>df</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between groups</td>
<td>40.99</td>
<td>1</td>
</tr>
<tr>
<td>Within groups</td>
<td>4,154.3</td>
<td>26,669</td>
</tr>
<tr>
<td>Total</td>
<td>4,195.30</td>
<td>26,670</td>
</tr>
</tbody>
</table>

Table 3.9: ANOVA analysis to compare the average edit script size in library updates in the entire Maven repository and library updates with breaking changes in dependencies.

The analysis is significant with $F = 12.16$ and a $p$-value of 0, indicating that there exists a significant difference in the amount of work performed in library updates with breaking changes and library updates in general. The analysis contains 24,565 libraries from the Maven repository and 2,106 libraries with at least one breaking change since its previous version. The mean edit script size per method of the Maven repository group is 0.376 and the mean for the 2,106 systems is 0.657. This means that for two systems with 100 methods, the edit script size for a system with breaking changes in library updates will be 65.7 and the edit script size for a library update in general will be 37.6, which is a difference of approximately 75%. The ANOVA analysis indicates that there exists statistical support for the third scenario, $\mu_{bc} > \mu_{maven}$, which means the average edit script size per method tends to be larger for library updates with breaking changes than for library updates in general. This means that breaking changes occur in library updates where a relatively large amount of code is changed. This could indicate that developers pay less attention to backward compatibility when they work on a large library update.

To answer RQ1: Breaking changes are widespread, even in non-major releases. Client libraries actually rely on the changed functionality, leading to compilation errors when using a newer release of a library. Library developers tend to disregard backward compatibility when working on larger library updates.
3.6. Semantic Versioning Adherence Over Time

To find out if the adherence to semver has changed over time, we plot the number of major, minor and patch releases through time and the number of releases containing breaking changes over time. This plot is shown in Figure 3.10.

The figure shows that the ratio of major, minor and patch releases is relatively stable and around 15%, 30% and 50%, respectively. The percentage of major releases per year seems to decrease slightly in later years.

Regardless of release type, one in every three releases contains breaking changes. This percentage is relatively stable but slightly decreasing in later years. One out of every four releases violates semver (“breaking if non-major”), but this percentage also slightly decreases in later years: from 28.4% in 2006 to 23.7% in 2011.

To answer RQ2: The adherence to semantic versioning principles has increased over time with a slight decrease of breaking changes in non-major releases from 28.4% in 2006 to 23.7% in 2011.

3.7. Update Behavior

The key reason to investigate breaking changes is that they complicate upgrading a library to its latest version. To what extent is this visible in the Maven Dependency Dataset? What delay is there typically between a library release and the usage of that release by other systems? Is this delay affected by breaking changes?

To investigate the actual update behavior of systems using libraries, we collected all updates from the Maven repository that update one of their dependencies. Thus, we investigate usage scenarios within the maven dataset.
We obtained a list of 2,984 updates from the Maven repository of the form \((S_x, S_{x+1}, L_y, L_{y+1})\) where \(L\) is a dependency of \(S\) which was updated from version \(y\) to version \(y+1\) in the update of \(S\) from \(x\) to \(x+1\). For example, when the Spring framework included version 3.8.1 of JUnit in version 2.0, but included version 3.8.2 in version 2.1, Spring framework performed a minor update of JUnit in a patch release.

<table>
<thead>
<tr>
<th>Update</th>
<th>Major</th>
<th>Minor</th>
<th>Patch</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major</td>
<td>543</td>
<td>189</td>
<td>82</td>
<td>814</td>
</tr>
<tr>
<td>Minor</td>
<td>651</td>
<td>791</td>
<td>227</td>
<td>1,669</td>
</tr>
<tr>
<td>Patch</td>
<td>150</td>
<td>54</td>
<td>297</td>
<td>501</td>
</tr>
<tr>
<td>Total</td>
<td>1,344</td>
<td>1,034</td>
<td>606</td>
<td>2,984</td>
</tr>
</tbody>
</table>

Table 3.11: The number of updates of different types of \(S\) and simultaneous updates of dependency \(L\).

Table 3.11 shows the number of updates of different types of \(S\) and \(L\) in the Maven repository. The table shows that most major updates of dependencies (543) are performed in major updates of \(S\), and most minor updates of dependencies (791) are performed in minor updates of \(S\). The same is true for patch updates of dependencies, which are most frequently updated in patch updates of \(S\) (297).

Figure 3.12: An example of a timeline with a system \(S\) updating library \(L\).

To further investigate update behavior of dependencies, we calculate the number of versions of \(L\) that \(S\) lags behind, as illustrated in Figure 3.12. The figure shows an example of three versions of \(S\), and a dependency \(L\) of \(S\). On January 1, \(L_1\), a patch update, is released. \(S_1\) decides to use this version in its system. On March 1, a major update of \(L\) is released, \(L_2\). The next release of \(S\), \(S_2\), happens on April 1. This release still includes \(L_1\), although \(L_2\) was already available to include in \(S_2\). The same is true for \(S_3\), which could have included \(L_3\) but still includes \(L_2\). The period that \(S\) has been using \(L_1\) is from February 1, to April 1. The total time that \(S\) has a dependency on \(L\) is from February 1 to August 1.
This example illustrates that there can exist a lag between the release of a new version of $L$ and the inclusion in $S$. In this example, $S_3$ lags one minor release behind, and could have included $L_3$. The time $S_3$ theoretically could update to $L_3$ is between May, 1 and August, 1.

For each system $S$ and each of its dependencies $L$, we calculate the number of major, minor and patch releases that version of $S$ lags behind. The release dates of $S_x$ and $L_y$ are used to determine the number of releases after $L_y$ but before $S_x$.

Table 3.13 shows percentiles for the number of major, minor and patch versions that dependencies $L$ of system $S$ are lagging as compared to the latest releases of $L$ at the release date of $S$. For instance, when a system released a new version at January 1, 2013 and that release included a library with version 4.0.1 but there have been 10 minor releases of that library before January 1 and after the release date of version 4.0.1 that could have been included in that release of $S$, the number of minor releases lagging is 10 for that system-library combination. These numbers are calculated for each system-library combination separately.

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>p25</th>
<th>p50</th>
<th>p75</th>
<th>p90</th>
<th>p95</th>
<th>p99</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>Minor</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>101</td>
</tr>
<tr>
<td>Patch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>13</td>
<td>46</td>
</tr>
</tbody>
</table>

Table 3.13: Percentiles for the number of major, minor and patch dependency versions lagging.

The table shows that the number of major releases that $S$ lags on average tends to be smaller than the number of minor and patch releases lagging. The distributions are highly skewed, with a median of 0 for all three release types and a 75th percentile of 1 for minor and patch releases, indicating that the majority of library developers include the latest releases of dependencies in their own libraries. The numbers also indicate that developers tend to better keep up with the latest major releases than with minor and patch releases, as indicated by the 90th percentile of 1 for major releases and a 90th percentile of 5 for patch releases.

To better understand the reasons underlying the update lag, we investigate two properties of libraries that could influence the number of releases that systems are lagging: the edit script size and the number of breaking changes of these dependencies. We hypothesize that people are reluctant to update to a newer version of a dependency when it introduces a large number of breaking changes or introduces a large amount of new or changed functionality. To test this, we investigate whether a positive correlation exists between the number of major, minor and patch releases lagging in libraries using a dependency and the number of breaking changes and changed functionality in new releases of that dependency. We calculate Spearman correlations between the number of versions lagging and the number of breaking changes and edit script size in these versions.

The results are shown in Table 3.14. The table shows Spearman correlations, which are calculated on 13,945 observations and all have a $p$-value of 0. The correlations are generally very weak, with the maximum correlation being 0.1440 between the number of minor versions lagging and the number of breaking changes in these versions.
Table 3.14: Spearman correlations between the size of the update lag of $L$ and breaking changes and the edit script size in the next version of $L$.

<table>
<thead>
<tr>
<th></th>
<th>Breaking changes</th>
<th>Edit script size</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major versions lagging</td>
<td>0.0772</td>
<td>-0.0701</td>
<td>-0.0465</td>
</tr>
<tr>
<td>Minor versions lagging</td>
<td>0.1440</td>
<td>0.1272</td>
<td>-0.0434</td>
</tr>
<tr>
<td>Patch versions lagging</td>
<td>0.0190</td>
<td>0.0199</td>
<td>0.3824</td>
</tr>
</tbody>
</table>

The numbers indicate that, in general, people are more reluctant to update major, minor and patch releases with a larger number of breaking changes, but the effects are very small. Alternatively, one could argue that people tend to ignore breaking changes and changed functionality in new versions of dependencies, perhaps because they do not even know a priori whether a release introduces breaking changes. Thus, there exists a lag in these dependencies, regardless of breaking changes or changed functionality.

The correlation between the edit script size and the number of major versions lagging is even negative with a value of -0.0701, which indicates that major library versions with a larger amount of new or changed functionality are generally included slightly faster than releases with less changed or new functionality. The correlation between the number of breaking changes and the edit script size and the number of patch versions lagging is negligible with values of 0.0190 and 0.0199, with significant p-values.

The results indicate that although the number of breaking changes and the edit script size of a library does seem to have some influence on the number of library releases systems are lagging, the influence generally is not very large.

### 3.8. DEPRECIATION PATTERNS

As we have seen, breaking changes are common. To deal with breaking changes, the Java language offers deprecation annotations. For the use of such annotations, semantic versioning provides the following rules for deprecation of methods in public interfaces:

>a new minor release should be issued when a new deprecation tag is added. Before the functionality is removed completely in a new major release, there should be at least one minor release that contains the deprecation so that users can smoothly transition to the new API.\(^{20}\)

Thus, whenever there is a breaking change (which must be in a major release), this should be preceded by a deprecation (which can be in a minor release).

In this section, we investigate whether this principle is followed in practice. We investigate how many libraries actually deprecate methods, and if they do, how many releases it takes before these methods get deleted, if at all. We also find out if there is indeed at least one minor change in between before the method is removed, as \texttt{semver} prescribes.

Table 3.15 shows different possible deprecation patterns. The table uses a typical library with 4 releases (two major, two minor). For each pattern in the table, we count its frequency in the maven data set. As the table shows, there are a couple of different ways to deprecate and delete methods in major or minor releases, some of which are correct according to \texttt{semver} (column \(c\)).

Cases 1 and 2 in Table 3.15 show an example of a private method with and without
3.8. Deprecation Patterns

<table>
<thead>
<tr>
<th>#</th>
<th>v1 (maj.)</th>
<th>v2 (min.)</th>
<th>v3 (min.)</th>
<th>v4 (maj.)</th>
<th>c</th>
<th>i</th>
<th>Freq.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pr m1</td>
<td>pr m1</td>
<td>pr m1</td>
<td>pr m1</td>
<td>y</td>
<td>n</td>
<td>63,698</td>
<td>24.34</td>
</tr>
<tr>
<td>2</td>
<td>pr m2</td>
<td>pr m2</td>
<td>pr @d m2</td>
<td>pr @d m2</td>
<td>y</td>
<td>n</td>
<td>113</td>
<td>0.04</td>
</tr>
<tr>
<td>3</td>
<td>pu m3</td>
<td>pu m3</td>
<td>pu m3</td>
<td>pu m3</td>
<td>y</td>
<td>n</td>
<td>110,613</td>
<td>42.27</td>
</tr>
<tr>
<td>4</td>
<td>pu m4</td>
<td>pu @d m4</td>
<td>pu @d m4</td>
<td>pu @d m4</td>
<td>y</td>
<td>y</td>
<td>793</td>
<td>0.30</td>
</tr>
<tr>
<td>5</td>
<td>pu m5</td>
<td>pu m5</td>
<td>-</td>
<td>-</td>
<td>n</td>
<td>y</td>
<td>86,449</td>
<td>33.03</td>
</tr>
<tr>
<td>6</td>
<td>pu m6</td>
<td>pu @d m6</td>
<td>-</td>
<td>-</td>
<td>n</td>
<td>y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>pu m7</td>
<td>pu m7</td>
<td>pu m7</td>
<td>pu @d m7</td>
<td>n</td>
<td>y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>pu m8</td>
<td>pu @d m8</td>
<td>pu @d m8</td>
<td>-</td>
<td>y</td>
<td>y</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>pu m9</td>
<td>pu @d m9</td>
<td>pu m9</td>
<td>pu m9</td>
<td>n</td>
<td>y</td>
<td>16</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 3.15: Possible method deprecation patterns. @d = deprecated tag, c = correct, i = interesting; pr = private; pu = public; – = method deleted.

deprecation tags. As the table shows, the first case occurs in 24.24% of all methods. Since semver is only about versioning and changes in public interfaces, these cases are therefore not investigated further. Case 3 shows a public method that is neither deleted nor deprecated, which is the most common life cycle for a method (42% of the cases). Case 4 shows a public method that is deprecated, but is never removed in later versions. According to the principles regarding deprecation as stated in semver, this is correct behavior. As the table shows, this is the most common use of the deprecation tag, even though it is used in just 793 methods. Case 5 shows a public method that is removed from the interface but never declared deprecated, which is not correct: This is the typical case of introducing a breaking change in a minor release. Case 6 deprecates the method, but deletes it in a minor release, which would not be correct. This case does not occur. Case 7 declares the method deprecated in a major release, which would also be incorrect (and which does not occur). Case 8 shows an example of deprecation by the book, exactly as prescribed by semver. The method is declared deprecated in a minor release, there is another minor release that also declares the method deprecated and in the next major release, the method is removed. This correct pattern does not occur at all in the Maven repository. Case 9 shows a method that is undeprecated, about which semver does not explicitly contain a statement.

As the table further shows, public methods without a deprecated tag in their entire history are in the majority with 42.27%. Surprisingly, the number of public methods that ever get deprecated in their entire history is only 793, or 0.30%. The number of public methods that get deleted without a deprecated tag is 86,449, or 33.03%. The number of methods that get deleted after adding a deprecated tag to an earlier version is 0 (cases 6 and 8). Finally, the number of methods that get “undeprecated” is 0.01%.

These results are surprising since they suggest that developers do not apply deprecation patterns in the way that semver proposes. In fact, developers do not seem to use the deprecated tag for methods very often at all. Most public methods get deleted without applying a deprecated tag first (case 5), and methods that do get a deprecated tag are almost never deleted (case 4). This suggests that developers are reluctant to remove deprecated functionality from new releases, possibly because they are afraid to break backward compatibility. Case 8 is, according to semver, the only proper way to deprecate and delete methods. However, the pattern was not found in the entire Maven
repository.

To answer RQ4: *Developers do not follow deprecation guidelines as suggested by semantic versioning. Most public methods are deleted without applying a deprecated tag first, and when these tags are applied to methods, these methods are never deleted in later versions.*

### 3.9. Discussion

The results of this study indicate that the stability of interfaces and mechanisms to signal this instability to developers leaves much to be desired. One in every three interfaces contains breaking changes, and additionally, one in three interfaces that should not contain breaking changes actually does. The usage of the deprecation tag and the deletion of methods in the Maven repository show that the average developer tends to disregard the effects his actions have on clients of a library.

#### 3.9.1. Signaling Interface Instability

Our results show that developers do not tend to follow the best practices encoded in **semver**, even though the used versioning schemes suggest a semantic pattern. If developers would adhere completely to **semver** and their releases contain the same amount of breaking changes as found in the Maven repository, the number of major releases should be much larger than is currently the case. This low adherence is surprising since there are no other mechanisms available, except versioning schemes and deprecation tags, which signal interface instability. Possible explanations are that library developers are not aware of existing semantic versioning practices, they are not aware that they have introduced breaking changes, they do not expect that the changes they make have actual impact on client systems, or they simply do not care. Either way, we argue that the principles set out by **semver** should be followed by every developer of software libraries, or any piece of software of which the interface is used by other developers.

In our opinion, ultimately, better designed and more stable interfaces leads to a lower maintenance burden of software in general. When a library user, or a user of any piece of publicly available functionality knows that there are expected changes when upgrading to a newer version, the developer can anticipate this and choose to postpone or include the update. Strict adherence to semantic versioning principles also forces library developers to think hard about the functionality they release, and about the design of the public interface they are releasing. It is increasingly hard for library developers to change their overall design of their interface after it has been published. This problem becomes worse the more users actually use the interface. Releasing a new major release can effectively signal that continuity of the old interface should not be expected and that radical changes may be present. However, when this mechanism is only partially used, which we have shown is the case in the Maven repository, it becomes unclear what exactly a major release means.

Another explanation for the lack of discipline in interface versioning is that the Java modularization mechanism is not suited to provide all visibility levels as desired by developers. For instance, developers sometimes release “internal” packages. These are packages that should be hidden from outside developers and are only meant to be used
by the developers themselves. The problem with internal packages is that they are publicly visible, meaning that outside developers have complete access to these packages, just like regular packages. What is missing from the Java language is another layer of visibility, which hides internal packages from outside users. An example of a mechanism that does provide this level of visibility is the modularization structure of the OSGi framework. Additionally, entire libraries are sometimes released that are only meant to be used by the developers themselves, even without the use of internal packages. Java or the Maven repository also do not provide support to prevent external users from using these libraries. In fact, these libraries should have never been released in the Maven repository to begin with.

The low number of methods that use the deprecation tag in the entire Maven repository was surprising. A possible explanation for this is that classes can also be deprecated completely, without individually deprecating all methods in that class. Our analysis will not detect these cases. Future work could further investigate whether developers deprecate entire classes instead of deprecating only single methods.

3.9.2. OTHER VERSIONING STANDARDS

Semantic versioning is not the only standard for versioning software libraries. For instance, the OSGi alliance has released their own semantic versioning manifesto\(^{21}\) and contains comparable guidelines as the ones in `semver`. Furthermore, there exist several alternative versioning approaches\(^{22}\), but the versioning schemes described in these approaches do not seem to be used in the Maven repository, as can be seen in Table 3.2. For this reason, only adherence to the principles stated by `semver` was checked in this chapter.

In future work, the adherence to `semver` in libraries that use the OSGi framework could be investigated. We expect that the adherence to `semver` is higher in packages that use OSGi since OSGi provides an additional layer of visibility which would prevent counting breaking changes in internal packages.

3.9.3. ACTUAL USAGE FREQUENCIES

In this chapter, we do not take into account the difference between internal and non-internal packages. We also do not take into account the actual usage of packages, classes and methods with breaking changes. It makes a difference whether a public method in the interface of a library is used frequently by other developers, such as `AssertEquals` in JUnit, or the method is not used at all by other developers. However, semantic versioning principles generally do not state that breaking changes in major releases can only occur in parts of the library that are never used, but instead states that breaking changes should never be present in minor and patch releases, regardless of actual usage. The same is true for breaking changes in internal and non-internal packages.

In the metrics introduced in the next chapter, we incorporate actual usage frequencies in our metrics.


3.9.4. Release Interval and Edit Script Size

Table 3.5 showed that major releases have smaller release intervals and also contain less functional change. We expected that major releases have larger release intervals instead. This could be explained by the fact that developers often start working on a major release alongside the minor or patch release (by creating a branch) of the previous version, which would decrease the actual release interval.

The table also shows that major releases generally contain less changed functionality than minor releases, as measured by edit script size. A possible explanation for this is that developers create a new major release especially for backward incompatible changes in its API, and new functionality is added later. Seen this way, a major release can be interpreted as a signal that gives information on significant changes in the interface of a library, while saying nothing about the amount of changed functionality in the release.

3.9.5. Major Version 0 Releases

Semver states that “Major version zero (0.y.z) is for initial development. Anything may change at any time. The public API should not be considered stable.” We did not consider whether the effects as tested in this chapter also hold for releases with a major version of zero. The number of releases having a major version of 0 is 10.44% (13,162 / 126,070), which is a substantial part of all releases. Future work could investigate whether the principles as tested in this chapter are also visible in releases with a major version of 0. We expect that the number of breaking changes in these releases will be considerably higher than other releases.

3.10. Threats to Validity

3.10.1. Release Dates

The release dates of libraries as obtained from the Maven repository are sometimes incorrect, as demonstrated by the disproportionately large number of libraries with a release date of November 5th, 2005 (2,321, 1.5%). These data points were excluded from our analysis, but we do not have absolute certainty of the correctness of the remaining release dates. Another indication that release dates were not always correct is the fact that an ordering based on release dates and an ordering based on version numbers of a single artifact does not always give the same rankings. In these cases, the ordering in version numbers was assumed to be correct. These possibly invalid data points influence our analysis on the number of days between releases, but we assume that our statistical analyses provides us with a robust average.

3.10.2. Version Strings

We only investigated the changes in subsequent library versions which both have a “proper” version string, i.e. a specified major, minor and patch release number. When a prerelease string was included in the version number, no analysis was performed on the number of breaking changes since semver does not state whether prereleases can contain breaking changes. This does not introduce a bias in our study since we want to test whether libraries that do have a proper versioning scheme adhere to semver.
3.10. Threats to Validity

Not all subsequent versions of methods could be recognized while scanning for the deprecation patterns in Section 3.8. Library versions were parsed separately, leaving the problem that different objects representing the same method in different versions should be connected with each other. For performance reasons, this was done by text matching of method names and the number of parameters. Overloaded methods with the same number of parameters were not taken into account in this analysis. Future work could further investigate whether deprecation patterns are different for methods with overloaded versions with the same number of parameters.

3.10.3. Deprecation Tags

The low number of deprecation tags detected in the Maven repository is surprising. To make sure all deprecation tags were recognized, we scanned these tags in two different ways. First, a textual search was performed to search for literal occurrences of the string “@Deprecated”. Second, when a deprecated tag was found in a library, the complete library was parsed and and AST’s were created. This approach therefore makes it impossible to miss a deprecated tag. In future work, we could further investigate causes for the low number of deprecated tags.

3.10.4. External Validity and Generalizability

Our findings are based on an exploration of semantic versioning principles in the Maven repository. It is unknown whether the results can be reproduced in other software repositories mentioned before, such as NuGet, OSGi bundles, Ruby gems23, or, for example, the GitHub repository. We have already seen that NuGet has a different approach to update dependencies than Maven, but how often this actually introduces breaking changes with compilation errors is unknown. As mentioned before, other guidelines similar to semver have been released, so adherence to these guidelines can be investigated in a similar way as done in this chapter. Further research is needed to determine whether the patterns as found in this chapter hold in (industrial) sofware systems instead of open-source software libraries.

Future work could investigate to what degree the patterns found in our dataset are representative for software libraries outside the Maven repository, software libraries written in other languages than Java or software systems in general. To test our hypothesis that other library repositories also show the same patterns, further research is needed. Future work could also replicate the same patterns in a set of industrial software systems.

There was substantial computing power involved to obtain data for this chapter: data was obtained on a supercomputer with 100 processing nodes with an aggregated running time of almost six months. Without access to the same amount of computing power, the data will be very hard to reproduce. The data is nevertheless available for download24.

23http://www.rubygems.org
24See Chapter 2 for the download location.
3.11. CONCLUSION

In this chapter, we have looked at versioning principles as adopted by over 100,000 open source libraries distributed through Maven Central. We investigated to what degree versioning schemes as used by library developers provide library users with signals about backward incompatible changes in that release. One of such versioning schemes, semantic versioning, provides developers with a clear set of rules regarding the use of major, minor and patch version numbers, and we have tested these rules on our dataset.

Our findings are as follows:

- The introduction of breaking changes is widespread: Around one third of all releases introduce at least one breaking change. We see little difference between major and minor releases with regards to the number of breaking changes: One third of the major as well as one third of the minor releases introduce at least one breaking change (RQ1).
- The number of breaking changes in non-major releases has only decreased marginally over time (RQ2).
- Updates of dependencies to major releases are most often performed in major library updates, and similarly for updates of minor and patch releases and dependencies. Major releases of dependencies tend to take longer to be upgraded than minor or patch releases. There exists a small influence of the number of backward incompatibilities and of the amount of change in new versions on this lag (RQ3).
- Developers do not follow deprecation guidelines as suggested by semantic versioning. Most public methods are deleted without applying a deprecated tag first, and when these tags are applied to methods, these methods are never deleted in later versions (RQ4).

We can conclude that in general, developers spend little effort to communicate backward incompatibilities or deprecated methods in releases to users of their libraries. Although one can argue that not all developers may be aware of semantic versioning principles, we have assumed that most developers are aware of the intent of these principles: providing information about the amount of work done in a release and providing information about the stability of the interface of the library. We therefore argue that semantic versioning principles should be embraced more widely by the developer community.
4

MEASURING INTERFACE INSTABILITY

As we have demonstrated in the previous chapter, interface instability is a frequently occurring phenomenon in the Maven repository. To further investigate interface instability, we propose new metrics to measure this instability in more detail. We start this chapter by providing a real-world case study of a commercial company which demonstrates several issues associated with interface instability and backward incompatibilities. Next, we introduce a set of new metrics to assign a score to the stability of the interface of a library. We do this by counting the number of new methods in a library release as compared to its previous version, the amount of change in existing methods, the ratio of change in old methods compared to change in new ones and the percentage of new methods in each snapshot. We use the usage frequency of library methods by other libraries as weights for our metrics to give frequently used methods more impact in the final score. These weights are obtained from more than 2,300 snapshots of 140 industrial Java systems. We finally describe three scenarios and an example of the application of our metrics.¹

4.1. INTRODUCTION

Backward compatibility is a major concern for any library developer. If a new version of a library introduces breaking changes, then system developers are either forced to update their system to work with the new version or they must keep using the old version of the library (for a visual example, see Figure 4.1). Library developers, on the other hand, want to release new versions of their software to include new features, improve existing ones or fix bugs. Library developers are constantly faced with a trade-off between keeping backward compatibility and live with mistakes from the past or start over and introduce breaking changes, but at the expense of a loss of backward compatibility. A good library

¹Parts of this chapter have been published as “Measuring software library stability through historical version analysis”, ICSM 2012 [90].
should ideally be built in such way that the public interface is never broken, but this may prove to be impossible in practice.

An example of a company that spends great effort to keep their Application Programming Interface (API) backward compatible is Microsoft with its Windows SDK, which is used by all Windows programmers. Today, many old software systems still run on the latest version of Windows, thanks to a high degree of backward compatibility. Design decisions made in the early phases of the SDK, including mistakes, still impact developers using this interface today.

Complete API stability may be hard to achieve, and in practice, different libraries can have different degrees of API stability. There are several different properties of libraries which could indicate this stability. For instance, if the number of parameters of a library method changes, library users have no choice but to change each place where a call to this method occurs. When a public method is removed in a next version of a library, developers are also required to remove all calls to this method. More subtle are internal implementation changes while method interfaces are being kept constant, but even those changes could have an impact on systems using these libraries, since behavior could change in an undesired way.

Our goal is to introduce new metrics to measure interface and implementation stability. To that end we analyze historical versions of software libraries, obtain ratings for these versions and weigh these ratings by the number of times methods, classes or packages are being used. To collect data on dependencies used in industrial systems we create an infrastructure to extract third-party library dependencies for Java as defined in Maven build files. We apply our metrics to the most frequently used Apache Commons libraries and we give an example of the application of our metrics to a library, for instance to determine if a library is in a state of maintenance or active development.

In Section 4.2, we start with a motivating example of third-party library usage in a commercial company. In Section 4.3, the problem statement and a definition of library stability are given. In Section 4.4, we discuss related work in the field of API usage, migration and evolution. In Section 4.5, we describe our dataset used to calculate our library
4.2. Motivating Example

To describe the issues regarding dependencies on third-party libraries in large commercial and open source software systems, we give an example of a commercial company which uses several third-party libraries in a custom-developed web application of considerable size (approximately 4000 Java files and 200,000 lines of code). This application depends heavily on several libraries such as the Spring framework\(^2\), Apache Struts\(^3\) and Hibernate\(^4\). For confidentiality reasons, the system and company name cannot be provided.

When the application was first built in 2004, third-party library dependencies were managed using Maven. Version numbers of the latest versions which were available at that time were hard-coded in the configuration files of the project. These libraries were not updated to more recent version in the next 7 years, which resulted in a large “maintenance debt” of lagging versions. For instance, version 1.0 of the Acegi authentication and security framework (started in late 2003) was being used while this library was included in the Spring framework and was renamed “Spring Security” 2.0.0 in 2008. In the meantime, several breaking changes were introduced in new versions of the Spring Security framework as well as critical safety-related bug fixes and improvements.

Due to expected compatibility issues when upgrading Acegi, this update was deferred as long as possible. In the old setup, user authentication was handled through the Acegi library which communicated with an LDAP authentication server. To improve authentication and to facilitate single sign-on, Atlassian Crowd was contracted, a web-based authentication service. However, the Acegi framework was not capable of communicating with Atlassian Crowd and Acegi therefore had to be replaced. Spring Security was chosen as a natural successor although the library was rewritten from scratch and several breaking changes were introduced. The latest version available during the update process was 3.0.6, which changed significantly from Acegi 1.0.

Since Spring Security 3.0.6 is part of the more general Spring Framework, the entire Spring Framework had to be updated too. Considering the large dependence of the application on this framework, this would mean that a large part of the application had to be adapted to work with this new version. Since the Struts framework was also used and the new version of the Spring framework could not work with the old version of Struts, this had to be upgraded as well. Java code had to be adapted due to the transformation from Acegi to Spring Security. Also, since the syntax of the expression language used in Java Server Pages (JSP) was changed between Struts 2.0.9 and 2.2.3.1, all web pages in which dynamic content was presented using JSP had to be updated with the new syntax.

\(^2\)http://projects.spring.io/spring-framework  
\(^3\)https://struts.apache.org  
\(^4\)http://hibernate.org
Eventually, a week was spent to implement the changes and upgrades. There was a test suite available, both in the form of unit tests written in Java and automated browser interface tests created with Selenium. Developers working on the system commented that without this test suite the impact of such an update would be much harder to assess.

This case illustrates several issues with third-party library dependencies. First, it shows the accumulation of maintenance debt when deferring updates of libraries. Second, it shows that there may come a moment in the future in which there is no choice but to update to a new version in which case a much larger effort has to be put in than in the case of smaller incremental updates. Third, it shows a case of a library that disregards backward compatibility and introduces breaking changes: the Struts library changed JSP expression language syntax which would require a large rework effort in systems using this syntax. Fourth, it also shows that transitive dependencies of included libraries can increase the total amount of work required to update to a new version of a library, even if an upgrade of these transitive dependencies was originally not intended. Finally, it shows the risk of using deprecated and legacy versions of libraries which can contain security weaknesses or critical bugs.

4.3. **Problem statement**

The example in the previous section shows that there are several issues involved with the usage of third-party libraries in general. Not every library is maintained with the notion of backward compatibility in mind, while any developer working with a third-party library immediately notices any change made to its public interface. The less library developers ensure continuity and stability of its public interface, the harder upgrading to the latest version of a library becomes. This poses a challenge for developers of an API, who have to think carefully about the public parts of an interface they publish. After releasing an API it is very difficult to make large-scale changes to it since other developers count on already released parts, and changing only even a small part of this interface will require developers to adjust their implementations. Since requirements keep changing and systems keep evolving over time, designing the correct interface that is stable and backward compatible enough in subsequent releases but also flexible enough to adapt to changing requirements can be challenging.

We do not address all of these issues here, but we do provide a method to measure the stability of a library, which could cause problems as mentioned above. We consider an API to be stable if functionality is not removed from a public interface once it has been added. In the case of a Java system, this means that methods, classes or packages are not removed from the interface once they have been added and method signatures are not changed (adding, removing or changing parameters or renaming methods). We consider the implementation of a library to be stable with regard to a certain metric if metric values in a system are relatively constant through time.

The definition of library stability that we assume in this chapter is the following: **library (in)stability is the degree to which the public interface or implementation of a software library changes through time in such way that it potentially requires users of this library to rework their implementations due to these changes.**
4.4. **Related Work**

The problem of changing API interfaces has been recognized by other authors [32, 37, 54] and has been researched through different approaches than the one we take here. For instance, tools have been proposed to detect API evolution and to suggest refactorings to get up-to-date with the latest version of an API [54, 87]. Dagenais and Robillard [32] propose SemDiff, a tool that recommends replacements for framework methods that were accessed by a client program and deleted during the evolution of the framework.

Uddin et al [108] proposes a way to detect changes in the usage of an API, which differs from our work, in which we investigate changes over time in libraries themselves. The starting point of their data collection framework is similar to ours: they store client evolution patterns which represent a time series of changes which consists of added and removed methods calls to the API, with each snapshot having a time stamp. From this point on, however, Uddin et al take a different approach and investigate how to represent these client-side change patterns mathematically and how to infer temporal API usage patterns over time.

Other work in API usage mining often focuses on the detection of usage patterns of API methods in systems that call these methods. It can reveal how an API method is normally called and what preparing statements have been executed [115, 117]. Similarly, Thummalapenta [106] presents a framework for the detection of hotspots and coldspots in API's which can help to show relevant code examples in documentation of third-party libraries.

Furthermore, usage of certain libraries and specific methods has also been investigated. This research often shows usage statistics of certain parts of libraries (“hidden” or “public”) [50, 68, 80] or collects statistics on the frequencies of use for methods in the library. In our work we want to use this type of information to help in the decision to include a certain library in a software project, or to choose a better alternative if one is available.

4.5. **Dataset**

We use the Maven Dependency Dataset to calculate our metrics on. The Maven build system has the desirable property that it requires the specification of a version number when including a library dependency in a build file, which we can take advantage of in this analysis. This chapter focuses on one particular set of libraries: the Apache Commons libraries. In earlier work we have identified the Apache Commons libraries as the most commonly used libraries [89] in other software libraries, making it a suitable case study. Some descriptive characteristics of the Apache Commons libraries are provided in Table 4.1.

Since we are interested in the implications of stability on the actual use of a library, we need a set of systems making use of libraries. To that end, we use a collection of 2487 snapshots of 140 industrial Maven-based systems of which source code is available at the Software Improvement Group (SIG). The systems come from the same set we have used in earlier work [89]. Statistics on the use of the Apache Commons libraries by our set of subject systems are provided in Table 4.2. In Table 4.2, library versions that exist but are never included are omitted.
<table>
<thead>
<tr>
<th>Library name</th>
<th>LOC</th>
<th>Classes</th>
<th>#Mtds</th>
<th>#S</th>
<th>Latest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Commons Collections</td>
<td>26,323</td>
<td>422</td>
<td>3,945</td>
<td>6</td>
<td>Dec-’11</td>
</tr>
<tr>
<td>Apache Commons Lang</td>
<td>19,475</td>
<td>122</td>
<td>2,338</td>
<td>6</td>
<td>Jan-’11</td>
</tr>
<tr>
<td>Apache Commons HTTPClient</td>
<td>17,171</td>
<td>171</td>
<td>1,944</td>
<td>3</td>
<td>Aug-’07</td>
</tr>
<tr>
<td>Apache Commons Beanutils</td>
<td>11,375</td>
<td>127</td>
<td>1,284</td>
<td>5</td>
<td>Mar-’10</td>
</tr>
<tr>
<td>Apache Commons IO</td>
<td>8,086</td>
<td>100</td>
<td>1,053</td>
<td>7</td>
<td>Oct-’11</td>
</tr>
<tr>
<td>Apache Commons Codec</td>
<td>4,554</td>
<td>64</td>
<td>503</td>
<td>4</td>
<td>Nov-’11</td>
</tr>
<tr>
<td>Apache Commons Logging</td>
<td>26,80</td>
<td>27</td>
<td>311</td>
<td>3</td>
<td>Nov-’07</td>
</tr>
</tbody>
</table>

Table 4.1: Descriptive statistics of the Apache Commons libraries. #Mtds=Nr. of Methods, #S=Nr. of Snapshots

### 4.6. Analyzing Maven Dependencies

To obtain usage frequencies, Maven build files are scanned for third-party library dependencies. Each Maven project (pom.xml) file can contain a dependency section with information on the name and version of used libraries. For an example of a Maven dependency, see Figure 4.2. Our dataset contains multiple snapshots of systems on different points in time, each pointing to possibly different versions of third-party libraries. Each system and third-party library version is tagged with a version number and snapshot date.

Maven has a complex system to link the right versions of libraries to a deliverable, such as a jar file. This is not a trivial process since projects can contain multiple pom.xml files which can each point to different versions of the same library. Maven assures that only one version of a library is included inside a single deliverable.

```xml
<dependencies>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.1</version>
  </dependency>
</dependencies>
```

Table 4.2: An example of a dependency in a Maven build file.

Dependencies can be analyzed and viewed in an hierarchical representation using the Maven dependency plugin but this requires a fully compiling project. This in turn requires the complete set of all pom.xml files that are referenced in a project. In our dataset, the collection of pom files is often incomplete and therefore a reconstruction

---

5 [https://commons.apache.org/collections/](https://commons.apache.org/collections/)
6 [https://commons.apache.org/lang/](https://commons.apache.org/lang/)
8 [http://commons.apache.org/proper/commons-beanutils](http://commons.apache.org/proper/commons-beanutils)
9 [https://commons.apache.org/io/](https://commons.apache.org/io/)
10 [https://commons.apache.org/codec/](https://commons.apache.org/codec/)
has to be made which uses approximately the same rules as the resolving engine of Maven but can handle missing parent poms or otherwise incomplete references.

The Maven build file can contain dependency sections which contain a groupId, artifactId and version element (see Figure 4.2). The groupId, artifactId and version of a dependency together uniquely identify a certain library, which can be obtained through a public Maven repository. Specifying dependency versions was not required in older versions of Maven (before version 3) and versions of libraries are therefore often missing in pom files used with older versions of Maven. Also, a version can be mentioned in a “dependency management” section in a pom file higher in the directory hierarchy which specifies that a specific version of a library should be used, if it would ever be included.

Pom files are ordered hierarchically, in such way that a pom file present in a child directory overrules settings from a pom file present in the parent folder of that directory. Pom files can specify module dependencies in which only a selection of poms can be

<table>
<thead>
<tr>
<th>Library</th>
<th>Version</th>
<th>Date</th>
<th>Times used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons Beanutils</td>
<td>1.6.1</td>
<td>18-feb-'03</td>
<td>253</td>
</tr>
<tr>
<td></td>
<td>1.7.0</td>
<td>02-aug-'04</td>
<td>1776</td>
</tr>
<tr>
<td></td>
<td>1.8.0</td>
<td>01-sep-'08</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>1.8.2</td>
<td>13-nov-'09</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>1.8.3</td>
<td>28-mar-'10</td>
<td>67</td>
</tr>
<tr>
<td>Commons Codec</td>
<td>1.3</td>
<td>10-jul-'04</td>
<td>684</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>09-aug-'09</td>
<td>622</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>29-mar-'11</td>
<td>129</td>
</tr>
<tr>
<td></td>
<td>1.6</td>
<td>20-nov-'11</td>
<td>1</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>2.1</td>
<td>21-oct-'02</td>
<td>317</td>
</tr>
<tr>
<td></td>
<td>2.1.1</td>
<td>29-may-'04</td>
<td>98</td>
</tr>
<tr>
<td></td>
<td>3.0</td>
<td>25-jan-'04</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>3.1</td>
<td>23-jun-'04</td>
<td>609</td>
</tr>
<tr>
<td></td>
<td>3.2</td>
<td>14-may-'06</td>
<td>1863</td>
</tr>
<tr>
<td></td>
<td>3.2.1</td>
<td>07-dec-'11</td>
<td>1375</td>
</tr>
<tr>
<td>Commons HttpClient</td>
<td>2.0.2</td>
<td>10-oct-'04</td>
<td>237</td>
</tr>
<tr>
<td></td>
<td>3.0.1</td>
<td>07-may-'06</td>
<td>245</td>
</tr>
<tr>
<td></td>
<td>3.1</td>
<td>18-aug-'07</td>
<td>1053</td>
</tr>
<tr>
<td>Commons IO</td>
<td>1.1</td>
<td>10-oct-'05</td>
<td>308</td>
</tr>
<tr>
<td></td>
<td>1.2</td>
<td>19-mar-'06</td>
<td>425</td>
</tr>
<tr>
<td></td>
<td>1.3.1</td>
<td>13-feb-'07</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>1.3.2</td>
<td>02-jul-'07</td>
<td>73</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>21-jan-'08</td>
<td>1076</td>
</tr>
<tr>
<td></td>
<td>2.0.1</td>
<td>26-dec-'10</td>
<td>188</td>
</tr>
<tr>
<td></td>
<td>2.1</td>
<td>11-oct-'11</td>
<td>27</td>
</tr>
<tr>
<td>Commons Lang</td>
<td>2.1</td>
<td>12-jun-'05</td>
<td>1205</td>
</tr>
<tr>
<td></td>
<td>2.2</td>
<td>28-jul-'07</td>
<td>369</td>
</tr>
<tr>
<td></td>
<td>2.3</td>
<td>13-feb-'07</td>
<td>1475</td>
</tr>
<tr>
<td></td>
<td>2.4</td>
<td>19-mar-'08</td>
<td>1922</td>
</tr>
<tr>
<td></td>
<td>2.5</td>
<td>07-apr-'10</td>
<td>783</td>
</tr>
<tr>
<td></td>
<td>2.6</td>
<td>16-jan-'11</td>
<td>371</td>
</tr>
<tr>
<td>Commons Logging</td>
<td>1.0.4</td>
<td>10-jun-'04</td>
<td>1507</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>03-jun-'07</td>
<td>2299</td>
</tr>
<tr>
<td></td>
<td>1.1.1</td>
<td>22-nov-'07</td>
<td>1563</td>
</tr>
</tbody>
</table>

Table 4.2: The usage statistics of the Apache Commons libraries
Table 4.3: The process of Maven repository mining. Starting with separate dependency files, the result is a database with all metrics per snapshot including source code of dependencies.

included in a particular build target. Pom inheritance and module specification are two mechanisms which work independently of each other.

To reduce the complexity in resolving the right version number for each dependency, we make simplifying assumptions, which result in the selection of the latest version of a library when two or more different versions of the same library are included. The assumptions are the following:

- If no explicit version of a dependency is mentioned, the version as mentioned in the dependency management section of a hierarchically higher pom file is used. If there is no dependency management section present in any pom file in the snapshot higher in the hierarchy, the last library version before the release date of the project snapshot is obtained.
• The parent pom is always assumed to be in the parent directory of the child pom since it is a convention to put the parent pom in the parent directory.

• All poms present in a snapshot folder are assumed to be used inside a project. This is a safe assumption to make because poms that are present in a snapshot are most likely to be actually used inside that snapshot.

• When parent poms are missing, all independent child poms are assumed to be included in the same parent pom. This assumption is safe to make since inside a Maven project, a top-level pom usually exists which bundles a project together.

• When two siblings mention a different version of the same dependency, the latest version is included. When other conflicts between versions of the same dependency arise, the latest version is also chosen. When no version is available through any of the preceding rules, the dependency is ignored.

When other conflicts between versions of the same dependency arise, the latest version is also chosen. When no version is available through any of the preceding rules, the dependency is ignored. A graphical summary of an example system which includes multiple versions of JUnit is provided in Figure 4.4.

Since the dependencies section in a Maven dependencies file can contain both publicly available and internal dependencies, internal dependencies were removed by matching dependency names with project names. After this, results were manually checked to remove false positives. This resulted in a list of third-party library dependencies per snapshot for each system. Next, binary, source and JavaDoc jars of all collected dependencies were downloaded from the central Maven repository, if available. After downloading, all jar files were automatically extracted. For descriptive statistics of collected dependencies, see Table 4.2.

For each system snapshot, source code of the system and source code of all used third-party libraries were combined in a single folder. The Software Analysis Toolkit (SAT) of the SIG was used to calculate a wide range of metrics related to size in lines of code (LOC), complexity by means of the McCabe value as well as call graph information. The result of this process is a data file containing metrics on method-level for each
snapshot of systems and third-party library code that is called from this system. Package, class and method names are stored separately to make aggregation of metrics to these levels possible. By comparing versions and names of third-party library dependencies between snapshots it is possible to detect additions, removals, and version changes of these libraries.

In the next section we consider different variables to include in a library stability metric.

4.7. Metric Ingredients

There are several possible metrics and measurement methods that could be considered for inclusion in a metric of the stability of library implementation and interface. The criterion we maintain for a library stability metric is that it should be representative for the amount of work that is required when library developers update a certain library to a newer version. Included metrics should therefore have a rationale that is consistent with this criterion. Since we want to investigate the stability of both implementation and interface of a library we also consider metrics that are an indicator for the amount of implementation “churn”. We expect that even though these changes do not become visible at the public interface, there is still a potential amount of rework effort required due to a potential change in behavior.

Table 4.3 summarizes change characteristics for Commons Logging and Commons Collections. Commons Collections has more different snapshots and has more methods that are removed and deleted in each snapshot than Commons Logging and Commons Logging is therefore considered to be more stable.

<table>
<thead>
<tr>
<th>Library</th>
<th>S</th>
<th>Unique methods (diff)</th>
<th>Total McCabe (diff)</th>
<th>Total LOC (diff)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logging</td>
<td>1</td>
<td>263</td>
<td>454</td>
<td>1521</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>301 (+75, -37)</td>
<td>653 (+199)</td>
<td>2818 (+1297)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>311 (+11, -1)</td>
<td>667 (+14)</td>
<td>2918 (+100)</td>
</tr>
<tr>
<td>Collections</td>
<td>1</td>
<td>1504</td>
<td>2580</td>
<td>9969</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1398 (+0, -106)</td>
<td>2395 (-185)</td>
<td>9482 (-487)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3357 (+2218, -259)</td>
<td>5583 (+3188)</td>
<td>18199 (+8717)</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3821 (+568, -104)</td>
<td>6503 (+920)</td>
<td>20452 (+2253)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>3945 (+125, -1)</td>
<td>6749 (+246)</td>
<td>21207 (+755)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>3945 (+0, -0)</td>
<td>6749 (+0)</td>
<td>21207 (+0)</td>
</tr>
</tbody>
</table>

Table 4.3: Example of the differences in number of methods, total McCabe and total LOC for two Apache Commons libraries

4.7.1. Candidate Variables

The candidate variables to include are the following:

Unit removals

Units (methods, classes or packages in Java) that are being removed from a public interface require rework from the developers that call these units. Therefore, counting the number of removed units per snapshot is a good indicator for the stability of an API. We
detect renamed/moved units as units that are removed first and added later. This is acceptable since a unit’s rename or move also requires rework effort and can therefore be counted as a unit removal.

**UNIT ADDITIONS**
Units that are added to a library do not influence the stability of the existing interface directly but could serve as an indirect indicator of the amount of effort that is spent on extending the library. Examples of method additions and removals in two libraries can be found in Table 4.3.

**LOC an McCabe changes**
Changes in McCabe and LOC values serve as indirect measure for the amount of work going on in a library since implementation can change while an interface stays constant. It is also not possible to make a distinction between changes that alter external behavior and changes which do not alter behavior of the library (such as refactorings) by looking at the McCabe value or the LOC alone. We nevertheless believe that these metrics provide an indication of the amount of work performed in a certain library.

**PARAMETER CHANGES**
In principle, a change in the signature of an existing API method always requires rework when this method is used. However, we do not include parameter type and number changes because we expect that the frequency of such changes is too low to base a stability metric on. We will therefore ignore signature changes for the moment and focus on size-related measurements instead.

We incorporate these variables in our metrics as can be seen in Section 4.8. First, we discuss how we weight historical values of the same metric and how we incorporate usage information in our metrics.

### 4.7.2. Weighting Measurements Historically
Because multiple historic values of the same metric exist, a method is needed to aggregate multiple measurements in time to a single value. After calculation of the difference in each metric between two snapshots we obtain a set of absolute differences for each unit in a system. To reduce this set to a single number, we use a historic weighting scheme to put emphasis on more recent snapshots. For our analysis, we choose a geometric series to weight each snapshot: a metric in snapshot \(s\) (counting backwards) is weighted with \(1/2^{s-1}\). This reflects our belief that changes made more recent in time weight more heavily than changes made longer ago. This is illustrated in Figure 4.5.

### 4.7.3. Weighting Measurements via API Usage
Besides weighting each snapshot differently we also give weight to each unit in a snapshot, for which we use the frequency of use as a weight. This means that more frequently used units have more influence in the final metric. This way, changes in units that are called frequently are emphasized and have a greater influence on the final metric of a library than changes in methods that are never called. Section 4.9.1 shows an overview of these frequencies of use.
We assume that methods are part of the public interface of a library if they can be called from a system using this library, regardless of the mechanism used to ensure this. This means that we ignore the mechanism provided in Java to make a distinction between public, package-private, private and protected methods and classes and we use the actual usage of methods inside a corpus of industrial systems as a weight. Units that are called frequently will have more influence in the final metric than units which cannot be called externally, which receive a weight of 0.

In the next section, we present our metrics which incorporate previously discussed variables, usage information of libraries and our historical weighting scheme.

4.8. Metric Definitions

We propose 4 metrics, which are displayed in equations 7 to 10 and are defined as follows:

- **WRM**: The weighted number of removed methods
- **CEM**: The amount of change in existing methods
- **RCNO**: The ratio of change in new to old methods
- **PNM**: The percentage of new methods

The metrics can be explained as follows. The WRM uses the weighted number of removed methods from an interface as a value for \( R_x \) in equation 6, which is a measure for interface stability since the removal of units from a public interface becomes immediately visible for users of a library. The more times a removed method is being used the more it increases the WRM.

CEM gives an indication of the amount of change in existing methods. It can give an impression of the activity and “volatility” of the development of a library. It measures the amount of change in cyclomatic complexity (McCabe) between two versions of a library. It is calculated by summing over the differences between McCabe values for each method in both versions and weighting the result with the times each method is being used in a certain reference set. A high CEM value indicates a library version with
large amount of change (churn) in existing and frequently used methods compared to its previous version. Changes in cyclomatic complexity were preferred over changes in the number of lines of code because it is expected to better represent changes in actual complexity between versions.

RCNO uses the ratio in metric difference between new and existing units which can be used to determine the amount of work being performed in new units relative to old ones. This ratio is smaller than 1 if more work is being performed in old methods and is greater than 1 if more work has been done in new methods. It is useful for determining whether a system is in a state of maintenance or active development. Developers can only spend a limited amount of time per release on new features and maintenance and the time spent on one activity cannot be spent on the other.

PNM calculates the percentage of new methods that have been added in each snapshot and can provide information on the expansion rate of an API.

Equations 1 to 3 are functions which select appropriate units: \( U_o \) (old) is the collection of units that are in snapshot \( s \) and in snapshot \( s+1 \). Similarly, \( U_n \) (new) is the collection of units which are new in snapshot \( s+1 \) compared to snapshot \( s \). \( U_r \) (removed) are the units that are in \( s \) but not in \( s+1 \). Equation 5 states that the combined delta between two snapshots of all units is the difference between a metric value in snapshot \( s+1 \) and \( s \), weighted by the times each unit is being called. The result is normalized by total weight of units in \( \Delta U(s, s+1) \). \( \Delta U_n \) is the difference between 0 and the metric value for each new unit while \( \Delta U_o \) is the difference between the metric value in \( s \) and \( s+1 \). Equation 5 expresses the absolute weighted difference of each unit in snapshot \( s+1 \) compared to \( s \) per snapshot.

\[
U_o(s, s+1) = U_s \cap U_{s+1} \tag{4.1}
\]

\[
U_n(s, s+1) = U_{s+1} \setminus U_s \tag{4.2}
\]

\[
U_r(s, s+1) = U_s \setminus U_{s+1} \tag{4.3}
\]

\[
hw(s) = \frac{1}{2^{s-1}} \tag{4.4}
\]

\[
\Delta U(s, s+1) = \frac{1}{\sum_{u \in \Delta U} w_u} \sum_{u \in \Delta U} w_u |M_{u,s+1} - M_{u,s}| \tag{4.5}
\]

\[
R = \sum_{s=1}^{\lfloor |S| \rfloor} hw_s R_s \tag{4.6}
\]

\[
WRM = \sum_{u \in U} w_u U_r \tag{4.7}
\]

\[
CEM = \Delta U_o \tag{4.8}
\]
\[ \text{RCNO} = \frac{\Delta U_n}{\Delta U_o} \tag{4.9} \]

\[ \text{PNM} = \frac{|U_n|}{|U_o| + |U_n|} \tag{4.10} \]

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u)</td>
<td>A specific unit</td>
</tr>
<tr>
<td>(U)</td>
<td>The set of all units in a system</td>
</tr>
<tr>
<td>(s)</td>
<td>A specific snapshot number (snapshots are ordered on date, are numbered backwards and the latest snapshot gets number 1)</td>
</tr>
<tr>
<td>(S)</td>
<td>The set of all snapshots of a system</td>
</tr>
<tr>
<td>(U_s)</td>
<td>All units in snapshot (s)</td>
</tr>
<tr>
<td>(U_o) (old)</td>
<td>All units in (s+1) also in (s)</td>
</tr>
<tr>
<td>(U_n) (new)</td>
<td>All units in (s+1) but not in (s)</td>
</tr>
<tr>
<td>(U_r) (removed)</td>
<td>All units in (s) but not in (s+1)</td>
</tr>
<tr>
<td>(hw(s))</td>
<td>The historical weight of a snapshot</td>
</tr>
<tr>
<td>(M_{u,s})</td>
<td>The value of metric (M) of unit (u) at snapshot (s)</td>
</tr>
<tr>
<td>(w_u)</td>
<td>The weight of a unit (the total times the unit is used in our dataset)</td>
</tr>
<tr>
<td>(R)</td>
<td>Generic placeholder metric function</td>
</tr>
<tr>
<td>(R_x)</td>
<td>A specific metric (WRM, CEM, RCNO or PNM) to be used in the placeholder function</td>
</tr>
</tbody>
</table>

Table 4.4: Explanation of used symbols in equations 1 to 10

Eventually, all snapshots are combined in a single metric formula by summing over the separate metrics and giving less weight to snapshots further away in time. Ultimately, these metrics make it possible to aggregate values at the unit level to a single value at the system level while capturing changes through time and weighting for frequency of use and recency of snapshots. For an explanation of symbols used in equations 1 to 10, see Table 4.4.

4.9. Apache Commons Findings

With all metric definitions in place, we can analyze the metric values for Apache Commons in the context of our suite of 140 benchmark systems. We start by analyzing which library methods are used most frequently, followed by a discussion of the actual metric values.

4.9.1. Apache Commons API Usage

In Table 4.5, the most frequently called methods per library are shown. As can be seen in this table, the most frequently used method constitutes a large percentage of the total number of calls to a library. Also, only a small percentage of methods is actually called from our dataset. Most methods are used only once or not at all. This is also illustrated in Figure ??, which gives a visual impression of the spread and concentration of method
calls through each library. We use this information to determine the weight for each unit \( w_u \) in equation 5 in the previous section.

<table>
<thead>
<tr>
<th>Library</th>
<th>Method name</th>
<th># Calls</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons Logging</td>
<td>Log.info</td>
<td>15114</td>
<td>28.54%</td>
</tr>
<tr>
<td>Commons Lang</td>
<td>StringUtils.isNotEmpty</td>
<td>10033</td>
<td>10.75%</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>NotPredicate constructor</td>
<td>2461</td>
<td>30.14%</td>
</tr>
<tr>
<td>Commons Beanutils</td>
<td>DynaBean.get</td>
<td>875</td>
<td>30.71%</td>
</tr>
<tr>
<td>Commons IO</td>
<td>FileUtils.readFileToString</td>
<td>835</td>
<td>11.55%</td>
</tr>
<tr>
<td>Commons HttpClient</td>
<td>HttpClient.executeMethod</td>
<td>428</td>
<td>9.23%</td>
</tr>
<tr>
<td>Commons Codec</td>
<td>Base64.decodeBase64</td>
<td>124</td>
<td>30.77%</td>
</tr>
</tbody>
</table>

Table 4.5: The number of times the most frequently used method is called and the percentage of the total number of calls.

Table 4.6: The relative distribution of calls to methods of third-party libraries. A darker shade of grey means a certain method is called more. Methods are sorted on method and package name. This figure does not show method names or the exact number of times each method is called but gives a visual impression of the spread and relative usage of library methods. In parentheses is the total number of methods in that library.

4.9.2. METRICS
The results for the four metrics are shown in Table 4.6. An absolute value and a rank is provided for each metric. The metric values are dimensionless and range from 0 to infinity; smaller values indicate greater stability.

Table 4.6 shows that there are 4 systems with a value of 0 for WRM. This indicates that there have been no methods in those libraries which were used in our dataset and were removed from a next snapshot of the library. The WRM can be seen as an indicator for the absolute number of methods which will have to be adapted due to method removals from these libraries.

Since multiple systems have the same score for WRM, the value of CEM can provide additional information. Of all systems with a 0 for WRM, Commons Logging is the system with the lowest score for CEM (0.0124). This means that the amount of change
4. MEASURING INTERFACE INSTABILITY

<table>
<thead>
<tr>
<th>Library</th>
<th>WRM (Rank)</th>
<th>CEM (Rank)</th>
<th>RCNO (Rank)</th>
<th>PNM (Rank)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons Logging</td>
<td>0 (1)</td>
<td>0.0124 (2)</td>
<td>0 (1)</td>
<td>0.2669 (2)</td>
</tr>
<tr>
<td>Commons Beanutils</td>
<td>267.25 (3)</td>
<td>1.257 (7)</td>
<td>10.175 (3)</td>
<td>0.4931 (6)</td>
</tr>
<tr>
<td>Commons Codec</td>
<td>31 (2)</td>
<td>0.7 (6)</td>
<td>1.8833 (2)</td>
<td>0.5083 (7)</td>
</tr>
<tr>
<td>Commons HttpClient</td>
<td>0 (1)</td>
<td>0.1239 (4)</td>
<td>146.84 (4)</td>
<td>0.1937 (1)</td>
</tr>
<tr>
<td>Commons IO</td>
<td>0 (1)</td>
<td>0.0484 (3)</td>
<td>273.16 (5)</td>
<td>0.4281 (5)</td>
</tr>
<tr>
<td>Commons Lang</td>
<td>0 (1)</td>
<td>0.3456 (5)</td>
<td>481.09 (7)</td>
<td>0.3256 (3)</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>1062.0 (4)</td>
<td>0.0077 (1)</td>
<td>339.65 (6)</td>
<td>0.3715 (4)</td>
</tr>
</tbody>
</table>

Table 4.6: Values and ranks for the four metrics

in existing methods is relatively low in Commons Logging compared to other systems. Commons Logging also scores a 0 for RCNO, which can mean three things: (1) there is no difference in existing methods, (2) there is no difference in new methods (only empty method bodies have been created) or (3) all changes occur in methods which are never called. From this metric, it is not possible to distinguish between these cases, but further inspection showed that all metric differences occurred in methods which were never called. Commons Logging scores 0.2669 for PNM, which is the second place. This indicates that a relatively small number of new methods are added in each snapshot. Metrics should not be interpreted as percentages or amounts directly since snapshots have been weighted differently with the use of a historical weighting scheme.

4.10. SCENARIOS

To further interpret the results of our metrics we present three scenarios and an example of the application of our metrics.

Consider the following scenarios, in which a software developer or project manager needs to:

1. decide whether or not to depend on a certain third-party library to perform a certain function;
2. decide whether or not to create a wrapper around a third-party library to encapsulate dependencies on this library to reduce risks;
3. determine if a library is in a state of maintenance or active development.

We demonstrate how to apply our metrics and how these metrics can help in making a decision in these scenarios.

Assume that a developer wants to know if he should include the Commons Codec library in their software project or if he should write his own codec methods. There are several trade-offs to consider when making this choice. Commons Codec is a library which contains highly specialized functionality and it is likely to take a large effort to rebuild. It also requires a large amount of specialized knowledge which the developer may not have.

On the other hand, if the implementation or interface of the library is unstable, including this library may become a risk to the stability of the software itself. Other code may need to be rewritten to adapt to the new library interface. When developing the
functionality internally, the developer also has greater control over added features and maintenance of the code. Table 4.6 shows that Commons Codec scores 31 for metric WRM, which is the second place. This means that the API of Commons Codec tends to stay relatively stable regarding the removal of frequently used methods.

Another consideration to make is whether the latest version of a library already contains required functionality or if it is not included or completed yet. PNM shows the percentage of new methods in each snapshot. If required functionality is already included then it is not necessary to wait for the latest updates but when a new piece of functionality is required which is not implemented yet, such as a new video codec, it may be worth the wait. In the case of Commons Codec, the library has the highest score (0.5083), which indicates a large degree of new methods in each snapshot. A similar reasoning can be followed when looking at metric RCNO, which shows the ratio of work in new to old methods. Commons Codec has a score of 1.8, which shows that there is more work going on in new methods than in old ones. To obtain a more detailed picture of the evolution of a system, scores and metrics per snapshot can be obtained, which are shown in Table 4.7.

| s  | |U| | |Uo| |ΔU| |ΔUo| |Uf| |hw| metric values |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 215 | - | - | - | - | - | - |
| 3 | 318 | 107 | 1.32 | 211 | 0.7 | 4 | 1/4 | 0, 0.7, 1.88, 0.33 |
| 2 | 373 | 80 | 0 | 293 | 0 | 25 | 1/2 | 62, 0, 0, 0.21 |
| 1 | 503 | 130 | 0 | 373 | 0 | 0 | 1 | 0, 0, 0, 0.25 |

Table 4.7: Details of the metrics for Apache Commons Codec

Table 4.7 shows that there were 4 methods removed in the first snapshot, there were 25 methods removed in snapshot 2 and there were 0 methods removed in snapshot 1 (latest). The percentage of new methods is relatively high in snapshot 3: 33%. In the latest two snapshots, ΔUn is 0. The release notes of Commons Codec 1.5 (snapshot 2) state that new methods were added to the public interface, but apparently these are not yet used in our dataset. Similarly, ΔUo is 0 for the latest two snapshots, which means that there has not been change in methods that are being called in our dataset.

To get even more information, names and metric values for methods in Un, Uo and Uf for each pair of subsequent snapshots can be obtained. With this information, identification of most frequently used and most changing methods becomes possible. We do not show this list here but this information could be included when using our approach in practice. This way, identification of potential issues in removed or changed methods in third-party libraries becomes possible.

Our metrics can be used to get a clearer picture of the historic stability of a library. Although we do not expect that a small difference in these metrics has a large effect on a system which uses this library, an unstable library can pose a problem in the long run, when deferred updates are accumulated as “technical debt”. Our metrics also provide a way to test whether the reputation of a library in the open source community is really deserved, which is for a large part based on provided functionality but may also be influenced by the stability of a library in the past and the way the library deals with breaking changes. We try to catch these aspects in our metrics. Our metrics can provide more
practical help when a decision needs to be made to create a wrapper around a library which encapsulates changes in that library.

**4.11. DISCUSSION**

We have introduced several new concepts and metrics to measure the stability of the implementation and interface of a library through time. In this section, we discuss implications and limitations of our study and we discuss directions for future work.

**4.11.1. USED METRICS**

We have used the McCabe value as metric value in our formulas. Instead, different metrics and units can be used. For example, it could be useful to calculate the number of methods per class as metric and the usage frequency of each class as weight. This would provide information on a higher level of abstraction than McCabe or LOC values per method and would give an idea about the evolution of the size of each class through time.

The size of libraries differs significantly and we do not know how big the effect of system size is. In future work, we plan to investigate this effect of size and to adjust for this effect. We expect that a large part of stability is independent of system size and our metrics can therefore also be used without this normalization.

It can be difficult to interpret dimensionless and rangeless numbers directly, which can only be properly done after comparison or benchmarking. For this reason, these metrics should be calculated for a greater set of libraries. In future work, we plan to calculate our metrics on the complete Maven repository, containing over 300,000 artifacts with multiple versions. Such a large collection of metrics enables a more detailed analysis of distributions and percentiles.

**4.11.2. USING HISTORICAL DATA**

We use historical data from software repositories to calculate a metric which gives an impression of the “volatility” of the interface and implementation of a library. To achieve this, only historical data is used and a question is to what extent this can be used to predict future developments. With this approach, unlikely and unforeseen trend breaks will not be detected but it gives a good indication of the historical trend of a library. Assuming that the same development team keeps working on the code, developer behavior is expected to follow a similar trend as in the past.

Another issue with our approach is that code of historical snapshots is required to calculate a metric. This limits the applicability of our method to open-source libraries or libraries of which source code is otherwise available. By performing our method on the most frequently used third-party libraries we hope to provide practical knowledge and pointers to best practices of API stability which can be applied to other libraries as well.

Ideally, more fine-grained data should be used for analysis such as change sets per commit from a version control system. This information is not always available, however, and metrics depending on this information would limit their applicability to systems with access to their version control systems. Our metrics do not have this requirement.
4.11.3. Estimating Rework Effort

To properly estimate the amount of rework effort required after the upgrade of a library, an experiment needs to be performed in which the impact of changes in our dataset is investigated. Differences between metric values of units with connections to third-party units and units without connections could be compared. This way, an estimate in terms of time or money could be roughly calculated. This experiment would also provide a validation of our metrics since we defined the effect of library instability to be a large amount of required rework that has to be performed after upgrading that library. If libraries with high scores for our metrics also cause larger differences in metric values in units with links to third-party units than in units without these links, then our metrics have been validated against a real-world dataset, assuming that the chosen metric difference is a good indicator for the amount of expected rework.

4.11.4. Transitive Calls and Dependencies

We have ignored transitive third-party references and transitive third-party method calls. When method $m_1$ calls method $m_2$ which in turn calls a third-party method $m_3$, then in our current analysis, only $m_2$ would be impacted by a potential change in $m_3$. In future work, transitive method calls could be included using a similar weighting scheme as applied to snapshots: methods that are closer to a third-party method in the call chain are potentially more influenced by a change than methods further away.

4.12. Threats to Validity

4.12.1. Internal Validity

Our simplified Maven dependency resolution system always chooses the latest version of a library in case of a conflict. The real Maven dependency resolution system is somewhat more complex and contains advanced heuristics as well. It is therefore possible that we included the wrong library version in certain snapshots, but we expect that the occurrence of this issue is negligible in our dataset.

Our dependency framework also potentially misses changes in the number of parameters of methods since this can be confused with an added overloaded method in the next snapshot. We did not investigate the number of parameters as metric value in this chapter but future work using this metric value should investigate this issue further.

4.12.2. Construct Validity

We chose to use a geometric series as weight for previous snapshots since this series has the property that the ratio of subsequent terms is constant and that the sum is finite. An alternative weighting scheme would change the final outcome of the metric. We believe that our choice is justifiable, since snapshots further away in time are deemed less important than more recent snapshots. More research is needed to understand alternative metric schemes and their impact on metric outcomes.

The choice of metric is also of great importance in our metric. We chose to use the difference in McCabe value. This has the consequence that when the LOC of a unit changes without adding a decision point (e.g. if, while, case), no difference is detected. To investigate the differences, the same metrics should therefore also be cal-
culated with the LOC as metric. Our equations provide a general framework in which different metrics and units can be used. More research has to be performed before these alternatives can be used in practice.

4.12.3. **EXTERNAL VALIDITY**

Our dataset used to obtain frequencies of use is considered to be representative for a wide range of industrial areas, such as insurance, banking, logistics and government. However, analyzed systems were all written in Java and use Maven as build configuration tool. We estimate that reuse of existing open-source components is more prevalent in the Java community than in communities of other programming languages and therefore a possible bias may exist which overestimates the frequency of use for third-party library methods. We do not believe that restricting our dataset to systems which use Maven as build tool introduces a bias since it only helps to identify exact dependencies and version numbers. Other build tools, such as Apache Ant, do not explicitly state version numbers but still have the possibility to include third-party libraries.

4.13. **CONCLUSION**

We have presented four stability metrics which calculate the stability of the public interface and implementation of a library based on the weighted number of removed methods, the change in metric values in existing units, the ratio between change in new and old methods and the percentage of new methods per snapshot. We investigated results of our metrics and showed an example of their application to an open source library.

Our contributions are the following:

- A case study of the upgrade of third-party libraries in a commercial software system which shows several issues associated with the use of these libraries;
- A framework for fact extraction and analysis of library dependencies in Java projects built with Maven;
- A proposal for four library stability metrics, incorporating weighting schemes for recency and for API usage;
- Instantiation of the metric framework to the Apache Commons libraries

Furthermore, we have identified a number of scenarios explaining how these metrics can be used.
5

IMPACT OF INTERFACE INSTABILITY

Best practices in software development state that code that is likely to change should be encapsulated to localize possible modifications. In this chapter, we investigate the application and effects of this design principle. We investigate the relationship between the stability, encapsulation and popularity of libraries on a dataset of 148,253 Java libraries. We find that bigger systems with more rework in existing methods have less stable interfaces and that bigger systems tend to encapsulate dependencies better. Additionally, there are a number of factors that are associated with change in library interfaces, such as rework in existing methods, system size, encapsulation of dependencies and the number of dependencies. We find that current encapsulation practices are not targeted at libraries that change the most. We also investigate the strength of ripple effects caused by instability of dependencies and we find that libraries cause ripple effects in systems using them and that these effects can be mitigated by encapsulation.\(^\text{1}\)

5.1. INTRODUCTION

Encapsulation is an important design principle in modern software development. The famous “Gang of Four” describe design patterns [47] which have the primary goal of encapsulating change and hiding implementation details. Booch [20] states that encapsulation should be used to localize changes to specific places in a system. In the end, these principles should make it easier and cheaper to modify a software system and to implement new requirements.

Although there is general consensus among developers that encapsulation principles should be used, little is known about the actual usage and the effects of these principles in real-world software systems. In this chapter, we therefore investigate encapsulation principles and their relationship with various system properties on a set of 148,253 library versions with a total of more than 350 million lines of code. By measuring library stability, encapsulation and stability of dependencies, we can investigate whether de-

\(^{1}\)Parts of this chapter have been published as “Testing principles, current practices and effects of change localization”, MSR 2013 [92].
dependencies are being encapsulated, whether changes from these dependencies cause ripple effects in systems using them and whether encapsulation can decrease the impact of these ripple effects.

The goal of this chapter is to shed some light on current practices and effects of encapsulation: are software developers encapsulating the right software libraries, that is, the ones that change the most? We investigate the relationship between several properties of software systems, such as size, popularity and changes in these libraries and their dependencies. We also investigate factors that are correlated with breaking changes in library API's.

We measure encapsulation through a simple metric, which focuses on the desired effect of encapsulation: limiting the amount of code that is exposed to a library and thus exposed to potential changes in that library. We use the percentage of source files that import a certain library in a client to measure this. We measure stability of libraries and their API's through the change in existing methods, method removals and growth in new methods.

The structure of this chapter is as follows. In Section 5.2, the problem is stated. Section 5.3 explains how data was obtained and what techniques have been used to calculate our metrics. Section 5.4 describes the implementation of our approach. Section 5.5 contains descriptive statistics on our dataset. In Sections 5.6 and 5.7, results of our analysis can be found. In Section 5.9 and 5.8, threats to validity and a discussion can be found. Section 5.10 discusses related work and Section 5.11 concludes the chapter.

5.2. Problem Statement
5.2.1. Illustrative example
As an illustrative example of interface instability, we investigate the H2 relational database management system. This open source database system is written in Java and supports standard ISO-SQL and JDBC. H2 has a fairly stable release schedule, with a minor release approximately every 2-3 weeks. Table 5.1 shows the number of removed methods and classes from its API between a sample of (non-subsequent) releases. As can be seen in this table, the number of method and class removals from public interfaces is considerable. For instance, between version 1.2.133 (dated April 10, 2010) and version 1.3.158 (dated July 17, 2011) 68 method removals and 15 class removals occurred.

The actual usage of these methods and classes in other libraries or systems is not taken into account in this table; the impact of these interface changes on systems using H2 is therefore unknown. But this case nevertheless illustrates the amount of method and class removals from public interfaces that can occur during continuous development of a library.

When the removed methods and classes are being used by other systems and libraries, rework has to be performed because every method or class removal from a public interface causes a breaking change in libraries using that method or class. This results in compilation errors in systems and libraries which then have to be rebuilt and fixed before they can be executed again. The better dependencies to libraries are encapsulated at particular places, the less code has to be changed when such a breaking change occurs. We regard this to be the ultimate goal of encapsulation and the localization of changes:
Table 5.1: The number of breaking changes between different versions of the H2 database system. In the lower-left side of the table, below the diagonal, the number of method removals from public interfaces can be found. In the upper right side of the table class removals can be found.

to limit the amount of work required to make a change in a software system and to limit the amount of places where changes have to be made.

5.2.2. Research Questions

In this chapter, we aim to answer the following research questions:

- **RQ1**: How do library properties like size, stability, encapsulation and popularity relate to each other?
- **RQ2**: Which library properties influence the stability of a library?
- **RQ3**: How is encapsulation of library dependencies currently applied in practice?
- **RQ4**: Do unstable libraries cause ripple effects in systems that use them, and can these effects be mitigated by encapsulation?

In the next section, we begin by discussing concepts and the methodology we used. We then discuss our experimental setup to obtain metrics from source files. After this, we explore individual library properties to answer **RQ1** and **RQ2**. We then investigate relationships between libraries and the encapsulation of dependencies to answer research questions **RQ3** and **RQ4**. Throughout this chapter, we refer to the H2 database system as a running example to illustrate concepts and models.

5.3. Conceptual Framework and Methodology

In this chapter, we define *library stability* to be the amount of change in a library compared to its previous version. The less change, the more stable a library is. These changes may happen in such way that existing functionality is changed or existing interfaces are broken. This can have an effect on systems using these libraries and can possibly cause rework. We call the rework caused by library dependencies *ripple effects*. As stated before, modern software development principles state that functionality should be encapsulated, which has the goal of reducing the amount of effort required to implement a change, i.e. to reduce the size of the ripple effect.
We measure the amount of encapsulation of a certain dependency in a system through the isolation rating. We define the isolation rating for library L used in client C as the percentage of files in C that does not contain an import statement of library L. Higher isolation of library L in system C indicates better encapsulation of L and usage of L in fewer files of C, and thus possibly less rework caused by changes in L. The average outgoing isolation rating is the average rating of all libraries that C uses. It indicates the average encapsulation effort of developers responsible for implementing C. The average incoming isolation rating is the average isolation of library L in systems that use it, which indicates the amount of isolation deemed necessary for L across all users of L.

To measure library stability, we use three metrics which we defined in Chapter 4:

- **CEM (Change in Existing Methods)**
  CEM measures the amount of change in cyclomatic complexity (McCabe) between two versions of a library. It is calculated by summing over the differences between McCabe values for each method in both versions and weighting the result with the times each method is being used in a certain reference set. A high CEM value indicates that a library has a large amount of change (churn) in existing and frequently used methods compared to its previous version.

- **WRM (Weighted number of Removed Methods)**
  WRM is the number of removed methods weighted by the time each method is being used. A large WRM value indicates a library with a large amount of used methods removed from its interface as compared to its previous version.

- **PNM (Percentage of New Methods)**
  The PNM is the percentage of new methods that have been added to the next version of a library. A high PNM value indicates growth in the public interface of a library.

These metrics all measure library stability in a different way and provide a single number for metric differences in a library compared with its previous version. For the statistical analyses in this chapter, we select WRM as an dependent (outcome) variable to assess factors influencing library stability, since removing methods that are used by other systems always causes breaking changes in these systems and always requires rework. Although CEM also gives an indication of the amount of change that has occurred and this change is weighted by usage frequencies, it would also include non-relevant rework in methods since it is unknown which part of added lines of code or changes in the cyclomatic complexity will cause an observable difference in behavior from the perspective of a library user.

We answer **RQ1** and **RQ3** by computing correlations between properties of individual libraries. We measure ripple effects and the effects of encapsulation with linear regression techniques. These effects becomes visible statistically by constructing models which include stability of libraries, encapsulation of dependencies, and stability of these dependencies. We fit a linear regression model with WRM as dependent variable to assess the influence of other library properties on the stability of a library to answer **RQ2**. We finally consider statistical models to account for the interaction between encapsulation of dependencies and ripple effects caused by instability in these dependencies to answer **RQ4**.
A robust regression method (Huber and biweight iterations [73]) is applied to the linear regression models in this chapter, meaning that estimates of standard errors and p-values are robust against violations of normality, homoscedasticity and independence [44]. We have applied log transformations where needed, because the data is strongly skewed and visual inspection shows that the data is approximately normally distributed after applying a logarithmic transformation. To calculate correlations between library properties, Spearman rank correlation coefficients were used since we cannot assume a linear relationship between these properties. When fitting a linear regression model without taking the clustered structure of the network of library dependencies into account, incorrect conclusions could result due to model misspecification [2, 103]. To account for the graph structure of dependencies between libraries, more advanced statistical methods are required. Multilevel modeling [102] is used to fit a statistical model which takes into account these dependencies.

Regarding the relationship of popularity with other library properties (RQ1), we expect that popular systems have more stable interfaces for two reasons. First, developers of popular systems might feel limited in their freedom to change existing interfaces because a larger number of other systems depend on it. Second, we expect that libraries are more popular because they have more stable interfaces. This is beneficial for software developers because this reduces the expected amount of rework due to ripple effects when including these libraries. Either way, this becomes visible by calculating the correlation between library stability and network metrics such as the PageRank [86]. Systems that are more “central” in the network of library dependencies are expected to get a higher score for these metrics.

We do not make a distinction between test code and “production” code in our analysis since this distinction is not relevant for our research question. Ripple effects coming from a dependency can also occur in test code and the effects of method removals from interfaces are identical for test and non-test code.

5.4. IMPLEMENTATION

To calculate correlations between system properties, metrics of all libraries in the Maven repository were obtained. To obtain source code measurements such as the number of lines of code, number of methods and stability metrics, the Software Analysis Toolkit of the Software Improvement Group was used, which was adapted to run in parallel on multiple machines.

Specialized data structures were required to store the large amount of data. As described in Chapter 2, Berkeley DB\(^2\), an on-disk key-value store, was used to store all properties of methods on disk and to make statistical calculations possible. To calculate isolation ratings, source code jars were unzipped and source code was scanned for package declarations. This way, a collection of package names was obtained for each jar file. These names were reduced to one or multiple package prefixes, which were used to scan for dependencies in other libraries. An example of a package prefix of the H2 library is com.h2database. For each jar file, the number of files that contain an import

\(^2\)http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html
statement which starts with a package prefix were counted. Only prefixes of dependencies which were included in the corresponding pom.xml file of the library were checked. For instance, to calculate the isolation rating for H2 in systems using it, the number of files that contain an import statement starting with “import com.h2database” were counted. When multiple statements with the same prefix appear within a file, the file was counted only once. The final score is 1 minus the proportion of files importing a certain library, so a higher score means better encapsulation of a specific dependency in a library.

As noted before, the example of the H2 database system does not take into account the actual usage of removed methods and the impact on systems using these libraries is therefore unknown. Our metrics take the actual usage frequencies of methods into account by weighting each metric value of a method by the number of times this method is being used throughout the Maven repository. To obtain usage counts, java class files were disassembled from binary jars using javap -private -c -s, meaning that a bytecode dump of all class methods was created with method calls annotated with fully qualified names. These names were counted and also stored. In contrast to previous work [90], we add 1 to all usage frequencies, even if methods are never called. This ensures that all libraries get a non-zero metric score and prevents that most libraries receive a score of 0 for all metrics. Methods that are called more frequently are still weighted more and have more influence in the final stability score of a library.

5.5. Descriptive Statistics

In Table 5.2, descriptive statistics of the data used in this chapter can be found. The database comprises 148,253 java libraries, but source code of only 94,670 libraries is available due to a variety of reasons, such as corrupted jar files or jar files containing only non-source code files. The code that is available has a total of 350,571,247 SLOC, 4,174,150 classes and 37,406,546 methods.

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>p5</th>
<th>p25</th>
<th>p50</th>
<th>p75</th>
<th>p95</th>
<th>max</th>
<th>avg</th>
<th>sd</th>
</tr>
</thead>
<tbody>
<tr>
<td>nM</td>
<td>1.0</td>
<td>4.0</td>
<td>21.0</td>
<td>69.0</td>
<td>240</td>
<td>1.5k</td>
<td>56k</td>
<td>468</td>
<td>1.7k</td>
</tr>
<tr>
<td>nC</td>
<td>1.0</td>
<td>1.0</td>
<td>3.0</td>
<td>10.0</td>
<td>30.0</td>
<td>223</td>
<td>4.7k</td>
<td>52.23</td>
<td>166.7</td>
</tr>
<tr>
<td>LOC</td>
<td>1.0</td>
<td>39.0</td>
<td>203</td>
<td>650</td>
<td>2.2k</td>
<td>17.5k</td>
<td>382k</td>
<td>4.4k</td>
<td>15.7k</td>
</tr>
<tr>
<td>inD</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>3.0</td>
<td>10.0</td>
<td>76.0</td>
<td>19.6k</td>
<td>24.42</td>
<td>212</td>
</tr>
<tr>
<td>outD</td>
<td>1.0</td>
<td>1.0</td>
<td>2.0</td>
<td>5.0</td>
<td>8.0</td>
<td>18.0</td>
<td>211</td>
<td>6.5</td>
<td>7.02</td>
</tr>
<tr>
<td>inI</td>
<td>0.01</td>
<td>0.01</td>
<td>0.38</td>
<td>0.70</td>
<td>0.91</td>
<td>1.0</td>
<td>1.0</td>
<td>0.63</td>
<td>0.31</td>
</tr>
<tr>
<td>outI</td>
<td>0.01</td>
<td>0.01</td>
<td>0.51</td>
<td>0.75</td>
<td>0.98</td>
<td>1.0</td>
<td>1.0</td>
<td>0.67</td>
<td>0.26</td>
</tr>
<tr>
<td>WRM</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>98.0</td>
<td>1.5m</td>
<td>298</td>
<td>13k</td>
</tr>
<tr>
<td>CEM</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>5e-06</td>
<td>0.02</td>
<td>31.5</td>
<td>0.01</td>
<td>0.14</td>
</tr>
<tr>
<td>PNM</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.01</td>
<td>0.35</td>
<td>1.0</td>
<td>0.06</td>
<td>0.17</td>
</tr>
<tr>
<td>RCNO</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.6k</td>
<td>7.3m</td>
<td>3.4k</td>
<td>69k</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Descriptive statistics for libraries in the Maven repository. nM = number of methods, nC = number of classes, LOC = lines of code, inD = in-degree, outD = out-degree, inI = average incoming isolation rating, outI = average outgoing isolation rating.

The variation in system size (LOC) is large: the smallest system is only 1 LOC, compared to the largest system which is 382,000 SLOC. There are few systems larger than
2,000 lines of code: the 75th percentile is at 2,200 LOC. Inspection of a sample of libraries showed that there are libraries which only contain an empty interface, which explains the minimum of 1 line of code. Moreover, there exist “property jars” in the Maven repository, which only contain configuration files and no Java code. Libraries are used 24.42 times (in-degree) on average and use 6.5 other libraries (out-degree) on average. The maximum number of times a library is being used is 19,621 (this is JUnit 4.8.2). Note that values for nM, LOC, WRM and RCNO have large standard deviations, which indicates a great spread in data values. Most metrics follow a strong power law, in which most values fall within a certain range (for instance, close to 0.0 for WRM) and there exist a small number of extreme outliers.

The next section describes results obtained from further analysis. We first investigate relationships between properties of individual libraries. After this, we take into account dependencies between libraries.

5.6. INDIVIDUAL LIBRARY RESULTS

5.6.1. THE RELATIONSHIP BETWEEN LIBRARY PROPERTIES

To answer RQ1, we inspect relationships between properties of individual libraries. These relationships are shown in Table 5.4. In the lower left part of the table, below the diagonal, the strength of correlations can be found [28]. In the upper right part of the table p-values can be found. Using a Bonferroni adjustment of $105 (\sum_{i=1}^{14} i)$, all shown correlations are significant at the 0.0005 (0.05 / 105) level. P-values smaller than 1e-300 are denoted as 0 and nonsignificant correlations are not shown.

Four of the most interesting relationships are also presented graphically in Figure 5.3. The upper left panel exhibits the relationship between the log transformed WRM and CEM. Since the log is a monotone function, transformation of the two variables does not change their rank correlation. The rank correlation coefficient is 0.53, indicating that systems with more change in existing methods tend to have less stable interfaces. The upper right panel shows that bigger systems (measured in number of methods, $nM$) also tend to have less stable interfaces, with a correlation coefficient of 0.36. The number of data points ($n$) is different in the four graphs due to missing package prefixes.

The lower panels of Figure 5.3 shows the relationships between system size ($nM$) and the average incoming (inI) and outgoing isolation rating (outI). In the lower left panel, the average outgoing isolation rating is plotted against system size. The positive correlation coefficient of 0.47 indicates that dependencies on other libraries tend to be better encapsulated in bigger systems. The opposite does not seem to be the case: there exists only a very weak correlation of -0.11 between system size and the average incoming isolation rating, as can be seen in the lower right panel. This indicates that bigger libraries tend to be encapsulated only marginally less in systems that use them.

Table 5.4 further shows that network metrics, such as the PageRank and the Hub-biness and Authoritativeness from the HITS-algorithm [65] are weakly positively correlated with other system properties. This contradicts our hypothesis that more popular libraries are more stable and shows that popular libraries tend to change more. The table further shows that the four stability metrics WRM, CEM, PNM and RCNO are strongly correlated, indicating that rework in existing methods, building new methods and re-
moving old methods are activities that are often performed together.

The table also shows that there exists an almost perfect correlation between WRM and nMr, which is not surprising since WRM and nMr both measure the number of removed methods but WRM weighs them with the number of times they are being used. Similarly, nMn and PNM are also strongly correlated because they measure the same thing in a different way; PNM calculates the percentage of new methods in a snapshot while $nMn$ is the absolute number of new methods in each snapshot.

![Figure 5.3: Scatterplots and Spearman rank correlations between, CEM, WRM, number of methods (nM) and average incoming (inI) and outgoing isolation rating (outI). The axis limits have been adjusted and logarithmic transformations have been applied to demonstrate the relationships more clearly.](image)

### 5.6.2. Regression Results

Although the correlations in the previous paragraph provide a first insight in the relationship between library properties, they do not provide us any information on possible influences of library properties on breaking changes in library interfaces. To investigate this, we perform a linear regression analysis with WRM as the dependent (outcome) variable and multiple independent (predictor) variables. The results of this analysis can be found in Table 5.5. With this analysis, we can investigate possible explanations for library instability from the perspective of a single library.

We performed a linear regression analysis with $\log(WRM)$ as dependent variable and $\log(CEM)$, $\log(nM)$, $\log(PNM)$, average outgoing isolation rating (outI) and the outdegree as independent variables. The model is based on 7,394 observations and the results of
Table 5.4: Spearman rank correlation matrix for jar file properties. All shown correlations are significant at the 0.0005 (0.05 / 105) level. PageR. = PageRank, Hubb. = Hubbiness, Auth. = Authoritativeness.

<table>
<thead>
<tr>
<th></th>
<th>PageR</th>
<th>Hubb.</th>
<th>Auth.</th>
<th>WRM</th>
<th>CEM</th>
<th>RCNO</th>
<th>PNM</th>
<th>nM</th>
<th>nMn</th>
<th>nMr</th>
<th>InI</th>
<th>OutI</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hubb.</td>
<td>-0.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Auth.</td>
<td>0.96</td>
<td>-0.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRM</td>
<td>0.11</td>
<td>0.07</td>
<td>0.11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEM</td>
<td>0.13</td>
<td>0.07</td>
<td>0.13</td>
<td>0.53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCNO</td>
<td>0.14</td>
<td>0.08</td>
<td>0.14</td>
<td>0.58</td>
<td>0.83</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PNM</td>
<td>0.12</td>
<td>0.07</td>
<td>0.12</td>
<td>0.75</td>
<td>0.59</td>
<td>0.70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nM</td>
<td>0.19</td>
<td>0.17</td>
<td>0.17</td>
<td>0.36</td>
<td>0.39</td>
<td>0.43</td>
<td>0.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nMn</td>
<td>0.14</td>
<td>0.08</td>
<td>0.14</td>
<td>0.78</td>
<td>0.62</td>
<td>0.75</td>
<td>0.98</td>
<td>0.44</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nMr</td>
<td>0.11</td>
<td>0.07</td>
<td>0.11</td>
<td>0.52</td>
<td>0.58</td>
<td>0.76</td>
<td>0.36</td>
<td>0.78</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InI</td>
<td></td>
<td>0.24</td>
<td></td>
<td>0.11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutI</td>
<td>0.10</td>
<td>0.31</td>
<td>0.10</td>
<td>0.20</td>
<td>0.16</td>
<td>0.20</td>
<td>0.19</td>
<td>0.47</td>
<td>0.24</td>
<td>0.20</td>
<td>0.08</td>
<td></td>
</tr>
<tr>
<td>InD</td>
<td>0.45</td>
<td>0.06</td>
<td>0.33</td>
<td>0.12</td>
<td>0.13</td>
<td>0.15</td>
<td>0.12</td>
<td>0.23</td>
<td>0.16</td>
<td>0.11</td>
<td>0.17</td>
<td>0.06</td>
</tr>
<tr>
<td>OutD</td>
<td>-0.04</td>
<td>0.54</td>
<td>-0.02</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
<td>0.24</td>
<td>0.15</td>
<td>0.14</td>
<td>-0.09</td>
<td>0.39</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5: A regression model performed on the number of removed methods from library interfaces.

Expressed as a formula, the model looks as follows:

\[
\log(\text{WRM}) = 0.352 + 0.094 \log(\text{CEM}) + 0.939 \log(\text{nM}) + 0.694 \log(\text{PNM}) - 0.439 \text{outI} + 0.007 \text{outD}
\]

The effect of all predictors in our model is significant with p-values close to 0. The standardized coefficients ("Beta") in Table 5.5 indicate that the size of the system and the percentage of new methods are the two most important factors. The $R^2$ of the model is 0.3877, indicating that 38.7% of the total variability in $\log(\text{WRM})$ can be explained by the model. Furthermore, the p-value of the overall model is 0.000, indicating a rejection of the null hypothesis that all the slopes in the linear model are zero.

The model shows that there exists a positive linear relationship between CEM and WRM, indicating that systems which are more actively developed tend to have less stable interfaces. There also exists a positive linear relationship with PNM, which indicates that growing systems tend to have less stable interfaces. The average outgoing isolation rating influences WRM negatively, indicating that systems which encapsulate their dependencies better tend to have more stable interfaces. The number of dependencies
to other libraries is positively correlated with WRM, indicating that libraries with more external dependencies tend to have less stable interfaces.

Although correlations between these library properties and WRM of the opposite direction can be found in Table 5.4, this model shows us that after correcting for other library properties, this direction reverses. The results of our linear regression model thus indicate that the amount of breaking changes in library interfaces can be explained by the churn in existing methods, the size of the system measured in the number of methods, the percentage of new methods in each snapshot, the average outgoing isolation rating and the number of other libraries the library depends upon. The PageRank, Hubbiness and Authoritativeness were originally also included in this model but did not have a significant effect on instability in libraries and where removed from the model.

To answer RQ2: The encapsulation of dependencies has a significant positive effect, and the size of the library, the growth in new methods, the change in existing methods and the number of external dependencies have a significant negative effect on the stability of a library.

In the case of the H2 database system, we can predict the number of breaking changes in the public interface of a specific library version by filling in values for all independent variables. For instance, version 1.3.157 of the H2 database system has a CEM of $2.1 \times 10^{-4}$, 7058 units, a PNM of $8.8 \times 10^{-3}$, an average outgoing isolation rating of 0 and an outdegree of 0. Filling in these values in the regression formula leads to a predicted WRM of $e^{4.58} = 98.45$. The actual WRM for this library version is 64. The model also tells us that if the number of methods and the PNM value would be halved, predicted WRM would be $e^{3.60} = 32.04$. This illustrates that smaller, slower growing systems tend to have greater stability of public interfaces. Predicted values could be further reduced by increasing encapsulation of external dependencies in the system and decreasing the number of external dependencies. However, the model does not imply a causal relationship between predictors and the outcome variable and therefore care has to be taken when using this model for prediction, especially considering the $R^2$ of the model (0.3877), indicating that only 38.77% of variability in the outcome variable can be explained by the model.

5.7. Library Interdependency Modeling Results

In the previous section we investigated relationships between properties of individual libraries. In this section we take into account dependencies to other libraries. We start by creating a simple model which gives us an indication of the size of the ripple and encapsulation effect. After this, we take more of the complex structure of the data into account to see whether this advanced model confirms our simpler analysis.

5.7.1. Estimating the Encapsulation Effect

In order to estimate the effect of encapsulation on stability of libraries, we fit another linear regression model which investigates the effect of encapsulation on the stability of libraries and their dependencies. We want to model the effect of instability in dependencies (WRM_to) on stability in libraries using them (WRM_from), while correcting for encapsulation of these dependencies. We expect to find a positive relationship between instability in dependencies and library instability, indicating that library instability tends
to increase when instability in dependencies increases. We expect that the addition of isolation as a predictor will have a dampening effect, indicating that encapsulation is able to offset ripple effects.

The results are displayed in Table 5.6. Expressed as a formula, the model looks as follows:

\[
\log(WRM_{from}) = 3.24 + 0.037 \log(WRM_{to}) - 1.17 \text{is} 
\]  

(5.2)

The model is based on 6,813 cases, is significant with a p-value of 0 but has an R² of only 2%. This does not pose a problem since the effect found is highly significant and the isolation rating and the WRM in dependencies alone are not expected to explain a large part of the variability in the outcome variable.

The model shows that there indeed exists a positive effect of dependency instability on library instability while correcting for isolation (\(\text{is}\)). The intercept of 3.24 can considered to be the baseline change in libraries, regardless of dependency changes. The model is in line with our expectation to find a ripple effect: for every increase in \(\log(WRM_{to})\), there is a 0.037 increase in \(\log(WRM_{from})\). This is the residual ripple effect that remains when taking into account baseline changes in libraries and encapsulation of dependencies. It shows that, apparently, encapsulation is not fully capable of preventing ripple effects coming from dependencies. This model partly answers RQ4, but further analysis is performed in Section 5.7.3.

5.7.2. CURRENT ENCAPSULATION PRACTICE

To investigate whether dependencies that change more are isolated better, i.e., whether there exists a positive correlation between \(\log(WRM_{to})\) and isolation, we performed a Spearman rank correlation test. This gives a Spearman's \(\rho\) of 0.0295 (\(p=0\)), meaning that there does not exist a strong correlation between breaking changes from dependencies and isolation of those dependencies. We expected to find a positive correlation. The result indicates that existing encapsulation practices are not targeted at dependencies that change the most. This possibly means that developers are not aware which libraries change the most and thus do not isolate these libraries better.

To answer RQ3: current encapsulation practice is not targeted at the most unstable libraries.

5.7.3. MULTILEVEL MODEL FOR INTERFACE INSTABILITY

The robust regression technique as used in the previous paragraphs can provide us with useful initial estimates and is robust against violations of independence and nonnormal-
ity to some extent. To fully take into account the complete structure of the data, however, a more advanced statistical method has to be used. Instead of treating the complex network structure as a nuisance and control for it, this structure can also be incorporated in the model specification which enables us to perform a more sophisticated analysis of sources of breaking interface changes. A technique called hierarchical or multilevel modeling is capable of dealing with the type of relationships present in our data, such as a one-to-many relationship between a library and its dependencies.

Specifying a model that fully acknowledges all dependencies between observations enables us to get an unbiased and correctly estimated effect of the size of ripple effects. Multilevel modeling is commonly used in the social sciences, for instance to model the quality of care of nurses in a hospital. Each patient can be treated by one or more nurses and a single nurse can treat multiple patients. If we want to model the effect of work hours of a nurse on the health of a patient, we could run correlation test which correlates the blood pressure of patients, for instance, and the amount of hours each nurse works during multiple weeks. The problem is that multiple measurements of work hours over time belong to the same nurse and multiple measurements of blood pressure over time belong to the same patient. Also, patients can move in and out of the ward, thus not receiving care of the same nurse anymore. When not accounting for dependencies such as these, incorrect conclusions would be drawn from the analysis [2, 103].

The same principles apply in our dataset, of which Figure 5.7 shows an example. As can be seen in panel A of this figure, there can be multiple library versions pointing to multiple versions of other libraries. The tabular form of properties presented in Figure 5.7A can be found in Table 5.8. Since statistical analysis requires data to be stored as one observation per row, data duplication results, which leads to violations of independence of observations. To see why, see Table 5.8, where multiple violations of independence of observations can be identified. The first type, duplicated measurements, appear because there exist one-to-many relationships between a library and its dependencies. In Table 5.8, Libraries A1, C1 and D1 have multiple dependencies and thus reappear with corresponding \( WRM_{from} \) values. A time dependency is also present between A1 and A2. Incorporating time dependencies in our model acknowledges the fact that measurements of the same library over time are more likely to be correlated than measurements of independent libraries. Summarizing both duplicated measurements and time dependencies, measurements eventually belong to a group of the same artifact, as can be seen in the latest two columns. Finally, the isolation ratings marked as independent observation in the table are independent since no related libraries are involved. The duplication is a coincidence in this case.

### 5.7.4. Model specification

Part B of Figure 5.7 shows a piece of the corresponding multilevel model. We use a model with three levels: individual measurements nested in dependency versions nested in dependencies. The groupings at higher levels are expected to influence lower levels: measurements that are grouped at a higher level are expected to be correlated more than measurements which are not. \( \log(WRM_{from}) \) is defined for the individual library at level 1. These libraries point to other libraries, which is level 2 in our model. The libraries at level 2 are multiple versions of the same library, which is level 3 in our model. The
model acknowledges that the same dependency version has an effect that is expected to be correlated between different libraries using it, since the same dependency is causing the effect. The multilevel modeling technique then takes care to incorporate the clustered structure of the data as specified in our model. This way, estimates of the influence of different dependencies and dependency versions on instability and ripple effects are obtained. We also apply the same robust regression techniques as in the previous analyses.

Similar to the model in Section 5.7.1, we want to model the effect of WRM in library dependencies ($WRM_{to}$) on WRM of libraries ($WRM_{from}$), while taking into account isolation of these dependencies. This enables us to investigate RQ4 further while taking the clustered structure of the data into account. We choose to define a model that does justice to the reality of our dataset while reducing complexity of the model to a minimum at the same time; we therefore ignore the fact that libraries at level 1 are multiple versions of the same library.
We expect that not all dependencies are isolated similarly, but that certain libraries are more difficult to encapsulate than others. To incorporate this hypothesis in our model, we let the isolation rating vary among dependencies. This means that we expect a difference between the isolation of a logging framework and a database application, for instance. We also expect that library stability varies among artifacts, meaning that we expect differences in baseline instability between different artifacts. This is shown in Figure 5.9. Panel A of this figure shows the results of a group-specific intercept, meaning that the baseline instability in libraries is expected to be different between groups. Panel B shows a group-specific intercept, meaning that the dampening effect of isolation on ripple effects is expected to be different between groups.

Formally, we can write the general regression formula with variable intercept and variable slope in the following form:

\[
\log(WRM_{ijk}) = \beta_0 + \beta_1 \log(WRM_{jk}) + \beta_2 is + \epsilon_{ijk} \tag{5.3}
\]

\[
\beta_{0jk} = \gamma_{00} + \delta_{0j} + \delta_{0k}
\]

\[
\beta_1 = \gamma_{01}
\]

\[
\beta_{2k} = \gamma_{02} + \delta_{2k}
\]

In this formula, the WRM for library version \(i\) (level 1) pointing to dependency version \(j\) (level 2) of dependency \(k\) (level 3) is modeled as a regression line with an intercept and slope depending on the WRM of dependencies and the isolation of those dependencies. The grand mean of WRM across all libraries is denoted as \(\gamma_{00}\), each dependency version adds a version-specific intercept \(\delta_{0j}\) and each dependency adds a dependency-specific intercept \(\delta_{0k}\). Comparing this model to formula 2, \(\beta_0\) is 3.24, \(\beta_1\) is 0.037 and \(\beta_2\) is -1.17. The individual error term is not shown in formula 2.

### 5.7.5. Multilevel model explanation

The results of this model are shown in Table 5.10. The overall model is significant with a p-value of 0. The model yields a grand mean \(\gamma_{00}\) of 3.27, which is the average \(\log(WRM)\)
5.7. Library Interdependency Modeling Results

<table>
<thead>
<tr>
<th>Dependent variable</th>
<th>log(WRM\textsubscript{from})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1 Group variable</td>
<td>Dependency versions</td>
</tr>
<tr>
<td>Level 2 Group variable</td>
<td>Dependencies</td>
</tr>
<tr>
<td>Number of cases</td>
<td>6,813</td>
</tr>
<tr>
<td>Number of level 1 groups</td>
<td>1,277</td>
</tr>
<tr>
<td>Number of level 2 groups</td>
<td>641</td>
</tr>
<tr>
<td>Cases per level 1 group</td>
<td>min 1, avg 5.3, max 523</td>
</tr>
<tr>
<td>Cases per level 2 group</td>
<td>min 1, avg 10.6, max 649</td>
</tr>
<tr>
<td>Wald $\chi^2$</td>
<td>65.31</td>
</tr>
<tr>
<td>P($&gt;\chi^2$)</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Single library effects parameter estimates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Independents</strong></td>
</tr>
<tr>
<td>$\gamma_{00}$</td>
</tr>
<tr>
<td>$\gamma_{01}$</td>
</tr>
<tr>
<td>$\gamma_{02}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group effects parameter estimates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Estimate</strong></td>
</tr>
<tr>
<td>$\sigma(\delta_{0j})$</td>
</tr>
<tr>
<td>$\sigma(\delta_{0k})$</td>
</tr>
<tr>
<td>$\sigma(\delta_{2k})$</td>
</tr>
<tr>
<td>$\sigma($Residual$)$</td>
</tr>
</tbody>
</table>

Table 5.10: A multilevel model for removals of interface methods.

in libraries. This is very close to our earlier result of 3.24 in Equation 5.2. The ripple effect $\gamma_{01}$ attributable to dependencies is exactly the same, namely 0.037. This is expected since there are no group-specific parameters for $\beta_1$; the estimation method for single libraries (the middle table in Figure 5.10) is a linear regression method. The encapsulation effect is also similar, -1.079 compared to -1.17 in formula 2. The estimates for single libraries are thus comparable to our earlier model, indicating that the results of that model were not an artifact of model misspecification.

The bottom table shows variances for estimates of group effects. All group effects as specified in the model are significant, as indicated by the fact that the confidence intervals do not include 0. The variance in instability due to specific versions of dependencies is denoted as $\sigma(\delta_{0j})$ and is estimated to be 0.317. This means that each library version adds its own amount of instability to the total amount of library instability. The variance in the additive baseline effect for dependency groups at level 3 is denoted as $\sigma(\delta_{0k})$ and is 0.636. Even more interesting is the library-specific dampening effect of isolation on ripple effects, denoted as $\sigma(\delta_{2k})$, which is 1.203. This indicates that there exists a large variance in the effect that the isolation of dependencies has on ripple effects. This means that it depends on the specific library dependency to what degree ripple effects from this library can be dampened with isolation.

To answer **RQ4**: library dependencies cause ripple effects in libraries using them and these effects can be mitigated by encapsulation. The size of this mitigating effect is library-specific.
5.8. DISCUSSION

5.8.1. INFERRING ACTIONABLE ADVICE FROM STATISTICAL ANALYSIS

The strength of our analysis lies in the fact that a large dataset has been analyzed, the robustness of the used regression techniques and the acknowledgement of the complex structure of the data at hand. Since the statistical analysis performed in this chapter does not immediately provide actionable advice, it is worth investigating the practical consequences of our results. Looking at the analysis performed in Section 5.6.2, we find that breaking interfaces are correlated with system size, rework in existing methods, the growth of the system, the average outgoing isolation rating and the number of outgoing dependencies.

It is not possible to infer causal relationships from this model but it implies that a system that grows faster and has more rework tends to break existing interfaces sooner. Software developers working on big or fast-growing systems should therefore pay attention to existing interfaces to maintain backward compatibility, which might be neglected during fast system growth. Systems that have more external dependencies tend to have more breaking interface changes, but from the simple linear regression it is unclear if this comes from changes in external dependencies. Since system size is also in this model and thus corrects for the fact that bigger systems tend to have more external dependencies and more breaking interface changes in general, the significant effect of the number of dependencies on the number of breaking changes in this model cannot be accounted to system size.

The formula in Section 5.7.1 indicates that encapsulation has a dampening effect on the stability of a library, although the statistical model does not require that changes as measured in dependencies are the actual cause of changes in libraries using them. However, the model distinguishes between a baseline effect (the constant of 3.24) and the additive effect of instability in dependencies with a value of 0.037, which indicates an additive effect per dependency, regardless of the baseline effect in the system without dependencies.

The multilevel model better accounts for the complex structure of the data and confirms the result of the straightforward analysis of Section 5.7.1. This means that the results found in Section 5.7.1 are not an artifact of model misspecification. Additionally, there seems to be a large library-specific encapsulation effect, which indicates that it depends on the specific dependency (for instance, Log4j or the Spring framework) what the dampening effect of encapsulation on ripple effects will be. Ripple effects from a certain dependency may be dampened more by encapsulation than ripple effects from another dependency.

5.8.2. RELATIONSHIP WITH ASPECT-ORIENTED PROGRAMMING

The isolation rating as defined in this chapter is closely related to the concept of scattering from the field of aspect-oriented programming. Scattering is the degree to which a certain feature is distributed among multiple program modules. Our isolation rating does not measure the scattering of single features but only looks at the distribution of library dependencies in source files. A library can be seen as a collection of related fea-

5.8. **Discussion**

atures, but if it is seen as a single feature then the isolation rating is 1 minus the scattering of a library in a system.

Another connection to aspect-oriented programming can be found in the concept of a cross-cutting concern. A cross-cutting concern is a feature that is difficult to encapsulate at a certain place because it needs to be used throughout the entire system. A classical example of this is logging. Our isolation rating does not distinguish between libraries that can be seen as cross-cutting concerns and libraries that are not properly encapsulated due to programming style. Libraries which implement cross-cutting concerns are inherently difficult to encapsulate, but libraries which do not have this property can be improperly encapsulated because developers just did not spend enough effort to encapsulate a dependency, while ideally, it should have been. Future work can make a distinction between these two types of situations.

In this chapter, we ignored the underlying mechanism used to achieve encapsulation since we assume that all encapsulation mechanisms will eventually lead to a smaller percentage of system code being exposed to a dependency. We assume that the more files import a library, the more exposure the system has to this library, and the higher the chance that a modification in that library will cause a ripple effect in the system using it.

5.8.3. **Model misspecification risk**

In applying complex statistical techniques such as the multilevel modeling technique in this chapter, the chance that models are misspecified increases as the complexity of the model increases. On the other hand, ignoring complexities present in the data structure will lead to incorrect inferences from models that are too simplistic. We tried to find a balance between a model that gives enough information to test our hypothesis while at the same time not being overly complex. The models considered in this chapter yield consistent results, thus strengthening our confidence that investigated relationships are present in the data, although the exact coefficients may need to be regarded as approximations given the fact that some model misspecification risk still remains present.

To our knowledge, the multilevel method has not been applied before in the software engineering research community. However, the multilevel modeling technique is capable of dealing with the complexity of our data structure and we therefore believe that this technique is the best way to make statistical inferences from our dataset. We also think that this technique should be applied more often in software engineering research, since situations with data dependencies as described in Section 5.7.3 are expected to be common.

5.8.4. **Testing multiple hypotheses**

We performed multiple correlation tests on the same dataset and a large part of them turned out to be significant. For this reason, concerns may rise about the increased chance of type I errors. Due to the large size of our dataset, tested hypotheses often have extremely small p-values, making almost all correlations statistically significant. This makes p-values of correlations less relevant and we have therefore focused more on the strength of correlations.

We applied a Bonferroni adjustment factor on the correlations in table 5.4, but there exist conceptual objections against the use of Bonferroni adjustments in general. First
of all, the truth of a hypothesis does not depend on the number of tested hypotheses. Additionally, Bonferroni adjustments increase the chance of type II errors (not finding correlations that do exist in reality). Statistically speaking, they reduce the power of the test. For a more elaborate discussion on the objections of using Bonferroni adjustments in general, see [88] and [83]. Nonetheless, even with a conservative Bonferroni adjustment factor of 105 as applied in this chapter, the largest part of tested correlations is significant at the 0.0005 level.

5.9. Threats to Validity

5.9.1. Internal Validity

We assume that the number of files that import a dependency is a good indicator for encapsulation of dependencies and modularity of design, but we do not try to automatically detect the type of mechanism to achieve this encapsulation. We assume that all encapsulation mechanisms and architectural patterns will eventually lead to a lower percentage of files importing a library. We assume that the number of unused imports per file is not large enough to influence our results.

Due to the large size of the dataset it is impossible to manually acquire package prefixes with which library imports can be recognized. Some libraries use multiple package prefixes, making automatical detection more difficult. Some files in our dataset do not have a package prefix and are therefore not included in the calculation of our isolation metric. We expect that there does not exist a bias in systems that have missing package prefixes.

We automatically assign snapshot numbers to subsequent versions, but manual inspection shows that this sometimes gives erroneous results. This will lead to incorrect stability ratings between two versions of a library. However, due to the large scale of the experiment, data errors like these will be only present in a small percentage of the total dataset and will not be strong enough to influence large-scale correlations. This is confirmed by manual inspection of a sample of jar files.

5.9.2. External Validity

The external validity of our results is large due to the size of our dataset. Although only open source third-party Java libraries are included, we do not have reason to believe that the results will be different for libraries written in other programming languages since our conceptual framework is language-agnostic and applies to any programming language in which external dependencies are defined in source code.

5.10. Related Work

Similar to the kind of relationships investigated in this chapter, Mohagheghi [81] et al. performed an experiment which investigates the relationship between defect density, stability and the impact of component size on defects. They found that components that are reused more change less. This was our initial hypothesis on the relationship between popularity and stability but this relationship is not present in our dataset.

Ripple effects have been investigated by others, for example by Herzig [57], who investigates the long-term impact of code changes by detecting dependencies between
code changes and by measuring their influence on software quality, maintainability and development effort. Black [18] takes a more formal approach and measures the ripple effect through the use of matrix algebra, which enables exact calculation of places in code that likely need to change, instead of large-scale statistical approximations as used in this chapter.

Cossette et al. [29] manually checked a set of API incompatibilities in newer versions of Java library versions and determined what the correct adaptations are to migrate from the older to the newer version of a library. A more general estimate of the amount of rework caused by these migrations can be found in the regression model in Section 5.7.1.

In previous work [90] we defined each metric to aggregate over all snapshot differences, while weighting the most recent differences more than older differences. In this paper, we only look at the four stability ratings as compared to the immediately preceding version of the library.

5.11. Conclusion

In this chapter, we made the following contributions:

- An investigation of the relationship between interface stability, encapsulation and stability in dependencies of almost 100,000 libraries;
- A statistical model to explain change in library interfaces;
- An investigation of the effect of encapsulation on ripple effects caused by unstable libraries.

Our analysis gives insight in the relationship between system properties such as size, stability and encapsulation. In particular, we come to the conclusion that library stability is influenced by the change in existing methods, the growth in the system, system size, encapsulation of dependencies and the number of dependencies. We also observed that current encapsulation practice does not seem to be targeted at libraries that change the most. Our analysis further shows that library dependencies cause ripple effects in systems that use them and that these effects can be mitigated by encapsulation.
Systems that include third-party libraries may want to update their dependencies when new functionality is developed, security patches are released, bugs are fixed or the interface of the library is improved. However, interface changes in these library updates may cause rework. In this chapter, we introduce a new change impact analysis technique, that uses compilation errors to measure the impact of a breaking change in a library. A breaking change is a change that is guaranteed to cause rework when the changed piece of functionality is used. Applying our technique to the Maven repository, a set of more than 100,000 open-source Java libraries, shows that breaking changes are widespread and that they have an impact on client systems, which are forced to adapt their implementations when using the changed functionality. Library properties that are associated with breaking update behavior are investigated, and the amount of work performed in libraries that introduce breaking changes is calculated. We also investigate factors that are associated with the dispersion of errors across different files in client systems.

6.1. Introduction

Developers that use third-party libraries regularly need to include the latest version of these libraries in their system to benefit from the newest features, security patches, bug fixes, and performance improvements. However, upgrading to a newer version of a library can cause rework, since existing interfaces may be broken in multiple ways. For instance, new parameters can be added to public API methods, return types may have been changed or methods may have been removed entirely. When these methods are used by client systems, these systems have to be adapted to work with the new version of the library.

As an example of a library update and its impact, see Figure 6.1, which shows a library class, Lib1, and a system class that uses it, System1. Two changes have been introduced in version 2 of Lib1: method foo added a parameter bar and method doStuff

1Parts of this chapter are submitted for review as “Semantic Versioning and Impact of Breaking Changes in the Maven Repository”, Journal of Systems and Software (JSS), 2015.
changed its return type from int to String. This causes two errors in System1: Calling `c1.foo()` now gives a compilation error since it expects an integer as parameter, and `c1.doStuff()` returns a String instead of an int, which also gives a compilation error.

The two changes to Lib1 are breaking changes (also called binary incompatibilities)\(^2\), which are changes that always require recompilation and adaptation of a client using the changed functionality. The amount of rework caused by breaking changes is a lower bound on the total amount of work that needs to be performed to update to a newer version of a library.

In this chapter, we introduce a new method to automatically estimate the amount of work required to update to newer versions of a dependency. We apply this method to the Maven Dependency Dataset, containing more than 100,000 Java libraries. With our method, we collect compilation errors in clients caused by all breaking changes, and from this data we collect statistics and perform regression analyses to estimate the number and spread of errors that need to be fixed after applying a library update. Our methodology is not only applicable to systems using libraries but it can be applied to estimate the size of ripple effects in any piece of software that uses the public interface of another piece of software, even within the same software system.

Several automated techniques have been proposed to adapt software systems to changes in interfaces of the libraries they use [31, 36, 54, 116]. However, we believe that most developers apply a pragmatic approach to change impact analysis, which means they simply apply the change, recompile their code and see where errors appear, and then find out where changes have to be made. This approach is the one we mimic here. With regards to changes in behavior, developers often simply re-run their test suites to see where behavior of the system has been changed. In this chapter, however, we do not consider changes in runtime behavior of systems but only investigate compilation errors.

This chapter is structured as follows. Section 6.2 contains the problem statement. Section 6.3 describes our change injection algorithm. Section 6.4 discusses implementation details. Section 6.5 explores the relationship between breaking changes and errors.


Figure 6.1: Example of a library update and impact on a system. Lib1 contains two changes, method `foo` with a new parameter `int bar`, and method `doStuff` with a return type of `String` instead of `int`. The compilation errors as a Java compiler would detect them are underlined in red.
6.2. Problem Statement

Functionality that is added to a library should be available in next versions of that library while, ideally speaking, the public API of a library should not change. This prevents rework for clients of that library when updating to a newer version. Each change has the potential to introduce compilation errors in clients using that library, or change the behavior of a library.

In this chapter, we aim to answer the following research questions:

- **RQ1**: What fraction of library updates contain breaking changes, and what impact do these changes have on client systems?
- **RQ2**: Which library characteristics are shared by libraries that frequently introduce breaking changes, and as a result, cause compilation errors?
- **RQ3**: How much work is associated with breaking changes in libraries?
- **RQ4**: Which factors are associated with a large dispersion of compilation errors in client systems caused by breaking changes?

The change types in the table are all guaranteed to cause rework when the changed unit is used. These breaking changes therefore point to places that are guaranteed to need adaptation. In reality, other places may need to be adapted as well but we only investigate the compilation errors as caused by these breaking changes since they can be obtained automatically.

We demonstrate that in practice, backward compatibility is neglected on a large scale in software libraries, which leads to a considerable amount of rework. We also investigate which properties libraries have in common that tend to introduce breaking changes. This helps to get a better understanding of underlying processes associated with the creation of breaking changes in libraries, and thus could also help to show software developers how to prevent such changes in their own libraries.

We assume that both the number of errors and the spread of these errors determine the amount of work that needs to be performed. The more errors need to be fixed, the more effort it takes to fix them. Also, the more spread out these errors are across methods, classes and packages, the more time it takes to understand the context of each different location and thus the more time it takes to fix them.

In this chapter, we will focus on the 10 most frequently occurring change types, which were determined by counting all breaking changes in the entire Maven repository. Section 6.5 shows a frequency table of the 20 most frequently occurring breaking change types. The analyzed breaking changes are the following:

1. **Method removals (MR)**: The removal of a method from a class. This leads to errors in all calls to that method.

Section 6.6 explores the relationship between breaking changes and the amount of work performed in libraries. Section 6.7 investigates factors associated with a large dispersion in errors across multiple methods. Section 6.8, 6.9, and 6.10 contain the discussion, threats to validity and related work, respectively, and Section 6.11 concludes the chapter.
2. **Class Removals** (CR): The removal of a complete class. This leads to errors when instantiating this class, when calling methods of this class, when other classes extend this class, or when other classes use fields of this class.

3. **Field removals** (FR): The removal of class fields. This leads to errors in all places where this field is used directly, if it is a public field.

4. **Parameter type change** (PTC): The change of a parameter type. This leads to errors in all method calls at which the old parameter type in the same position cannot be implicitly cast to the new type and no overloaded, pre-existing method with matching parameter types is available.

5. **Return type change** (RTC): The change of the return type of a method. This leads to errors in all places where the return type of that method is explicitly stored in a variable and the new return type of the method cannot be cast implicitly to the old type of the variable.

6. **Interface Removals** (IR): The removal of a complete interface. This leads to errors in classes that implement this interface and in polymorphic method calls.

7. **Nr. parameters change** (NPC): The addition or removal of parameters from a method. This leads to errors in all places where the method is being called with the old number of parameters and no overloaded, pre-existing method with matching parameter types is available.

8. **Method added to interface** (MAI): This leads to errors in all classes that extend the interface and do not yet implement the new method.

9. **Field type change** (FTC): The change of the type of a field. This leads to errors in all places where the old field type is still assumed and no implicit cast is available.

10. **Constant field removals** (CFR): A constant field has been removed. This leads to all places where the field is referenced.

To answer **RQ1**, we show the results of our analysis in Section 6.5. To answer **RQ2**, we investigate library characteristics of libraries introducing breaking changes, which can be found in Section 6.5.2. To answer **RQ3**, we investigate the influence of different change types on edit script size in Section 6.6. To answer **RQ4**, we perform an analysis of the dispersion of compilation errors in Section 6.7.

In the next section, we describe our change injection algorithm.

### 6.3. Change Injection Algorithm

A conceptual overview of our approach is shown in Figure 6.2. Source code of a client system \((S_x)\) is scanned and compiled with source code of a single dependency \(L_y\) of \(S_x\) (denoted with (1)). All breaking changes between \(L_y\) and its next version \(L_{y+1}\) are calculated, as well as the edit script to convert the first version into the second \((\Delta L_{y,y+1}\), denoted with (2)). Each breaking change is inserted individually in \(L_y\). Errors appearing in \(S_x\) after inserting these changes are then stored. The edit script size and breaking changes in \(\Delta L_{y,y+1}\) are combined to estimate the number of changed statements per breaking change. \(S_{x+1}\) denotes a next version of \(S_x\), which could have updated \(L_y\) to
6.4. IMPLEMENTATION

To study the effects of library updates on other libraries, we investigated the set of all source code artifacts from a snapshot of the Maven repository, dated July 30, 2011. This snapshot contains 144,934 binary jar files and 101,413 source jar files for a total of 22,205 different libraries. For a detailed description of this publicly available dataset, see [91].

$L_{y+1}$. Any breaking changes in $\Delta L_{y,y+1}$ would lead to work in the update from $S_x$ to $S_{x+1}$, if the changed code is actually used in $S_x$.

In this chapter, we refer to any library that includes another library (a client library) as $S_x$, and we refer to the included library as $L_y$. Although we denote a next version of $L$ with $L_{y+1}$, this does not mean that $L_{y+1}$ has to be a subsequent version of $L_y$. Any version of $L$ which has an release date after $L_y$ is included in the set of next versions of $L_y$.

![Figure 6.2: Conceptual overview of our approach.](image)

The procedure to inject library changes is formally described in Algorithm 1 and can be explained in more detail as follows. For each library $L$ (e.g., “JUnit”), all versions are collected (line 3). For each of these versions, a list of all libraries using $L_y$ is obtained (using $L_y$, line 5). For each library version $L_y$ (e.g., “Junit 3.8.1”) in the repository, a list of all future versions is created (line 6). For each pair of current and next version $U(L_y, L_{y+1})$ (the transitive closure over all next versions of $L_y$), all public API changes are determined ($\Delta L_{y,y+1}$, line 10). Each change $C \in \Delta L_{y,y+1}$ is inserted into $L_y$ and the compilation errors are collected in all systems $S_x$ that use $L_y$ (lines 11-22). First, all files in $S_x$ and $L_y$ are compiled and linked together ($S_x-L_y$, line 13). Then, pre-existing errors in $S_x-L_y$ are stored in $errStart$ (line 14).

A single change is then injected in the code of $S_x-L_y$ (line 15). Code is recompiled with the inserted change (line 16). Errors are again collected in $errEnd$ (line 17), and pre-existing errors are removed from $errEnd$ (line 18). The remaining errors are stored for this combination of a change, system, library and library update (line 19), and can later be grouped by change types, versions and libraries. Afterwards, the change is reverted (line 20).
1: \( \text{errStored} \leftarrow \emptyset \)
2: \textbf{for each} library \( L \) \textbf{do}
3: \hspace{1em} \textit{allVersions} \leftarrow \text{all versions of } L
4: \hspace{1em} \textbf{for each} version \( L_y \in \text{allVersions} \) \textbf{do}
5: \hspace{2em} \textit{usingL} \leftarrow \text{all source jars } S_x \text{ using } L_y \in \text{repository}
6: \hspace{1em} \textit{possibleUpdates} \leftarrow \text{all possible updates}
7: \hspace{2em} \{U \langle L_y, L_y+1 \rangle | L_y+1 \in \text{allVersions},
8: \hspace{3em} L_y+1 \text{ newer than } L_y \}
9: \hspace{1em} \textbf{for each} update \( U \langle L_y, L_y+1 \rangle \in \text{possibleUpdates} \) \textbf{do}
10: \hspace{2em} \Delta L_{y,y+1} \leftarrow \text{all changes between } L_y \text{ and } L_y+1
11: \hspace{2em} \textbf{for each} \( S_x \in \text{usingL} \) \textbf{do}
12: \hspace{3em} \textbf{for each} change \( C \in \Delta L_{y,y+1} \) \textbf{do}
13: \hspace{4em} \text{Compile code of } S_x-L_y
14: \hspace{4em} \text{errStart} \leftarrow \text{collect compile errors in } S_x-L_y
15: \hspace{4em} \text{Inject } C \text{ in code of } L_y
16: \hspace{4em} \text{Recompile code of } S_x-L_y \text{ with } C \text{ injected}
17: \hspace{4em} \text{errEnd} \leftarrow \text{collect compile errors in } S_x-L_y
18: \hspace{4em} \text{errors}(S_x,L_y,L_y+1,C) \leftarrow \text{errEnd} \text{ – errStart}
19: \hspace{4em} \text{errStored} \leftarrow \text{errStored} \cup \text{errors}(S_x,L_y,L_y+1,C)
20: \hspace{4em} \text{Revert } C \text{ in code of } L_y
21: \hspace{3em} \textbf{end for}
22: \hspace{2em} \textbf{end for}
23: \hspace{1em} \textbf{end for}
24: \textbf{end for}
25: \textbf{end for}

\textbf{Algorithm 1}: Change injection algorithm.
In total, 126,070 update pairs \((L_y, L_{y+1})\) have been extracted from the Maven repository. Out of all these potential updates, 48,143 pairs contain an \(L_y\) that is actually used by an \(S_x\). Out of these 48,143 pairs, 3,260 pairs actually contain breaking changes (6.8%).

From the build scripts (\texttt{pom.xml}) of each jar file, dependencies on other jar files were extracted. Source code in each source jar was automatically extracted and was compiled with the Eclipse JDT Core API\(^3\), which is the compiler of the Eclipse IDE. The Maven build system itself was used to obtain a list of other libraries that \(S_x\) and \(L_y\) need to compile successfully. The binary class files for each of these dependencies where added to the classpath of the compiler. Visitors for classes, methods and parameters were used to obtain data. The entire repository was processed on the DAS-3 Supercomputer using 100 nodes in parallel in approximately 20 days, for an aggregate running time of 5.5 years.

Clirr was used to obtain breaking changes between two versions of a library. Clirr compares two versions of a library on the binary level and reports any changes in its public API. We only extract the 10 most common types of changes found by Clirr, as described in Section 6.2. The results of this analysis were stored in a MySQL database.

In this chapter, we perform several analyses on the same dataset but with a different number of observations. This is due to different selection criteria and exclusion of observations because of missing data, which depends on the specific analysis performed.

In the next section, we first show the frequency of occurrence of breaking changes and errors in the Maven repository. We then investigate the relationship between the number of breaking changes and the number of errors.

### 6.5. Breaking Changes and Errors

For reference, Table 6.3 shows again the 20 most frequently occurring breaking change types and their frequency of occurrence in the entire Maven repository. The breaking changes in these table are obtained from the 126,070 potential updates \((L_y, L_{y+1})\). The most common breaking change is the method removal, with a frequency of 177,480. The next two frequently occurring change types are class and field removals. In the rest of the chapter, we only use the top 10 from this table.

Table 6.4 shows overview statistics for the 10 different types of breaking changes detected by applying Algorithm 1 to the entire Maven repository.

The table shows the number of breaking changes and the number of compilation errors these changes cause. For instance, class removals occur 168,743 times and cause a total of 1,645,518 compilation errors when applying the algorithm to the entire repository. The most frequently occurring breaking change is the method removal, occurring 177,480 times in the repository and causing 1,645,518 compilation errors in total. For method removals, there are 3,983 unique jar files that contain compilation errors caused by breaking changes in 505 unique jar files. Another type of frequently occurring breaking change is the class removal, which appears 126,334 times in our dataset and causes 1,645,518 errors.

The average number of errors per breaking change is also shown in Table 6.4. It shows that a constant field removal (\(CFR\)) has the highest average number of errors per change: 52.18. Furthermore, field type changes (45.16), field removals (32.8) and parameter type

\(^3\)http://www.eclipse.org/jdt/core
Table 6.3: The 20 most occurring types of breaking changes detected by Clirr.

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Method has been removed (MR)</td>
<td>177,480</td>
</tr>
<tr>
<td>2</td>
<td>Class removed (CR)</td>
<td>168,743</td>
</tr>
<tr>
<td>3</td>
<td>Removed public field (FR)</td>
<td>126,334</td>
</tr>
<tr>
<td>4</td>
<td>Parameter has changed its type (PTC)</td>
<td>69,335</td>
</tr>
<tr>
<td>5</td>
<td>Return type of method has been changed (RTC)</td>
<td>54,742</td>
</tr>
<tr>
<td>6</td>
<td>Removed interface (IR)</td>
<td>46,852</td>
</tr>
<tr>
<td>7</td>
<td>Number of arguments changed (NPC)</td>
<td>42,286</td>
</tr>
<tr>
<td>8</td>
<td>Method has been added to an interface (MAI)</td>
<td>28,833</td>
</tr>
<tr>
<td>9</td>
<td>Changed type of field (FTC)</td>
<td>27,306</td>
</tr>
<tr>
<td>10</td>
<td>Constant field removed (CFR)</td>
<td>12,979</td>
</tr>
<tr>
<td>11</td>
<td>Removed from the list of superclasses</td>
<td>9,429</td>
</tr>
<tr>
<td>12</td>
<td>Field is now final</td>
<td>9,351</td>
</tr>
<tr>
<td>13</td>
<td>Accessibility of method has been decreased</td>
<td>6,520</td>
</tr>
<tr>
<td>14</td>
<td>Accessibility of field has been weakened</td>
<td>6,381</td>
</tr>
<tr>
<td>15</td>
<td>Method is now final</td>
<td>5,641</td>
</tr>
<tr>
<td>16</td>
<td>Abstract method has been added</td>
<td>2,532</td>
</tr>
<tr>
<td>17</td>
<td>Added final modifier</td>
<td>1,260</td>
</tr>
<tr>
<td>18</td>
<td>Field is now static</td>
<td>726</td>
</tr>
<tr>
<td>19</td>
<td>Added abstract modifier</td>
<td>564</td>
</tr>
<tr>
<td>20</td>
<td>Field is now non-static</td>
<td>509</td>
</tr>
</tbody>
</table>

changes (13.79) cause a relatively large number of compilation errors as compared to other change types. On average, a breaking change causes 18.72 errors.

Applying all possible library updates and collecting all compilation errors gives a total of 595,158 breaking changes of the 10 most occurring change types and a total of 11,139,014 compilation errors because of these changes. To answer RQ1: Breaking changes are widespread. Not only do libraries introduce them: client systems actually rely on the broken entities causing 18.72 compilation errors per change on average, leading to rework at the client side to fix and test the required adaptations.

6.5.1. The relationship between breaking changes and errors

To further investigate the relationship between breaking changes and the number of errors caused by these changes, we calculate the correlation between these properties. The Spearman rank correlation between the number of breaking changes in $\Delta L_{y,y+1}$ and the number of errors in $S_x$ caused by these changes is 0.65 ($p = 0$), indicating a significant positive relationship between breaking changes and compilation errors caused by these changes, as expected.

To investigate further how many errors each breaking change introduces, we perform the following regression analysis:

$$\ln(NE)_i = \beta_1 \ln(NBC)_i + \epsilon_i$$

with $NE$ being the number of errors in $S_x$ and $NBC$ being the number of breaking changes in $\Delta L_{y,y+1}$. We do not estimate a constant since each error must be caused by a breaking change. Both $NE$ and $NBC$ are log-transformed because the data is lognormally dis-
6.5. Breaking Changes and Errors

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Frequency</th>
<th>#Errors</th>
<th>#E/F</th>
<th>#sys</th>
<th>#uniq</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MR</td>
<td>177,480</td>
<td>1,524,498</td>
<td>8.59</td>
<td>8,328</td>
<td>960</td>
</tr>
<tr>
<td>2</td>
<td>CR</td>
<td>168,743</td>
<td>1,645,518</td>
<td>9.75</td>
<td>3,983</td>
<td>505</td>
</tr>
<tr>
<td>3</td>
<td>FR</td>
<td>126,334</td>
<td>4,143,723</td>
<td>32.80</td>
<td>8,028</td>
<td>960</td>
</tr>
<tr>
<td>4</td>
<td>PTC</td>
<td>69,335</td>
<td>956,314</td>
<td>13.79</td>
<td>5,357</td>
<td>547</td>
</tr>
<tr>
<td>5</td>
<td>RTC</td>
<td>54,742</td>
<td>288,939</td>
<td>5.28</td>
<td>4,478</td>
<td>433</td>
</tr>
<tr>
<td>6</td>
<td>IR</td>
<td>46,852</td>
<td>95,250</td>
<td>2.03</td>
<td>1,657</td>
<td>130</td>
</tr>
<tr>
<td>7</td>
<td>NPC</td>
<td>42,286</td>
<td>533,741</td>
<td>12.62</td>
<td>5,701</td>
<td>713</td>
</tr>
<tr>
<td>8</td>
<td>MAI</td>
<td>28,833</td>
<td>126,427</td>
<td>4.38</td>
<td>4,746</td>
<td>562</td>
</tr>
<tr>
<td>9</td>
<td>FTC</td>
<td>27,306</td>
<td>1,233,095</td>
<td>45.16</td>
<td>4,324</td>
<td>485</td>
</tr>
<tr>
<td>10</td>
<td>CFR</td>
<td>12,979</td>
<td>677,234</td>
<td>52.18</td>
<td>3,354</td>
<td>317</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>595,158</td>
<td>11,139,014</td>
<td>18.72</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: The types of changes detected. Frequency = the number of times this change type occurred in an update, #Errors = The number of errors this update type caused in all $S_X$, #E/F = the average number of errors per breaking change, #sys = The number of distinct $S_X$ that contain errors because of this update, #uniq = The number of different updates of $L_Y$ that contain this change.

The results can be found in Table 6.5. The model is highly significant with a $p$-value of 0 and an adjusted $R^2$ of 88.79%. The estimated slope coefficient of NBC is 1.683, indicating that if the number of breaking changes increases by 1%, the number of errors is expected to increase by 1.683%.

```
<table>
<thead>
<tr>
<th>Dependent variable</th>
<th>In(NE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of observations</td>
<td>2,269</td>
</tr>
<tr>
<td>$R^2$</td>
<td>0.8879</td>
</tr>
<tr>
<td>Model $p$-value</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
```

Table 6.5: Regression analysis to estimate the relationship between breaking changes and errors

### 6.5.2. Libraries with Large Impact

To assess which library characteristics cause a large number of compilation errors in dependent systems, we investigate the correlation of breaking changes and errors with two library properties: the maturity and the size of a library.

We use the index of a release (any release, major, minor or patch) as a proxy for the maturity of a library, starting with 1 from the oldest release. We assume that the more releases a certain library had before the current release, the more mature it is. Alternative measures, such as the number of days since the first release, were considered inferior since a library can have a single release and another release 2 years later, which would indicate a mature library. The size of a library is measured as the number of methods in a library.

These properties are investigated for the following reason. We expect that the size of a library increases as the library matures. For this reason, we expect that the number of
methods and the release index are positively correlated. We also expect that it becomes increasingly hard for library developers to maintain backward compatibility as the maturity of a library increases, simply because the library has a larger interface that can be broken. Therefore, the correlation between the maturity and the number of breaking changes in a release is expected to be positive as well. The number of compilation errors is expected to have comparable correlations with these two properties, because of the direct relationship between breaking changes and the number of errors caused by these changes.

<table>
<thead>
<tr>
<th></th>
<th>#Breaking changes</th>
<th>#Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of methods</td>
<td>0.3291 ($p = 0$)</td>
<td>0.3392 ($p = 0$)</td>
</tr>
<tr>
<td>Release index</td>
<td>-0.015 ($p = 0.153$)</td>
<td>0.1078 ($p = 0$)</td>
</tr>
</tbody>
</table>

Table 6.6: Spearman rank correlations between the number of breaking changes, number of errors, number of methods, and the release index of a library.

Spearman rank correlation coefficients of these properties can be found in Table 6.6. There is a correlation of 0.3291 between the number of methods in a library and the number of breaking changes in that library, meaning that bigger libraries indeed tend to introduce more breaking changes.

The correlation between the release index and the number of methods in that library turns out to be only marginally positive with a value of 0.0278, meaning that there is practically no correlation between these two properties: most libraries do not seem to grow in the number of methods through time. This is contrary to our expectation. There is a negligible correlation between the number of breaking changes and the release index, indicating that libraries do not introduce more breaking changes when they become more mature, which is also contrary to our expectation. The correlation between the number of errors and the number of methods is also positive, indicating that larger libraries cause more compilation errors.

To answer **RQ2**: Bigger libraries tend to introduce more breaking changes and errors. Libraries do not grow when they become more mature, on average, and more mature libraries do not introduce more breaking changes.

### 6.6. Breaking Changes and Edit Script Size

In this section, we explain our approach to obtain an estimate of the size of work which contains breaking changes for different change types.

#### 6.6.1. Explanation

To assess the amount of work that a library developer has performed when breaking changes are introduced, we calculate the size of the edit script to convert $L_y$ into $L_{y+1}$. As was described in Chapter 3, the size of the edit script represents the total number of statements that must be inserted, deleted, moved or updated to transform $L_y$ into $L_{y+1}$. The size of the edit script cannot be directly translated into effort in terms of man-hours since two edit scripts of the same length can each take a different time to implement, but
it can nonetheless serve as an indicator for this effort. The edit script size is used as follows. First, the number of different change types in each update $\Delta L_y, y+1$ is determined. Then, we calculate the edit script size to update $L_y$ to $L_y+1$. From this data, we estimate the amount of work that is associated with a single breaking change with a regression model.

Algorithm 2 formally describes our approach to obtain edit script size data. The procedure to obtain all possible update pairs (lines 1-7) is similar to Algorithm 1. The algorithm calculates the edit script size and the number of breaking changes for all library updates.

```plaintext
1: for each library $L$ do
2:    allVersions ← all versions of $L$
3: for each version $L_y \in allVersions$ do
4:    possibleUpdates ← all possible updates
5:       $\{ U(L_y, L_{y+1}) \mid L_{y+1} \in allVersions,$
6:       $L_{y+1}$ newer than $L_y \}$
7: for each $(U(L_y, L_{y+1}) \in possibleUpdates$ do
8:    ess($L_y, L_{y+1}$) ← calcEditScriptSize($L_y, L_{y+1}$)
9: for each change type $c \in changeTypes$ do
10:       nrChanges($c, L_y, L_{y+1}$) ← $|\{ c \in \Delta L_y, y+1 \}|$
11: end for
12: end for
13: end for
14: end for
15: function calcEditScriptSize($L_y, L_{y+1}$)
16:    editScriptSize$_{L_y, y+1}$ ← 0
17: for each java file $\in L_y$ do
18:    $f_{y+1}$ ← find match for $f_y$ in $L_{y+1}$
19:    editScript$\_f_y_{y+1}$ ← calculate $\Delta f_{y+1}$
20:    editScriptSize$_{L_y, y+1}$ += $\mid editScript\_f_y_{y+1}\mid$
21: end for
22: return editScriptSize$_{L_y, y+1}$
23: end function

Algorithm 2: Procedure to obtain edit script size data.
```

To calculate the edit script size (lines 16-24), the following steps are taken. For each java file in $L_y$, the corresponding next version of the file is found in $L_{y+1}$ (line 19). The edit script to convert $f_y$ into $f_{y+1}$ is calculated (line 20), and the size of this edit script is added to the total edit script size of $(L_y, L_{y+1})$ (line 21). This data serves as dependent variable in the regression model of rework estimation. Finally, the number of times the 10 different update types occur in $\Delta L_y, y+1$ is calculated and stored (line 10). These numbers serve as the independent variables in our regression model.
6.6.2. **ChangeDistiller**

ChangeDistiller was used to calculate edit scripts in $\Delta L_{y,y+1}$ [41]. ChangeDistiller works on the level of individual source files, but was adapted to work on the level of jar files. This can be seen in lines 16-24 of Algorithm 2. For each two versions of a java source file, ChangeDistiller calculates the edit script to convert the first version into the second. In our approach, we see each jar file as a collection of java files. Each java file in the jar file is iterated and the corresponding next version of that file is found in $L_{y+1}$. The length of the edit script to convert $f_y$ into $f_{y+1}$ is added to the total edit script size for the jar file. To match versions of files, filenames that matched directly are considered to be two versions of the same file (for instance, two files with a filename ending in `java/src/foo/bar/Bar.java` are considered direct matches). Files that did not have a direct counterpart in the other version, meaning they were deleted, added, or moved, were matched using a token-based similarity algorithm similar as used by ChangeDistiller itself. When two file pairs exceeded the default token-based similarity threshold of 0.8, these files were considered to be moved. Our adaptation of ChangeDistiller returns a single number that represents the length of the edit script to convert $S_x$ into $S_{x+1}$. For each update in the Maven repository, this number is stored in our database.

6.6.3. **Regression Result**

From the data acquired through Algorithm 2, we estimate the influence of each breaking change type in $\Delta L_{y,y+1}$ by including the number of occurrences of each type as independent variables. The dependent variable is the size of the edit script of $\Delta L_{y,y+1}$. Table 6.7 shows the results of this regression, which is based on the 3,260 pairs containing breaking changes as described in Section 6.4. The actual number of observations is only 2,447 due to the exclusion of observations with missing data.

<table>
<thead>
<tr>
<th>Dependent variable</th>
<th>$ess(L_{y,y+1})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of observations</td>
<td>2,447</td>
</tr>
<tr>
<td>$R^2$</td>
<td>58.83%</td>
</tr>
<tr>
<td>Adjusted $R^2$</td>
<td>58.68%</td>
</tr>
<tr>
<td>Model $p$-value</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Indep.</th>
<th>#</th>
<th>Coeff.</th>
<th>Std. Err</th>
<th>beta</th>
<th>p-value</th>
<th>95% C.I.</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>0</td>
<td>5.001</td>
<td>1.096</td>
<td>-</td>
<td>0</td>
<td>2.851 - 7.151</td>
</tr>
<tr>
<td>MR</td>
<td>1</td>
<td>2.415</td>
<td>0.110</td>
<td>0.346</td>
<td>0</td>
<td>2.200 - 2.630</td>
</tr>
<tr>
<td>CR</td>
<td>2</td>
<td>0.539</td>
<td>0.109</td>
<td>0.069</td>
<td>0</td>
<td>0.325 - 0.753</td>
</tr>
<tr>
<td>FR</td>
<td>3</td>
<td>0.818</td>
<td>0.187</td>
<td>0.059</td>
<td>0</td>
<td>0.451 - 0.753</td>
</tr>
<tr>
<td>PTC</td>
<td>4</td>
<td>1.921</td>
<td>0.141</td>
<td>0.208</td>
<td>0</td>
<td>1.646 - 2.197</td>
</tr>
<tr>
<td>RTC</td>
<td>5</td>
<td>2.021</td>
<td>0.221</td>
<td>0.141</td>
<td>0</td>
<td>1.587 - 2.454</td>
</tr>
<tr>
<td>IR</td>
<td>6</td>
<td>0.684</td>
<td>0.218</td>
<td>0.043</td>
<td>0</td>
<td>0.256 - 1.113</td>
</tr>
<tr>
<td>NPC</td>
<td>7</td>
<td>2.734</td>
<td>0.191</td>
<td>0.204</td>
<td>0</td>
<td>2.360 - 3.108</td>
</tr>
<tr>
<td>MAI</td>
<td>8</td>
<td>2.534</td>
<td>0.193</td>
<td>0.178</td>
<td>0</td>
<td>2.156 - 2.913</td>
</tr>
<tr>
<td>FTC</td>
<td>9</td>
<td>1.239</td>
<td>0.367</td>
<td>0.049</td>
<td>0</td>
<td>0.518 - 1.960</td>
</tr>
<tr>
<td>CFR</td>
<td>10</td>
<td>omitted due to collinearity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.7: Regression analysis on the edit script size and different change types in libraries.
As can be seen in Table 6.7, the model as a whole is highly significant ($p = 0$) and has an adjusted $R^2$ of 58.68%, indicating that more than 58% of the variability in the edit script size between $L_y$ and $L_{y+1}$ is explained by the 10 different change types in the model. The model shows that all variables are significant at the 95% confidence interval, indicating that the all variables contribute significantly to the total edit script size in $\Delta L_{y,y+1}$. The coefficients in the model indicate the size of the performed rework in terms of tree edit operations to update a library from $L_y$ to $L_{y+1}$. For instance, the change type method removal (MR) has a coefficient of 2.415, indicating that a method removal in $\Delta L_{y,y+1}$ takes 2.415 edit script operations, on average.

As the table shows, all 10 breaking change types are associated with a significant edit script size, but some changes have a larger coefficient than others. For instance, a class removal and an interface removal only represent an edit script size of 0.539 and 0.684, respectively. This could be explained through the average size of classes or interfaces that are removed, which could be smaller than the average class. The constant of 5.0 indicates that the average library update which contains breaking changes has a “base level” average of 5 edit script lines.

As an example of the expected edit script size in a library update, consider a library which removes a class with 10 methods and two private fields in its next version. The predicted edit script size would then be $5.001 + 1 \times 0.539 + 5 \times 2.415 + 2 \times 0.818 = 19.251$. The constant of 5 indicates that a library change without any of the included change types takes an edit script size of 5, on average.

Comparing the standardized coefficients ($\beta$) for each of the 10 change types, it can be seen that the method removal (MR) and the parameter type change (PTC) have the largest influence on the total edit script size, with a $\beta$ of 0.346 and 0.208, respectively. Field removals, class removals and field type changes turn out to have relatively little influence on the total edit script size, with $\beta$'s of 0.059, 0.069, and 0.049, respectively. The constant field removal $CFR$ correlates too much with other change types and is therefore excluded automatically from the regression.

### 6.7. Dispersion Estimation

#### 6.7.1. Explanation

When fixing compilation errors caused by a library update, not only the number of errors is relevant in the estimation of total rework, but the dispersion of these errors across different places in the code is relevant as well. We expect that a change that causes 10 errors inside a single file is easier to fix than a change that causes 10 errors in 10 different files, since the code and context of each file has to be understood separately before its errors can be fixed. Fixing errors in multiple different files is therefore expected to take more time than fixing errors inside a single file.

Figure 6.8 shows the concept underlying our analysis. As can be seen in this figure, $L_{y+1}$ is used by three different libraries, $S_{x1}$, $S_{x2}$ and $S_{x3}$. Library $S_{x1}$ calls a method that was changed in $\Delta L_{y,y+1}$, and as a result, 2 compilation errors in one distinct method are introduced. Library $S_{x2}$ calls a method that was added, but this does not introduce any errors.

Library $S_{x3}$ calls a method that was removed, and therefore, 3 errors in 2 distinct
methods are introduced. One error in \( S_{x3} \) is not directly related to the call to \( L_{y+1} \) but is a cascading error, caused by the other method that directly calls \( L_{y+1} \). Overall, there are 3 distinct methods that are impacted because of the update of \( L_y \) to \( L_{y+1} \).

We perform a regression analysis to test factors influencing the dispersion of errors in different libraries \( S_x \) with changes in a dependency \( L_y \). As a measure of the dispersion of compilation errors across systems, we count the number of distinct files of all \( S_x \) that use \( L_y \) and which contain one or more compilation error after the update to \( L_{y+1} \).

### 6.7.2. Explanation of Independent Variables

We include 3 independent variables in our analysis, which are expected to explain a part of the variability in error dispersion across systems. These factors are the release index of \( L_y \), the number of methods in \( L_y \) and the relative usage frequency of \( L_y \) in the Maven repository. The rationale for inclusion of these independent variables is as follows. The number of methods in \( L_y \) is a measure of the size of that library. We expect that a larger size of \( L_y \) causes methods of this library to be used at more separate places in \( S_x \), because it is expected to contain more separate pieces of functionality.

We include the release index number as independent variable to measure maturity for the same reason it was included in the analysis in Section 6.5.2: we expect that libraries tend to increase the amount of functionality they provide over time as the library matures, which would lead to a larger scattering of errors across \( S_x \). With the inclusion of both the release index and library size as independent variables, we can test for the influence of both independent variables separately, while keeping the others constant.

We include the popularity as independent variable to correct for usage frequency of libraries, which is defined as the number of libraries that use \( L_y \) divided by the total number of libraries in the repository (144,934). When a library \( L_y \) is used more frequently, it will cause more errors in systems using it simply because it is used more. This independent variable corrects for that effect.

Note that these three variables are all properties of \( L_{y+1} \), and not of \( S_x \), although the
errors appear in $S_x$. We include properties of $L_{y+1}$ in this analysis since we expect that the error dispersion in $S_x$ is, to a degree, a property of the library causing the errors. Since error dispersion is also expected to be influenced by properties of $S_x$ itself, we do not expect that the included properties are fully able to explain all variability in the model, which would lead to a relatively low $R^2$.

We include the natural logarithm of the number of distinct files and the number of methods in $L_y$ because data analysis shows that these variables are lognormally distributed.

### 6.7.3. Regression results

Table 6.9 shows the result of this analysis.

<table>
<thead>
<tr>
<th>Dependent variable</th>
<th>$\ln(#\text{distinct files with error in } S_x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of observations</td>
<td>3,690</td>
</tr>
<tr>
<td>$R^2$</td>
<td>13.54%</td>
</tr>
<tr>
<td>Adjusted $R^2$</td>
<td>13.33%</td>
</tr>
<tr>
<td>Model $p$-value</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Independents</th>
<th>Coeff.</th>
<th>beta</th>
<th>Std.Err</th>
<th>$p$-value</th>
<th>95% C.I.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release index ($L_y$)</td>
<td>-0.007</td>
<td>-0.075</td>
<td>0.003</td>
<td>0.006</td>
<td>-0.013 - 0.002</td>
</tr>
<tr>
<td>$\ln(\text{nr. methods} (L_y))$</td>
<td>0.313</td>
<td>0.381</td>
<td>0.022</td>
<td>0.022</td>
<td>0.269 - 0.357</td>
</tr>
<tr>
<td>Popularity ($L_y$)</td>
<td>-47.68</td>
<td>-0.019</td>
<td>67.26</td>
<td>0.478</td>
<td>-179.6 - 84.27</td>
</tr>
<tr>
<td>constant</td>
<td>-0.262</td>
<td>-</td>
<td>0.131</td>
<td>0.045</td>
<td>-0.519 - -0.006</td>
</tr>
</tbody>
</table>

Table 6.9: Regression analysis to analyze factors associated with a large error dispersion.

The model has an adjusted $R^2$ of 13.33%, which is significant with a $p$-value of $0^4$. The release index and the natural logarithm of the number of methods in $L_y$ are significant predictors for the outcome variable but the popularity of $L_y$ is not. The beta coefficients show that the size of $L_y$ is the most important factor in explaining the dispersion of errors across $S_x$, with a beta of 0.381. The release index of a library has a coefficient that is significant but practically speaking not substantially different from 0.

This means that, after correcting for library popularity and maturity, the number of methods in a library is a significant predictor for the dispersion of errors in client systems. This could be explained by our theory that larger libraries contain more separate pieces of functionality, which have a bigger chance of ending up at different places in client systems.

To answer **RQ4**: *The size of a library tends to increase the dispersion of errors in client systems.*

### 6.8. Discussion

#### 6.8.1. Edit scripts as rework measure

The edit script size to convert a library into one of its next versions was chosen to represent the performed rework in that update. It was preferred over other measurements of

---

4For an explanation of this low $R^2$, see Section 6.9.4.
rereform, such as the difference between the number of lines of code (LOC) between two library versions, because we consider it to be the most detailed representation of changes available. Differences in LOC have the problem that when the contents of a method or file completely changes but the LOC stays exactly the same, no difference is detected. Differencing as used by version control systems was also considered to be inaccurate for the purpose of this analysis since it is sensitive to irrelevant changes in whitespace and comments.

6.8.2. **Semantic Versioning**

As was described in Chapter 3, Semantic versioning\(^5\) is a movement in software engineering that promotes strict rules regarding the increment of major, minor and patch version numbers (e.g. “v2.3.4” where 2 is the major version number, 3 is the minor version number and 4 is the patch version number) when releasing a new library. Ideally, incompatible API changes should only be included when the new release is a major release, i.e. the major version number is incremented. Minor and patch releases should only contain backward-compatible changes and bug fixes, respectively. In this chapter, we did not make a distinction between major, minor or patch releases because the goal of this chapter is to determine the amount of rework required when a breaking change in a library dependency exists, whether that change is included in a major release or not. Chapter 3 investigated the application of Semantic versioning and its impact in more detail.

6.8.3. **Other Applications**

The approach as described in this chapter can be used to perform general change impact analysis as well, without breaking changes or libraries. For instance, the Eclipse compiler could also be used in a similar way to perform a “mutation analysis” of public interfaces, where random changes are injected in the public API to determine how systems using that API would react. This is similar to the work performed in this chapter, but would not be limited to breaking changes that actually occur. The data as obtained by our approach could also be used to construct a “profile” for a library, which would indicate the expected amount and spread of errors when updating a certain library. We expect that different libraries will have different profiles and some libraries are easier to update than others, partly due to design and partly due to the problem the library solves.

6.9. **Threats to Validity**

6.9.1. **Error counting**

With respect to internal validity, when compiling Java code with Eclipse and the JDT, compilation errors can “mask” other compilation errors. If a package import cannot be found, for instance, the compiler will never reach compilation errors further down the file because it stops compiling. It is unknown how many times this happens in our dataset. It would mean, however, that the true amount of rework to fix “masking” compilation errors would be underestimated using our approach since since fixing these er-

---

\(^5\) [http://www.semver.org](http://www.semver.org)
rors would reveal new, previously unreported compilation errors which would have to be fixed in turn. On the other hand, collections of compilation errors can be manifestations of a change in the same object, such as a removed method, field or parameter. Fixing such errors would be faster than fixing the same amount of unrelated errors, for instance with a global search-and-replace action. More research is needed to assess the strength of over- or underestimation of the rework effect due to these two reasons.

6.9.2. Transitive closure of upgrade pairs
Another potential issue of internal validity is the calculation of the transitive closure of all future versions of each library. This was done for three reasons. First, in practice, developers can update to any later version of a library from their current version and can skip versions “in between”. Second, the number of breaking changes between, for example, version 1 and version 3 of a library is not the sum of the breaking changes between version pairs 1-2 and 2-3. This also gives rise to the need to consider each update pair separately. Third, the regression analyses as performed in this chapter are not influenced by the larger amount of data since, statistically, estimated coefficients will not be influenced by the possible duplication in the data. We use robust regression and robust standard errors to mitigate any risk rising from data duplication, which are common statistical methods to deal with this type of problem [6].

6.9.3. Selection of independent variables
With respect to construct validity, we chose a set of independent variables in our analysis, such as maturity, size, and the popularity of a library as variables that could influence rework effort and dispersion of errors across systems. We did not have the goal to create an exhaustive list of all possible variables that could influence rework effort and dispersion. There are possibly other variables at work that influence rework and dispersion besides the ones investigated in this chapter, which could significantly alter our analysis and conclusions. The restriction to investigate only the 10 most frequently occurring change types could also influence our conclusions. In future work, the influence of other variables can be investigated further.

6.9.4. Small $R^2$
The regression analysis in Table 6.7 has an $R^2$ of 58.68%, while the regression analysis in Table 6.9 has an $R^2$ of only 13.33%. Concerns may rise about the limited explanatory power of these models. The first model only includes 10 most frequently occurring breaking changes, which means that any change that cannot be explained by the independent variables is not taken into account. In other words, more than 40% of the edit script is explained by changes that cannot be related to the 10 breaking change types. This is expected since it is likely that there will be a large amount of changes which are not associated with any breaking change, such as methods that have a changed implementation without changing their method headers.

The model in Table 6.9 has an even lower $R^2$, but this is expected given the setup of the experiment. The model incorporates factors associated with $L_y$ to explain the dispersion of errors in $S_x$. The most important factors that would explain this dispersion can likely be found in $S_x$. For instance, a modular design of $S_x$ with better encapsulation
could mean that the calls to a certain library, for instance a database library, are located in a single class. Measuring these factors, however, are not the goal of the model. The \( p \)-value of the model (0) indicates that the model is highly significant. Because of the low \( R^2 \) of the model, the model should only be used for explanatory purposes, and not for prediction.

### 6.9.5. Generalizability

We only consider the impact of changing library code on other libraries using that code. What the degree of external validity for our study is, i.e. whether our results can be generalized to non-library software systems, needs to be determined in further research. It is also unknown whether the statistical patterns as found in this chapter will be found in industrial, closed-source software systems or systems written in other languages than Java. Further research is also needed to investigate whether these patterns hold in other software repositories, such as GitHub.

### 6.10. Related Work

The methodology that we use can be regarded as a change impact analysis technique, for which there already exist several alternative approaches [9, 94, 118]. For instance, call graph analysis techniques can obtain a graph that can point developers to places where rework is expected, such as done by Ren et al [94]. Other techniques use correlations of file properties or historically changed file pairs as a basis to determine files that are likely to change together, as in [119]. For an overview of change impact analysis techniques, see [72].

Cossette et al. [29] perform a manual retroactive study on API incompatibilities to determine the correct adaptations to migrate from an older to a newer version of a library. They also aim to determine recommender techniques for specific update types. In contrast, our work inserts updates automatically to find out what breaks, and only gives a global indication of the amount of work required to perform an update in terms of the number of compilation errors and the number of places that have to be fixed. Our approach does not provide any guidance how to perform an update but can point to places where work has to be performed.

Dig et al. [38] also performed a manual analysis of breaking changes occurring in five software libraries. They found that breaking API changes occur frequently in these systems and that the largest part of the breaking changes are behavior-preserving refactorings, which cause rework in client systems.

A lot of techniques have been proposed to automatically determine the refactorings to adapt to breaking changes in a library [31, 36, 54, 116]. The goal of our work is not to suggest or automatically carry out fixes to breaking changes, but simply to track and predict their impact.

Our automated change injection mechanism bears similarities to approaches applied in the field of automated software testing and, more specifically, error injection. Error injection techniques inject faults to find out if the resulting errors are covered by test cases. The goal of this chapter is different, however: we want to determine the amount of rework caused by applying library updates. For an overview of error injection tech-
niques, see [39].

Complete migrations to other libraries providing similar functionality has been investigated by [105]. In contrast to our work, Teyton et al. are concerned with a migration between different libraries performing similar functionality, rather than a migration between different versions of the same library.

6.11. CONCLUSION

We investigated the occurrence and the effects of 10 frequently occurring types of breaking changes in a set of more than 100,000 Java libraries. We presented a new method to systematically determine all compilation errors caused by these changes. We also presented an approach to obtain rework estimation data, which was used to calculate the required rework to implement different changes in dependencies. We investigated factors associated with a large dispersion of compilation errors across client systems.

In this chapter, we have demonstrated that:

• breaking changes are common;
• breaking changes occur in parts of library APIs that are actually used, and lead to compilation errors that need to be fixed when updating to a newer version of a dependency;
• bigger libraries introduce more breaking changes but more mature libraries do not introduce more breaking changes;
• there exists a significant difference in the amount of work performed in updates with and without breaking changes;
• The size of a library tends to increase the dispersion of errors in client systems induced by breaking changes in dependencies.
In previous chapters, we have calculated the number of compilation errors which provides an estimate of the amount of work needed to implement a certain change. Ideally, we would like to be able to express a certain amount of work in monetary terms or in time. In this chapter, we introduce a method that enables estimating the amount of time or money it takes to implement a software system of a certain size by using compression. There exist large differences in the amount of code that is needed in different programming languages to implement the same piece of functionality, which is in turn caused by differences in expressiveness or verbosity between programming languages. We use compression and Kolmogorov complexity to measure the amount of information present in a certain amount of code, which makes it possible to compare productivity levels of languages using a universal scale. We show that compression ratios of programming languages correspond to the notion of “high-levelness” and “low-levelness” of a language, and we demonstrate that the calculated compression ratios correlate to a large degree with estimates from software cost estimation methods based on function points. We calculate compression ratios on a large benchmark of industrial software systems written in a number of different programming languages. From the same benchmark, we also obtain an estimate for the average number of lines of code written per programmer per hour in a certain programming language.

7.1. Introduction
When writing the same piece of functionality in different programming languages, the result may very well be two pieces of code that differ in terms of number of lines and verbosity. One piece of code may be written in a very compact but highly expressive language, such as, for instance, Scala, and another may have been written in a low-level language such as Assembler. This is demonstrated in Figure 7.1 and Figure 7.2.
As can be seen, the differences between the two pieces of code are large but the result is exactly the same. Of course, the function println in Scala may ultimately execute comparable code as is written out explicitly in the Assembler example, but programmers do not have to type it themselves. Today, most developers would typically not chose a language such as Assembler to write software in, which shows that overall, developers have a tendency to prefer higher-level programming languages.

Figure 7.1: “Hello, World!” example in Scala.

In this chapter, we present a new approach to enable comparing functionality written in different programming languages, such as in the example above. The same approach can also be used to obtain an estimate of the amount of work is needed in time or monetary terms to implement a software system using a certain programming language. We use concepts from information theory, based on compression and Kolmogorov complexity, to reduce a piece of code to its true information content. We define the true information content of a piece of code to be the minimal amount of information that is needed to represent the functionality that it executes. We obtain an estimate of the size of the true information content by compressing all source code in a software system. The size of the compressed source code in bytes is a proxy for the size of the true information content of a software system.

We use a large benchmark of industrial software systems as data source for our analysis. We also use this benchmark to obtain code change (churn) estimates, which enables estimation of productivity per programmer per hour for different languages. Combined with the results of compressing source code, this makes it possible to estimate the rebuild value of a software system when rewritten in another programming language. In the consultancy practice of the Software Improvement Group (SIG), the rebuild value of a software system is used to estimate the value of a software system. This estimation is typically not used to actually rewrite a system from scratch, but the rebuild value never-
theless serves as an important indicator of the worth of a software system expressed in monetary terms for management.

The method that we introduce is based on the idea that the ratio of the size (in bytes) of compressed and uncompressed code for a certain language (in this chapter called the **compression ratio** of language $L$ and denoted as $CR_L$) gives information on the verbosity or expressiveness of a programming language. The compressed code can be seen as an estimate of the true information content of code because all duplication, structural or otherwise, is removed from the code. The more verbose a programming language is, the higher its compression ratio will be because it takes more code to implement the same piece of functionality.

We demonstrate that the compression ratio of a programming language correlates to a large degree with estimates for the number of source statements per function point $(SS/FP)$ of other software cost estimation methods. A **function point** is a measurement that expresses the amount of business functionality an information system provides to a user.[3]

The number of source statements per function point for a certain language (in this chapter denoted as $(SS/FP)_L$) is a metric that gives similar information as the compression ratio of that language $(CR_L)$, namely an indication of the average expressiveness (lower $CR_L$ and $(SS/FP)_L$) or verbosity (higher $CR_L$ and $(SS/FP)_L$) of a programming language.

A very low compression ratio is not automatically assumed to be a desirable property of a programming language, since code that does too much with a small amount of code may be hard to understand and maintain. On the other hand, we assume that code that is too verbose is harder to read and maintain as well. This implies that there exists an optimal compression ratio that implements...

To obtain compression ratios for 20 different programming languages, data of over 30,000 snapshots of more than 600 industrial software systems available at the SIG was used. A large number of systems in this dataset are currently in active development or maintenance. We also use the differences between snapshots to provide estimates for the amount of code a developer can create in a certain amount of time.

This chapter is structured as follows. First, we describe formulas and the general approach to perform function point-based productivity and sizing currently used in practice in Section 7.2. We then introduce compression as an alternative to these methods in Section 7.3. After this, we describe our data collection process in more detail in Section 7.4. Compression ratios and churn figures are then presented in Section 7.5. We discuss implications of our findings in Section 7.6, related work in Section 7.7 and we conclude the chapter with Section 7.8.

### 7.2. Function Point-Based Methods

There are several alternative methods available to perform calculations on software size, productivity and costs. A few examples are the QSM benchmark[1], the ISBSG dataset[2],

---

IFPUG\(^3\), and the Software Productivity Research table (SPR) from Capers Jones\(^4\). In the daily consultancy practice of the SIG, the SPR table is used to provide clients with estimates on the rebuild value of their software. For instance, it can be advised that a certain system is too expensive to maintain in the long run and can therefore better be rebuilt from scratch. Although the mentioned software cost estimation methods typically use different estimates for cost, productivity and size, at SIG a general framework of formulas is used that is assumed to underlie all of these methods. In this chapter, we present these formulas and we also present the formulas that underly our alternative method using compression.

An example of the kind of problem a consultant of SIG can work on is estimating the cost of rebuilding a software system in another programming language. Figure 7.3 shows the steps that consultants at SIG typically takes in such a scenario.

![Figure 7.3: Schematic overview of the software cost estimation approach as followed at the Software Improvement Group.](image)

An old system may have already been built in a programming language with a certain productivity estimate. This old system has a certain amount of source statements (old size). By calculating the amount of functionality present in this system, either by backfiring or using compression ratios, as we will see later, we can obtain a language-independent estimate of the functional size of a software system. When rebuilding an old system in a new technology, the estimated system size of the new system can be calculated from the compression ratio or the number of source statements per function point of the new language. Since the size of a software system gives no information about the amount of time it takes to build a system of that size, a benchmark with churn figures (development speed) is needed. With this benchmark, the historical rebuild value of a software system and the estimated rebuild effort of a new system can be calculated.

We define the historical rebuild value of a software system as the amount of man-years it would take to rebuild a system of the same size in the same language:

\[
MY_L = \frac{SS_L}{\left(\frac{SS}{MY}\right)_L}
\]

with

\(^3\)http://www.ifpug.org
\(^4\)http://www.namcookanalytics.com
7.2. Function Point-Based Methods

\[
\left( \frac{SS}{MY} \right)_L = \left( \frac{FP}{MY} \right)_L \times \left( \frac{SS}{FP} \right)_L
\]  

(7.2)

where \( MY \) is the number of man-years, \( L \) is the programming language, \( SS \) is the number of source statements and \( FP \) is the number of function points. \( (SS/MY)_L \) is a constant that can be looked up, for instance, in the SPR table. Similarly, \( (FP/MY)_L \) and \( (SS/FP)_L \) are also constants that can be looked up. Alternatively, company or team-specific figures can be used.

The historical rebuild value of software is thus calculated as the number of lines of code divided by the number of lines of code a programmer writes per year (1). The productivity of a programmer is defined as the number of function points per man-year multiplied by the number of source statements per function point (2). Productivity is thus defined as “intrinsic developer productivity” multiplied by “language verbosity”, which makes intuitive sense: a developer that is able to program more function points per man-year is more productive, but since formula (2) is denoted in source statements per man-year, a developer will need to write more source statements in a verbose language with a high number of source statements per function point to create the same amount of functionality.

In this definition, the “learning effect” is not taken into account, i.e. that the system will be developed faster the second time and will end up being smaller because developers have gained experience while building the system the first time.

Backfired function points [60] are defined as the implied amount of function points present in a system of a certain size, given the function point density of the corresponding programming language:

\[
FP_{BF} = \frac{SS_L}{\left( \frac{SS}{FP} \right)_L}
\]  

(7.3)

Backfiring function points is a way to estimate the amount of functionality present in a system without resorting to manually counting function points. In the consultancy practice of SIG, the number of backfired function points is used to discuss the estimated amount of functionality present in a software system.

Rebuild effort in a new technology is defined as the amount of man-years it would take to rebuild the same amount of functionality in another programming language:

\[
MY_L = FP_{BF} \times \left( \frac{SS}{FP} \right)_L \times \left( \frac{MY}{SS} \right)_L
\]  

(7.4)

The rebuild cost in monetary terms is simply the amount of man-years multiplied by the cost of a programmer per man-year:

\[
C_S = MY_L \times \frac{C_S}{MY}
\]  

(7.5)

where \( C_S/MY \) is a constant denoting the cost per developer per man-year, for instance, $100,000.

There exists a difference between \( SS \) and LOC, where \( SS \) is the number of source statements, excluding non-code lines, such as brackets, whitespace, and comments, and
LOC is the total number of lines in a software system. In practice, there is no universally agreed upon method to calculate SS from LOC; a simple estimate for SS based on a fraction of LOC could be used, such as 0.9. For the sake of simplicity, we do not convert between SS and LOC but assume they are identical.

7.2.1. Example
Consider an example of a system of 200,000 SS originally written in COBOL. The owner of the system wants to rebuild it using a modern object-oriented technology, such as Java. For COBOL, according to the SPR table, the number of function points a COBOL programmer can write in a year (FP/MY) is 113.76, and the number of source statements per function point (SS/FP) is 91.4. The number of source statements a COBOL programmer is expected to write in a man-year (SS/MY) is thus 113.76 \times 91.4 = 10,397.66.

The historical rebuild value, or the amount of time it would take to rebuild the same amount of code in the same language, is, according to the SPR table, 200,000/10,397.66 = 19.24 MY. This would mean that a single developer is expected to build a system written in COBOL of the same size in 19.24 years. If an average COBOL developer costs $100,000 per year, the rebuild value in monetary terms is $1,924,000. When finished, the system would be expected to contain 200,000/91.40 = 2188.18 function points. Since Java has an (SS/FP) of 45.5, when rebuilding the system in Java, the system would contain the same number of function points but would consist of 2188.18 \times 45.50 = 99,562.40 SS. Since FP/MY is 192.8 for Java, this would take 99,562.40/(192.8 \times 45.5) = 11.35 MY. Assuming that a single developer costs the same, the system would take 11.35 \times $100,000 = $1,135,000 to rebuild. Thus, rewriting a system in another language that is considered to be of a higher level (Java) reduces the amount of time and money it takes to implement the same amount of functionality.

7.3. Compression-Based Method
In the method introduced in this chapter, we use compression to calculate the true information content of code, and calculate how much code is needed to implement the same amount of functionality in another programming language. Because the same amount of functionality can be encoded in more or less characters in different languages (conceptually similar to SS/FP from the function point-based methods), and there exist differences in verbosity levels of specific programmers, compression can be used to find the “essential” information present in a software system and make software systems created by different programmers comparable. The concept of Kolmogorov complexity is central to our method, and is explained next.

7.3.1. Kolmogorov Complexity
The Kolmogorov complexity [67] of a piece of information can be described as the smallest size that piece of information can be compressed to. For instance, consider the strings

abababababababababababababababababab and
wboijetoiuatployqwlgibvijefiocplnqde.

While both strings have the same number of characters (36), the first string can be
7.3. COMPRESSION-BASED METHOD

much shorter described as “18 times ab”, while the second string of characters cannot be described any shorter, since it is a random sequence of characters.

The Kolmogorov complexity of a string of characters is theoretically incomputable, but can be approached from above with zip algorithms such as bzip2 [75]. To obtain the shortest string representation of a software system, we first concatenate all source code written in a certain language in a software system in a string, which is then compressed using bzip2.

Conceptually, zip algorithms are capable of removing any duplication or redundant information in code and can reduce a system to its true information content. For instance, when a software system contains a method that is duplicated across different places, the zip file will only store one complete reference to the method and point to that reference in other places in the zip file (dictionary compression). The algorithm is also capable of detecting common structure in different places in the source code, such as a complex for-loop that is duplicated in different places, although identifiers may be different.

7.3.2. Formulas

The formulas for the compression-based model look as follows.

Historical rebuild value for system $S$ and language $L$ and rebuild effort in language $L$:

$$MY_{L,S} = SS_S \frac{SS}{churn}$$  \hspace{1cm} (7.6)

Estimated rebuild size:

$$SS_{NL} = SS_{OL} \frac{CR_{OL}}{CR_{NL}}$$  \hspace{1cm} (7.7)

where $OL$ is the old programming language and $NL$ is the new programming language.

Rebuild cost:

$$C_S = MY_{NL} \frac{C_S}{MY}$$  \hspace{1cm} (7.8)

The compression-based model looks similar to the function point-based methods but contain a number of significant changes. To calculate rebuild effort in man-years (formula 7.6), we use $(churn/MY)_L$, which is similar to $SS/MY$ from the function point-based methods but we define churn to be the amount of new lines plus the amount of changed lines a developer writes between snapshots. We chose this definition because when developing a new system, developers typically not only write new code but also continuously refine previously written code. This number is obtained from our benchmark and the amount of changed lines was obtained through a heuristic which detects lines that are removed first and added later as being changed. Alternatively, it can be detected directly by a source code diff.

The estimated rebuild size in a different language is calculated by “deflating” the original SS by the compression ratio of the old language and then multiplying the deflated number by the compression ratio of the new language (formula 7.7). This amount of SS
can then be converted to man-years by dividing it by the observed benchmark churn for
the new language (again formula 7.6). The calculation to convert man-years to cost is
the same as in the function-point based methods (formula 7.9).

7.3.3. Estimating Team Size

To estimate the historical rebuild value and the rebuild effort in our new methodology
(formula 7.6), the churn per developer per man-year is needed per programming lan-
guage. In source code available at SIG, churn figures are only available for the entire
team, since a snapshot of the entire system is uploaded and the number of developers
that have worked on a system is not provided with the upload. For this reason, we ex-
plain how we have estimated team sizes for each company in our dataset, which are then
used to obtain productivity estimates per programmer per hour.

In practice, there often exists uncertainty over the exact number of developers work-
ning on a software system at any point in time. Especially historically, the number of
developers working on a software system at any moment in time is hard to obtain. This
has several reasons:

- There is no universally agreed upon set of activities that should be counted when
  measuring software development productivity;
- It is often unclear where the boundaries are between developers and people that
  perform activities that are more distant to actually programming the system, such
  as architects, designers, or back-office employees, and which activities should be
  included in the productivity calculation.
- It is often unclear what amount of time developers are actually working in any
  language on a certain system, since their day-to-day activities likely also include
  activities such as writing documentation or communicating with other develop-
  ers;
- Even project managers themselves often cannot tell or do not remember what the
  exact amount of developers that work on a certain system is, especially when the
  system was not developed recently. It is also possible that certain parts of the sys-
  tem are outsourced to another company, in which case productivity information
  is often lost.

For these reasons, we use a lower bound, most likely value and upper bound for the
amount of people working on a software project as input for our estimation method,
instead of using single point estimates. To achieve this, we have asked multiple consul-
tants working at SIG to provide their minimum estimate, their most likely estimate and
their maximum estimate for the number of developers working on software projects,
taking into account the uncertainties mentioned above. Consultants were told that the
most likely estimate is their “best guess”, the lower bound is the number of which they
are fairly certain the number of developers does not go below, and the upper bound the
number of which they are fairly certain the number does not go above. We told con-
sultants that the estimates apply to the latest two years of development. We did not tell
consultants what percentage the implied confidence interval spans, since try-outs have
shown that this would not have helped people estimating these numbers better.
We applied a Monte Carlo estimation technique, in which these three estimates define a triangular distribution for the number of developers working on a project. This way, an estimate can be obtained with a preferred uncertainty range, for instance, with an 80% confidence interval. For the sake of simplicity, we only show the lower, best and upper bound for our churn figures.

The formula to calculate churn per developer per hour is as follows:

\[
\frac{SS}{FTE/hr} = \frac{churn}{NP \times HPD \times ND}
\]  

where \(churn\) is the churn between two snapshots, \(NP\) is the number of programmers that are working on the project, \(HPD\) is the number of hours per day that developers work on the system, and \(ND\) is the number of days between the dates of the two snapshots from which the churn was obtained. We assume eight working hours per day, and the number of days between snapshots was obtained from the dates of the snapshots, excluding weekends. A lower bound, best guess and upper bound could also be used for the number of hours per day or the number of days between snapshots, although typically, these numbers are known with more certainty.

### 7.3.4. Example

Figure 7.4 shows an overview of our approach using compression ratios and benchmark churn.

Figure 7.4: An example of our methodology.

In our approach, we assume that an existing system of a certain size has been built (denoted with ①). If the size of an existing system is not available, an estimate based on
previous projects or an implied size calculated from a given time, budget or number of developers can be used.

Using a benchmark with churn data for the same language, an estimate for the historical rebuild value can be obtained (denoted with ②). As described before, a histogram showing an uncertainty range will be obtained, but we will use a minimum, most likely and maximum churn in this example, denoted with (min, best, max). Next, the compression ratio for COBOL (15.35) is used to convert the 200,000 SS into its true information content: 13,029.32 (denoted with ③). This quantity is dimensionless and represents the amount of information present in the system. Next, to calculate the amount of expected Java code, the compression ratio of Java is used (9.03), which leads to an expected system size of 117,654 SS of Java code.

Returning to the previous example, in the new model, the following calculations would be performed. Assuming the same system size of 200,000 SS in COBOL, and a minimum, most likely and maximum churn of (4, 6, 8) SS per developer per hour, the system would take 200,000/(4, 6, 8)/1,664 = (15.2, 20.2, 30.3) man-years to rebuild in COBOL. Assuming a compression ratio of 15.35 for COBOL and 9.03 for Java, the expected size in Java would be (200,000/15.35) * 9.03 = 117,654.70 SS.

Based on this number, the amount of time it takes to build a system of that size is calculated from the benchmark churn for Java (denoted with ④). Assuming an churn of (6.55, 7.20, 10.2) SS/FTE/hr for Java, it would take 117,655/(6.55,7.20,10.2)/1,664 = (7.0, 9.9, 10.9) MY to rebuild in Java. The calculation of rebuild cost is the same as in the function point-based methods.

### 7.4. Data Collection

The mean $SS/FP$ in Table 7.5 was obtained from the four software productivity benchmarks mentioned in Section 7.2. The ISBSG dataset does not explicitly provide $SS/FP$ but it was calculated as the total system size divided by the estimated number of implemented function points, after which averages for different languages were obtained.

Data for the calculated compression ratios was obtained from the datawarehouse of the SIG, containing 10,453 snapshot of 80 different systems, with dates ranging from January, 2008 to February, 2014. SIG delivers consultancy on software quality to their customers, and a software quality rating can be calculated automatically on source code of a software system. Clients of the SIG upload their source code periodically to receive a new quality rating. Most clients upload snapshots of their systems once a week, but the interval can vary among clients.

Compression ratios for each language were calculated by collecting all files in a system written in a certain language. For each system snapshot, consultants of the SIG have determined which files in the system belong to which programming language. This information is stored in a configuration file, and these files were used to select the proper files for each language. From these files, the total size of the source code was calculated by adding the size of all individual files in bytes. All files belonging to the same language

---

5 $52 \times 5 = 260$ working days per year, 4 sick days, 25 free days, 5 study days, 5 holidays, 13 days of other absence = 52 days of total absence, leading to $260 - 52 = 208$ working days per year. 8 hours per day give $208 \times 8 = 1,664$ hours per year. These numbers depend on country, company and culture, therefore, adjustments may be needed in different contexts.
in the same system were added to a zip file using the bzip2 compression algorithm. The file size of this zip file in bytes was stored. We use the median compression ratio because it is less sensitive to outliers than the mean.

The number of developers working on each software project were obtained through interviews with consultants that were associated with each client. In total, developer numbers were obtained for a set of 90 systems in total, in 20 different languages. Since most software systems are written in multiple languages and it was impossible to determine which amount of time developers worked on a particular language, only software systems with more than 90% of source code in the same language were selected. For these systems, it was assumed that 100% of source code was written in this language.

A moving average with a window with a size of 10 was used to smooth churn figures, since an analysis showed that the churn can vary wildly between snapshots. The moving average resulted in a more robust and steady churn figure over multiple snapshots.

In the next section, we calculate the correlation between the compression ratio and the other estimation models to validate our approach. After this, we show the table of compression ratios that was obtained from our benchmark. Finally, we show the churn numbers as they are estimated from our benchmark using the approach described above.

7.5. Results

7.5.1. Compression ratios

The median compression ratios as obtained from our benchmark are shown in Table 7.5. These compression ratios are compared with the average SS/FP from Table 7.9. When applying a Spearman rank correlation, the result is 0.63 and is highly significant ($p < 0.001$).

The table further shows that languages that are typically considered to be “low-level”, such as C and COBOL, have a high compression ratio: 16.16 and 15.35 respectively. Java and C# are relatively close together with compression ratios of 9.03 and 9.77, which is expected due to the similarities between these languages. Python, a language that is considered to be high-level, has a compression ratio of 4.88. These results match our intuition that Python is a higher-level language than C or COBOL, which require a larger number of statements to implement the same amount of functionality, yielding a higher compression ratio.

Table 7.1 shows conversion ratios between different languages ($CR_{OL}/CR_{NL}$). From this table, the expected amount of SS can be directly calculated when converting a software system to a new language. For instance, when the old system is written in C and consists of 100,000 SS and the new system will be written in Java, a conversion ratio of 1.8 is applied: the new system is estimated to contain $100,000/1.8 = 55,556$ SS.

A second validation of our approach is performed by comparing conversion ratios from Table 7.1 against absolute differences in SS/FP for each language pair from Table 7.5. A scatterplot of this data is shown in Figure 7.6. With a correlation coefficient of 0.56 ($p < 0.001$), further support is provided that differences in language verbosity or expressiveness as measured by the number of source statements per function point correspond to differences in compression rates.
Table 7.1: Conversion ratios \((CR_{OL}/CR_{NL})\) as calculated from compression ratios.
7.5. Results

<table>
<thead>
<tr>
<th>Language</th>
<th>Median CR</th>
<th>Mean SS/FP</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABAP</td>
<td>7.61</td>
<td>12.7</td>
<td>141</td>
</tr>
<tr>
<td>C</td>
<td>16.16</td>
<td>117.2</td>
<td>25</td>
</tr>
<tr>
<td>COBOL</td>
<td>15.35</td>
<td>87.3</td>
<td>496</td>
</tr>
<tr>
<td>C++</td>
<td>7.58</td>
<td>50.1</td>
<td>358</td>
</tr>
<tr>
<td>C#</td>
<td>9.77</td>
<td>30.3</td>
<td>1,907</td>
</tr>
<tr>
<td>Delphi</td>
<td>6.75</td>
<td>66.1</td>
<td>41</td>
</tr>
<tr>
<td>Java</td>
<td>9.03</td>
<td>41.8</td>
<td>4,243</td>
</tr>
<tr>
<td>Javascript</td>
<td>5.25</td>
<td>40.6</td>
<td>1,063</td>
</tr>
<tr>
<td>JSP</td>
<td>10.63</td>
<td>13.9</td>
<td>606</td>
</tr>
<tr>
<td>Objective C</td>
<td>6.99</td>
<td>42.7</td>
<td>52</td>
</tr>
<tr>
<td>Perl</td>
<td>7.01</td>
<td>31.8</td>
<td>3</td>
</tr>
<tr>
<td>PHP</td>
<td>5.67</td>
<td>12.8</td>
<td>45</td>
</tr>
<tr>
<td>PL/SQL</td>
<td>9.90</td>
<td>20.1</td>
<td>563</td>
</tr>
<tr>
<td>Python</td>
<td>4.88</td>
<td>21.3</td>
<td>82</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>11.17</td>
<td>35.8</td>
<td>8</td>
</tr>
<tr>
<td>Tandem</td>
<td>13.01</td>
<td>91.4</td>
<td>352</td>
</tr>
<tr>
<td>T-SQL</td>
<td>13.18</td>
<td>27.2</td>
<td>468</td>
</tr>
<tr>
<td>correlation:</td>
<td>0.63 (p &lt; 0.001)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.5: Compression ratios and the correlation with the average SS/FP for 17 different languages.

7.5.2. Benchmark Churn

In Table 7.7, we show churn figures per developer per hour for a set of languages. Because of the lower bound, best guess and upper bound present in the estimation for the number of developers, this table also shows a lower bound, best guess and upper bound for the developer productivity per hour.

As can be seen in the table, the average lower bound for developer productivity per hour is 3.24, the best guess is 4.09 and the upper bound is 7.22. When assuming a productivity of 4.09 SS per developer per hour, a developer is expected to write 33 SS per day, 164 SS per week, 709 SS per month and 8,507 SS per year. This number is averaged over all languages in the table.

The table further shows that certain languages have a very low productivity number, such as Ruby (0.14 SS/hour) or PL/SQL (1.10 SS/hour). The number of unique systems for Ruby is limited, and therefore, care should be taken to use the productivity numbers for Ruby directly. Furthermore, the numbers as obtained from our benchmark typically represent systems that are being maintained. This results in different estimates than a benchmark of systems that are being built from scratch and experience a period of initial fast growth.

In the next section, we describe five example scenarios that demonstrate the calculations of function point-based methods and our new methodology.

7.5.3. Example Scenarios

Table 7.8 shows five scenarios comparing function point-based methods with our new method. The table assumes software developer costs of $100,000 per year and 1,664 productive (billable) hours per year.
Scenario 1 shows the example discussed before, rebuilding 200,000 SS of COBOL into Java. The historical rebuild value for the new system, based on compression ratios, is $1.08 million, as compared to an estimate of $1.14 million with function point-based methods.

The second scenario shows a system in COBOL that will be rewritten in C#. In our dataset, the observed median churn for C# projects is considerably lower than the average: the median is 2.95. This results in a far higher estimate for this scenario using compression ratios: $1.18 million. This churn figure is obtained from a benchmark of 25 different systems written in C#, but the question is how representative this number is, which becomes visible when comparing the number with the estimate for Java, which is 7.20. When using the Java estimate for this scenario, the amount becomes $531,000.

Scenario 3 shows a conversion from Java to C#. Due to the relatively low productivity of C# systems in our benchmark, our estimate is almost twice as much as the estimate of the function-point based methods. Scenario 4 shows a system written in C that will be converted to Java. The conversion ratio from C to Java is 1.7, which is very close to the conversion ratio of COBOL to Java. This leads to an estimate of $1,029 million instead of an estimate of $745,000 using function points.

Scenario 5 shows a small system of 10,000 SS written in C that will be rewritten in Python. Again, the estimate of our method is higher with $30,000 instead of $25,000 using the old method. Looking at these examples, we see that due to differences in benchmark churn and the calculation method used, function point-based methods and our method result in estimates that can diverge but can nevertheless provide an estimate
that is reasonably close to the function point-based estimate.

### 7.6. Discussion

To the best of our knowledge, compression and Kolmogorov complexity has not been applied before as a way to compare productivity and expressiveness of programming languages on such a large scale. We have shown that it is possible to obtain valid estimates that correlate well with software productivity numbers from other sources. However, since there are no universally agreed upon standards of function point density or size of compressed source code, it is impossible to obtain a reliable benchmark with which to compare our method.

The differences in function point densities between different software productivity and cost estimation methods become apparent when comparing them. This is done in Table 7.9, which shows that there exist large differences between multiple benchmarks for the same programming language.

For instance, according to ISBSG, the language C has an implied SS/FP of 18.6, but according to IFPUG, the number is 225. This means that IFPUG estimates that the number of lines of code that are needed to implement a single function point is 12 times as high as the ISBSG estimate. Such large differences become visible for other languages as well, such as the estimate of Delphi with 29.1 SS/FP for SPR and 160 SS/FP according to IFPUG. This is more than five times as much. Practically, this would mean that the estimated cost to develop the same amount of functionality can differ as much as 12 times, depending on the benchmark used.

<table>
<thead>
<tr>
<th>Language</th>
<th>lower</th>
<th>best</th>
<th>upper</th>
<th>#ss</th>
<th>#s</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABAP</td>
<td>2.23</td>
<td>2.98</td>
<td>3.57</td>
<td>744</td>
<td>9</td>
</tr>
<tr>
<td>ASP .NET</td>
<td>1.60</td>
<td>3.16</td>
<td>3.35</td>
<td>54</td>
<td>7</td>
</tr>
<tr>
<td>BPEL</td>
<td>1.37</td>
<td>1.60</td>
<td>1.80</td>
<td>57</td>
<td>1</td>
</tr>
<tr>
<td>C++</td>
<td>7.52</td>
<td>8.88</td>
<td>18.2</td>
<td>488</td>
<td>6</td>
</tr>
<tr>
<td>C#</td>
<td>2.41</td>
<td>2.95</td>
<td>3.62</td>
<td>441</td>
<td>25</td>
</tr>
<tr>
<td>Java</td>
<td>6.55</td>
<td>7.20</td>
<td>10.3</td>
<td>812</td>
<td>17</td>
</tr>
<tr>
<td>Javascript</td>
<td>2.40</td>
<td>4.19</td>
<td>4.19</td>
<td>19</td>
<td>5</td>
</tr>
<tr>
<td>Lodestar</td>
<td>1.48</td>
<td>1.66</td>
<td>1.90</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>PL/SQL</td>
<td>0.95</td>
<td>1.10</td>
<td>1.50</td>
<td>83</td>
<td>4</td>
</tr>
<tr>
<td>PowerBuilder</td>
<td>7.25</td>
<td>10.9</td>
<td>21.7</td>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>Ruby</td>
<td>0.10</td>
<td>0.14</td>
<td>0.21</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>T-SQL</td>
<td>1.49</td>
<td>1.49</td>
<td>17.3</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>TIBCO</td>
<td>9.37</td>
<td>11.0</td>
<td>12.5</td>
<td>45</td>
<td>1</td>
</tr>
<tr>
<td>WSDL</td>
<td>1.76</td>
<td>1.76</td>
<td>3.52</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>XSD</td>
<td>2.18</td>
<td>2.39</td>
<td>4.68</td>
<td>66</td>
<td>7</td>
</tr>
<tr>
<td><strong>avg</strong></td>
<td>3.24</td>
<td>4.09</td>
<td>7.22</td>
<td>195</td>
<td>5.9</td>
</tr>
</tbody>
</table>

Table 7.7: Churn values as obtained from our benchmark. #ss = number of snapshots, #s = number of unique systems. lower = minimum, best = most likely/best guess, upper = maximum estimate.
### Table 7.8: Comparison of 5 rebuild scenarios using function point-based models and the model as introduced, assuming a cost of $100,000/FTE and 1,664 productive hours per year.

<table>
<thead>
<tr>
<th></th>
<th>Sc1</th>
<th>Sc2</th>
<th>Sc3</th>
<th>Sc4</th>
<th>Sc5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function point-based methods</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OL</strong></td>
<td>COBOL</td>
<td>COBOL</td>
<td>Java</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>SS</td>
<td>200,000</td>
<td>100,000</td>
<td>100,000</td>
<td>200,000</td>
<td>10,000</td>
</tr>
<tr>
<td>SS/FP</td>
<td>91.4</td>
<td>91.4</td>
<td>45.50</td>
<td>139,20</td>
<td>139,20</td>
</tr>
<tr>
<td>FP/MY</td>
<td>113.76</td>
<td>113.76</td>
<td>192.84</td>
<td>91.20</td>
<td>91.20</td>
</tr>
<tr>
<td>SS/MY</td>
<td>10,398</td>
<td>10,398</td>
<td>8774</td>
<td>12,695</td>
<td>12,695</td>
</tr>
<tr>
<td>MYOL</td>
<td>19.24</td>
<td>9.62</td>
<td>11.40</td>
<td>15.75</td>
<td>0.79</td>
</tr>
<tr>
<td><strong>FP</strong></td>
<td>2,188.2</td>
<td>1,094.1</td>
<td>2,197.8</td>
<td>1,436.8</td>
<td>71.8</td>
</tr>
<tr>
<td>$ * 1,000</td>
<td>$1,924</td>
<td>$962</td>
<td>$1,140</td>
<td>$1,575</td>
<td>$78.7</td>
</tr>
<tr>
<td><strong>NL</strong></td>
<td>Java</td>
<td>C#</td>
<td>C#</td>
<td>Java</td>
<td>Python</td>
</tr>
<tr>
<td>SS/FP</td>
<td>45.5</td>
<td>34.7</td>
<td>34.70</td>
<td>45.50</td>
<td>22.10</td>
</tr>
<tr>
<td>FP/MY</td>
<td>192.84</td>
<td>230.2</td>
<td>230.2</td>
<td>192.84</td>
<td>282.8</td>
</tr>
<tr>
<td>SS/MY</td>
<td>8,774</td>
<td>7,988.6</td>
<td>7,988.6</td>
<td>8,774</td>
<td>1,588</td>
</tr>
<tr>
<td>MYNL</td>
<td>99,562</td>
<td>37,965</td>
<td>76,264</td>
<td>65,374</td>
<td>1,588</td>
</tr>
<tr>
<td>$ * 1,000</td>
<td>$1,135</td>
<td>$475</td>
<td>$955</td>
<td>$745</td>
<td>$25</td>
</tr>
<tr>
<td><strong>Compression ratio-based method</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CROL</strong></td>
<td>15.35</td>
<td>15.35</td>
<td>9.03</td>
<td>16.16</td>
<td>16.16</td>
</tr>
<tr>
<td><strong>CRNL</strong></td>
<td>9.03</td>
<td>9.77</td>
<td>9.77</td>
<td>9.03</td>
<td>4.88</td>
</tr>
<tr>
<td>c/FTEOL</td>
<td>6.0</td>
<td>6.0</td>
<td>6.53</td>
<td>8.90</td>
<td>8.90</td>
</tr>
<tr>
<td>c/FTENL</td>
<td>6.53</td>
<td>3.22</td>
<td>3.22</td>
<td>6.53</td>
<td>6.0</td>
</tr>
<tr>
<td>MYOL</td>
<td>20.03</td>
<td>10.02</td>
<td>9.2</td>
<td>13.5</td>
<td>0.68</td>
</tr>
<tr>
<td>MYNL</td>
<td>10.83</td>
<td>11.88</td>
<td>20.19</td>
<td>10.29</td>
<td>0.30</td>
</tr>
<tr>
<td>$ * 1,000</td>
<td>$1,083</td>
<td>$1,188</td>
<td>$2,019</td>
<td>$1,029</td>
<td>$30</td>
</tr>
</tbody>
</table>

7.6.1. **Estimation in practice**

There were several problems regarding the estimates of the number of developers for different projects. The process to obtain the number of developers is time-consuming, and people are often not familiar with providing a lower bound, best guess and upper bound of an estimate. It is our experience that people are uncomfortable with estimation in general since they feel they “do not really know”. However, even with little information on the actual amount of developers working on a project, a reasonable estimate can still be made, as demonstrated in this chapter. We have observed a need to explain that the uncertainty of estimation is an integral part of the methodology chosen and is inherent in software development. Even with little data, a wide range between lower and upper estimate would be obtained, which could be gradually improved when more information becomes available or a new data collection effort is initiated.

We suspect that the unfamiliarity of consultants with the usage of a minimum, most likely/best guess and maximum estimate for the number of developers may have led to diverging interpretations of these terms. For instance, a consultant may interpret the “minimum” amount of developers to be the minimum amount of developers that were working on the project, at any time in the history of the project. Another interpretation could be that the minimum is the minimum amount of developers working on the project at the latest moment, i.e. when most of the developers were on holiday. We have
7.6. Discussion

Table 7.9: SS/FP and the average for four different benchmarks. Implicit ISBSG SS/FP calculated by dividing given SS by estimated function points.

<table>
<thead>
<tr>
<th>Language</th>
<th>QSM</th>
<th>ISBSG</th>
<th>IFPUG</th>
<th>SPR</th>
<th>Avg.</th>
<th>s.d.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABAP</td>
<td>6.0</td>
<td>16.0</td>
<td>16.0</td>
<td>12.0</td>
<td>12.7</td>
<td>5.8</td>
</tr>
<tr>
<td>C</td>
<td>97.0</td>
<td>18.6</td>
<td>225.0</td>
<td>128.0</td>
<td>117.2</td>
<td>85.4</td>
</tr>
<tr>
<td>COBOL</td>
<td>61.0</td>
<td>21.9</td>
<td>175.0</td>
<td>91.4</td>
<td>87.3</td>
<td>65.0</td>
</tr>
<tr>
<td>C++</td>
<td>50.0</td>
<td>17.1</td>
<td>80.0</td>
<td>53.3</td>
<td>50.1</td>
<td>25.8</td>
</tr>
<tr>
<td>C#</td>
<td>54.0</td>
<td>6.6</td>
<td>30.3</td>
<td>33.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delphi</td>
<td>9.1</td>
<td>160</td>
<td>29.1</td>
<td>66.1</td>
<td>82.0</td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td>53.0</td>
<td>11.3</td>
<td>80.0</td>
<td>22.9</td>
<td>41.8</td>
<td>30.9</td>
</tr>
<tr>
<td>Javascript</td>
<td>47.0</td>
<td>12.1</td>
<td>50.0</td>
<td>53.3</td>
<td>40.6</td>
<td>19.2</td>
</tr>
<tr>
<td>JSP</td>
<td></td>
<td>13.9</td>
<td>13.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Objective C</td>
<td>42.7</td>
<td></td>
<td></td>
<td>42.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perl</td>
<td>24.0</td>
<td>50.0</td>
<td>21.3</td>
<td>31.8</td>
<td>15.8</td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td></td>
<td></td>
<td>12.8</td>
<td>12.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PL/SQL</td>
<td>37.0</td>
<td>11.5</td>
<td>11.9</td>
<td>20.1</td>
<td>14.6</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td></td>
<td></td>
<td>21.3</td>
<td>21.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smalltalk</td>
<td>25.9</td>
<td></td>
<td>45.7</td>
<td>35.8</td>
<td>14.0</td>
<td></td>
</tr>
<tr>
<td>Tandem</td>
<td></td>
<td></td>
<td>91.4</td>
<td>91.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T-SQL</td>
<td>21.0</td>
<td>15.0</td>
<td>60.0</td>
<td>12.8</td>
<td>27.2</td>
<td>22.1</td>
</tr>
</tbody>
</table>

instructed consultants in such way that the definition was unambiguous: the estimates apply to the latest 2 years of development. Future work could more accurately investigate the actual number of developers and thus reduce the uncertainty present in the estimates in Table 7.7. However, we believe that the estimation method is robust to inaccuracies in estimation, since aggregation is performed over multiple clients and software systems.

7.6.2. ISSUES WITH BENCHMARK APPROACH

As the examples with C# show in Section 7.5.3, a benchmark with churn estimates can have the disadvantage that the numbers obtained from it may not truly reflect actual development practice. Judgment needs to be used when using benchmarked figures. In this case, a more representative number may be chosen, or when the reasons for the deviating productivity number can be found and these reasons also apply to the case at hand, the number can still be used.

The large difference in estimates for C# and Java could be explained by the fact that the number of developers for C#-projects has been overestimated, resulting in a productivity per programmer that is too low. Alternatively, it is possible that our dataset contains a relatively large amount of C# systems that are in “maintenance mode” as compared to Java systems. Another explanation is that developers in C# can use the libraries of the .NET framework, which possibly provide a higher productivity boost than the base class libraries in Java.

The issue of maintenance systems is not only present for systems written in C#, but for all other languages as well. The systems in our dataset are typically large, industrial software systems in a state of maintenance. There are few systems in the benchmark that were built from scratch from the moment the first snapshot was obtained. We ex-
pect that systems written from scratch can be developed faster, and using the churn from these maintenance-mode systems to calculate rebuild value may lead to an overestimation of required effort. We have (anecdotal) evidence that the productivity of newly built systems is twice as high as the productivity of systems that are actively maintained. Future research could further investigate whether this holds when the two type of systems are separated.

7.6.3. **CALCULATION OF COMPRESSION RATIO**

A problem with the usage of compression to approximate Kolmogorov complexity is that it is unclear how a piece of source code is precisely compressed, since the result is a binary file. This makes it impossible to find out what the underlying reasons for a high compression ratio are. The absence of the possibility to perform root-cause analysis also makes it harder to get the method accepted by practitioners. To shed some light on the effectiveness of the zipping algorithm for different software systems, further research is required. For example, an experiment could be set up which calculates the expected compression ratio for completely distinct, duplicated, or slightly modified pieces of code. Such an experiment could also be used to determine the difference in compression ratios for a software system with the same functionality written in different languages.

7.6.4. **VALIDATION**

Our results show that we obtain a correlation of 0.63 for the average SS/FP for 17 languages, and a correlation of 0.56 when correlating conversion ratios with differences in SS/FP. The number of languages that are included in this analysis is relatively low, and ideally, a larger number of data points would need to be obtained. However, the number of different languages that the four tables have in common is limited to this number. Given the large spread in estimates for SS/FP for the four different benchmarks, a high correlation is also not necessarily desirable and should only serve as a general validation of our approach.

7.6.5. **REAL-WORLD CASE STUDIES**

We have not yet gained experience with the extensive application of our model to practice, but we have indications that in several cases, our method returns more accurate estimates than function point-based methods. In future work, we plan to describe the lessons learned from applying our model to real-world client projects on a larger scale.

7.7. **RELATED WORK**

Besides the four benchmarks described before, there also exist several alternative software cost estimation methods that are used in practice. An example is the COCOMO method by Barry Boehm [19]. These methods all use various calculations based on function points and productivity estimates similar to the churn figures as used in this chapter.

There exists several other software cost estimation methods based on various approaches. A frequently used methodology is regression [23, 64], in which several explaining factors are used to predict a linear trend in an outcome variable (usually effort).
Another method is the usage of neural networks \cite{100}, in which a predictive model for effort is built using several explanatory variables, which can have mutual relationships that are more complex than that are often found in regression analyses.

More informal methods are also available, such as expert judgment \cite{61, 82}. These methods aim to catch expert’s opinion using more or less structured approaches. Closely related are methods based on work breakdown \cite{58}, which divide the system to be built into parts which are then estimated separately. For an overview of software cost estimation methods, see \cite{62}.

Compression has been applied in various domains to find the similarity between various sources of data. For instance, it has been applied to cluster music by genre \cite{27, 112}, to build an evolutionary tree of different mammals \cite{24, 27} or to classify natural languages \cite{74}.

### 7.8. Conclusion

In this chapter, we have compared language productivity using compression and we have introduced a set of formulas to calculate rebuild size and to perform cost estimation using compression. The data we have obtained for a set of languages shows that there exists a significant correlation between the number of source statements per function point and the degree to which code can be compressed. This demonstrates that compression can be used as a software cost estimation method. We have also calculated average churn figures for a set of programming languages from a large industrial benchmark. Future work could validate our approach with real-world use cases or by comparing cost estimates with outcomes of our method.
In previous chapters, we have determined that interfaces and implementations of software libraries keep changing. We have also determined that this causes measurable impact in libraries using these changed interfaces. In this chapter, we investigate the percentual and absolute growth of open-source and industrial software systems. This gives an overview of how fast software grows in general, before we continue to investigate desirable and undesirable growth in the next chapter. We employ a linear regression technique to model multiple software growth curves of different systems over time to obtain a daily absolute growth rate for industrial software systems. We calculate differences in size between snapshots from the Maven Dependency Dataset to obtain a yearly percentual growth rate for open-source Java libraries.

8.1. INTRODUCTION

It is generally assumed that software systems tend to grow over time in terms of lines of code and amount of functionality. A lot of research has been performed to estimate growth functions, which express the size of a software system as a function of time [48, 49, 55, 66, 96]. The goal of this research is often to investigate whether current growth figures can be maintained in the long run, with the underlying idea that the bigger software gets, the harder it must be to maintain the same percentual growth rate. This effect has long been recognized by Lehman's laws of software evolution [69–71]. Lehman's law of continuing growth states that “the functional content of an E-type system must be continually increased to maintain user satisfaction over time”. This is often perceived as stating that the size of a software system must increase in order to stay relevant [56].

The goal of this chapter is to obtain a growth function for open-source and industrial software systems, that can serve as a basis for a further distinction of software growth in a desirable and undesirable component in the next chapter. These growth functions provide insight in the actual growth speed of real-world industrial and open-source software systems. As we will find out in this chapter, the growth speed of industrial software systems depends on the size of that software system.
We start this chapter with a description of the set of industrial software systems available at the SIG that was used in our analysis. The Maven Dependency Dataset is also used, which was described in Chapter 2. We present the absolute growth of software systems in terms of lines of code per day by using a fixed-effects linear regression model that follows the “growth curve” of software systems over time and estimates a slope coefficient for this curve. Next, we calculate the percentual yearly growth of open-source software systems by extrapolating the growth between library versions. We conclude with a discussion of the implications of our findings for practitioners.

8.2. Dataset

To investigate daily software growth, we use an industrial dataset of 22,748 snapshots of 340 unique software systems of which source code is available at the SIG. These systems are written in 77 different programming languages with 66.7 snapshots per system on average. The dates of the snapshots range from January, 2001 to February, 2014. Snapshots are typically uploaded to SIG once a week, although the exact interval can vary. We use the total number of lines of code (LOC) per snapshot to calculate daily absolute growth.

Data was obtained from the Software Analysis Warehouse (SAW) of the SIG. Clients of SIG upload their source code periodically, after which the Software Analysis Toolkit (SAT) of the SIG processes this source code and extracts relevant metrics. These metrics are then imported into the SAW. In our analysis, we only use the name of the system, the date of the system and the size of the system in LOC for every moment in time for each system. Data was exported from the SAW and imported into Stata 12\(^1\) to be analyzed.

We calculate absolute daily growth numbers on the industrial dataset and yearly percentual growth numbers on the Maven Dependency Dataset because the number of days between releases in the industrial dataset is lower (\(\mu = 12.6, p_{50} = 7, \sigma = 52.8\)) than in the Maven Dependency Dataset (\(\mu = 71.1, p_{50} = 37, \sigma = 106.6\)). This means that the growth rates in the Maven Dependency Dataset can be more naturally expressed using yearly percentual growth rates.

8.3. Software Growth

8.3.1. Absolute Growth per Day

To calculate the absolute growth per day, we apply a panel data regression technique [21, 114] to our dataset. Panel data are multiple measurements through time of multiple subjects, in this case software systems.

Figure 8.1 shows an example of the kind of data that is analyzed using this technique. The figure shows the size of a selection of systems in LOC from 2010 to the beginning of 2014. A wide variety in growth patterns can be observed.

The goal of this analysis is to obtain growth coefficients that indicate the growth in LOC per day for an average software system. The regression model provides an “average” slope coefficient of the regression line that fits best on all growth lines such as shown in Figure 8.1. Additionally, to find out if systems with different sizes grow at different rates,

\(^1\)http://www.stata.com/
8.3. **SOFTWARE GROWTH**

we split up the data in different size categories: smaller than 10 KLOC, between 10 and 50 KLOC, between 50 and 100 KLOC, between 100 and 250 KLOC, between 250 and 500 KLOC, and larger than 500 KLOC.

The *date* variable is an integer which increases for each day; its estimated coefficient thus represents the average daily growth. The panel dataset is *unbalanced*, meaning that not all days have snapshots, the interval between snapshots is not guaranteed to be equal, neither within the same system nor between systems, and systems do not have the same number of snapshots. This, however, does not pose a problem for the chosen regression method.

Systems with less than five snapshots were excluded from the analysis. Additionally, “flatliners” were also excluded, i.e. systems of which a large number of snapshots without substantial differences over time in terms of LOC. A flatliner was defined as a system with less than 10% maximum absolute deviation from the average LOC over all snapshots of that system. Outliers were detected manually, and based on this inspection, three systems with extremely irregular growth curves (e.g., 1,000-fold increases in LOC between subsequent snapshots) were excluded from the analysis. The reason to exclude these systems is that there are a lot of systems present in the benchmark of SIG on which no active development is being done. These systems would unrealistically reduce growth figures when including them in the analysis.

Further inspection of the growth curves of the remaining systems showed a large degree of volatility in LOC values between snapshots. For this reason, a moving average on the LOC with a window of five snapshots was chosen as dependent variable in the growth model, which helps to identify the underlying long-term trend and smoothes out short-term differences between subsequent snapshots.

Figure 8.1: An example of the growth in LOC for a set of industrial software systems.
A fixed-effects generalized least squares model was chosen to model daily software growth. This model calculates the average growth coefficient per day, while taking the grouping of snapshots to systems into account. The results of the model on all snapshots combined can be found in Table 8.1. The model is highly significant with a \( p \)-value of 0 and 22,748 data points (snapshots), clustered in 340 systems. The average number of snapshots per system is 66.9. The coefficient of date shows that the average system is expected to grow with 17.05 LOC per day.

<table>
<thead>
<tr>
<th>Model statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td># Snapshots</td>
<td>22,748</td>
</tr>
<tr>
<td># Systems</td>
<td>340</td>
</tr>
<tr>
<td>Snapshots per system</td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>1</td>
</tr>
<tr>
<td>avg</td>
<td>66.9</td>
</tr>
<tr>
<td>max</td>
<td>274</td>
</tr>
<tr>
<td>( F(1, 22,407) )</td>
<td>616.96</td>
</tr>
<tr>
<td>( p )</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.1: Panel data regression combining all system sizes.

Table 8.2 shows the daily growth coefficients per size category. There exist large differences in the growth coefficient for systems of different sizes. Figure 8.2 shows a graph of the coefficients. As can be seen, between 50 and 100 KLOC, systems tend to grow the fastest with 124.17 LOC per day. Systems between 100 and 250 KLOC grow slower than systems between 50 and 100 KLOC with a growth rate of 57.61 LOC per day. Systems bigger than 500 KLOC show a negative growth, with -41.17 LOC per day.

<table>
<thead>
<tr>
<th>KLOC</th>
<th>#ss</th>
<th>#uniq</th>
<th>coeff.</th>
<th>lower</th>
<th>upper</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 10k</td>
<td>14,118</td>
<td>206</td>
<td>5.60</td>
<td>5.04</td>
<td>6.17</td>
<td>0</td>
</tr>
<tr>
<td>10k - 50k</td>
<td>3,452</td>
<td>58</td>
<td>16.22</td>
<td>13.45</td>
<td>18.98</td>
<td>0</td>
</tr>
<tr>
<td>50k - 100k</td>
<td>1,097</td>
<td>23</td>
<td>124.17</td>
<td>114.84</td>
<td>133.51</td>
<td>0</td>
</tr>
<tr>
<td>100k - 250k</td>
<td>2,184</td>
<td>28</td>
<td>57.61</td>
<td>51.87</td>
<td>63.35</td>
<td>0</td>
</tr>
<tr>
<td>250k - 500k</td>
<td>777</td>
<td>10</td>
<td>19.93</td>
<td>13.90</td>
<td>25.95</td>
<td>0</td>
</tr>
<tr>
<td>&gt; 500k</td>
<td>1,120</td>
<td>15</td>
<td>-41.17</td>
<td>-58.33</td>
<td>-24.02</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8.2: Panel data regression results for snapshots of different sizes. #ss = number of snapshots, #uniq = number of unique systems, coeff. = daily growth coefficient, lower/upper = 95% confidence interval.

The data indicates that systems larger than 100 KLOC tend to show “diseconomies of scale”: the amount of new code that gets added every day is lower than in systems smaller than 100 KLOC. Above 500 KLOC, system growth is negative, indicating that these systems tend to become smaller over time. A possible explanation for this is that after a certain size, systems are simply finished and they do not need to be extended with new functionality, but are refactored and redesigned which reduces the total size of the sys-
Another effect that could explain the negative growth rate is that developers working on systems bigger than 500 KLOC face increasing difficulty to add new functionality to the system due to its large size. Combined with a possible refactoring effect in these large systems, the overall result is that these systems decrease in size. The optimal system size in terms of growth capacity is, according to this analysis, a system with a size between 50 and 100 KLOC.

8.3.2. **Percentual Growth Per Year**

In addition to absolute growth rates per day, we calculate yearly percentual growth rates on all software libraries in the Maven repository. Since most software projects in the Maven repository release more than once a year and the interval between any two snapshots almost never spans exactly one year, we select each pair of snapshots that has the smallest interval bigger than 365 days and then obtain a yearly growth percentage by applying the following formula:

\[
g = \left(1 + \frac{LOC_t - LOC_{t-1}}{LOC_{t-1}}\right)^{\frac{365}{\text{days}}} - 1
\]

where \(LOC_{t-1}\) is the LOC of the first snapshot, \(LOC_t\) is the LOC of the next snapshot, and \(\text{days}\) is the number of days between the dates of the snapshots. Figure 8.3 shows an example of a system with 5 snapshots. In our analysis, snapshots 1, 4 and 5 would be selected, and two growth numbers would be obtained: between snapshots 1 and 4 and between snapshots 4 and 5, which would then be scaled back to 365 days.

Figure 8.3 shows the result of this analysis. Presented are yearly growth percentages with different selection criteria. For instance, the first row in the table shows the mean,
standard deviation and 95% confidence interval for the yearly growth percentage when no systems are excluded. However, there are several systems with growth percentages which could be considered unrealistic, and which are likely in error, such as a growth percentage of 10,000%. Also, depending on the required definition, negative growth could be excluded. The second row of the table shows summary statistics when growth percentages smaller than 0 and larger than 300% are excluded. This leads to a lower average growth of 10.9%. When including negative growth up to -50%, the growth percentage further decreases to 10.3% per year.

<table>
<thead>
<tr>
<th>Range</th>
<th>μ</th>
<th>σ</th>
<th>95% low</th>
<th>95% high</th>
</tr>
</thead>
<tbody>
<tr>
<td>(−∞, +∞)</td>
<td>14.9%</td>
<td>10.9%</td>
<td>11.9%</td>
<td>18.6%</td>
</tr>
<tr>
<td>[0, 300%]</td>
<td>10.9%</td>
<td>6.2%</td>
<td>7.7%</td>
<td>11.0%</td>
</tr>
<tr>
<td>[−50%, 300%]</td>
<td>10.3%</td>
<td>6.3%</td>
<td>7.4%</td>
<td>10.6%</td>
</tr>
</tbody>
</table>

Table 8.3: Statistics of yearly growth percentage with different selection criteria.

Note that there is a danger in multiplying, i.e. geometrically linking, these growth rates to calculate a growth rate over multiple years. This would lead to a size estimate that increases exponentially over time. Depending on the assumptions taken, the growth rate could also be calculated as a percentual increase from a given base year size, which results in linear growth.

8.4. DISCUSSION

We have calculated absolute and percentual growth rates for industrial and open-source software systems. On average, industrial software systems grow with 17 lines of code per day and open-source software systems grow with 10.8% per year. Larger industrial software systems tend to grow at lower speeds and systems bigger than 500,000 LOC tend to decrease in size.

Practitioners can use our results in two ways. First, the size of a software system after a period of time can be predicted using the growth rates as calculated in this chapter. Alternatively, the development time implied by these analyses can be calculated. This can be done both for the percentual growth rate as well as the absolute growth rate. The numbers in Table 8.2 and Figure 8.2 can be used to make such a prediction. For instance, a software library of 100,000 LOC and a growth speed of 10.8% per year is expected to have a size of $100,000 + (100,000 \times 0.108 \times 5) = 154,000$ LOC after 5 years, assuming linear growth.
To estimate the development time of an industrial software system using absolute growth, the following steps can be taken. Assuming that the system will eventually consist of 100,000 LOC, the first 10,000 LOC will be developed, according to the numbers in Table 8.2, with a speed of 5.60 LOC per day. The next 40,000 LOC will be developed with a speed of 16.22 LOC per day. In total, this leads to an expected development time of $(10,000/5.60 + 40,000/16.22 + 50,000/124.17)/365.25 = 12.7$ “team-years”. With a team size of 7 full-time developers, the total development time is expected to be $12.7/7 = 1.8$ years.

The negative coefficient of -41.17 LOC per day implies that a software system that is expected to be larger than 500,000 LOC will never be finished because the final size will never be reached due to the negative growth coefficient. This may be a too literal interpretation this model, but it nevertheless shows that it may be unrealistic to expect the same development speed from a team that works on a system of 500,000 LOC and a team that works on a smaller system.

A possible explanation for the slower development speed for smaller systems is that there is a certain amount of overhead present in each software project, such as designing the architecture, setting up the infrastructure and communication between team members. This has to be done for each project, but in a larger project, the same amount of overhead is distributed over a larger amount of LOC. This would lead to a lower productivity in smaller software projects. Another explanation is that smaller software systems are more efficient in implementing the same amount of functionality with fewer LOC, a concept that was introduced in Chapter 7 and which will be investigated further in Chapter 9. This would lead to an apparent lower productivity as measured in LOC for smaller systems, but this is only because an increase in LOC does not actually measure an increase in functionality. Nonetheless, these models can still be used to perform LOC-based predictions with.

Future work could further investigate different factors influencing the speed of growth for software systems. For instance, different variables, such as team size, experience of developers or other factors could be included in the fixed effects linear regression model to find out if these factors contribute to the growth speed or reduce it.

**8.5. CONCLUSION**

In this chapter, we have presented two growth models for industrial and open-source software systems. Open-source Java libraries tend to grow with 10.8% per year, and industrial software systems tend to grow with 17 LOC per day, but their growth depends strongly on the size of the system. The optimal system size in terms of growth is between 50,000 and 100,000 LOC. Larger industrial software systems grow slower, and systems larger than 500,000 LOC even tend to decrease in size. This may have several explanations. First, larger software systems may simply be finished or may become unmaintainable. In this case, a solution could be to split up the system into smaller parts. Smaller industrial software systems also have a reduced growth speed, which may be an artifact of measuring system growth in LOC.
As we have demonstrated in previous chapters, changes in public interfaces of software libraries are common. To quantify these changes, we have defined several metrics and we have measured the impact of them by measuring the amount of work done in client systems caused by these changes. In this chapter, we present a theoretical framework that integrates several of the metrics that we have defined previously. We connect the concepts of software growth, size, work done and software quality and we investigate the relationships between them. The central concept that we investigate is software growth, which we assume takes two forms: desirable and undesirable growth. We define desirable growth as an increase in the amount of unique functionality provided by a software system, and undesirable growth as an increase in the amount of code without an increase in the amount of unique functionality provided by a software system. We use compression to make a distinction between these two types of growth. We argue that the ultimate goal of software developers should be to achieve long-term sustainable software growth.

9.1. INTRODUCTION
Most software operates in environments where requirements keep changing and new functionality must be implemented to satisfy the changing needs of end users of the software. This puts demands on software developers and software systems themselves, which are constantly changed to incorporate new functionality. Additionally, software developers need to spend effort to keep their system maintainable and flexible for future changes.

The goal of this chapter is to present and investigate a theoretical framework that connects different manifestations of software evolution over time. The main concept we investigate is software growth and different metrics that measure this growth. In this study, we make a distinction between two types of software growth: “desirable” and “undesirable” growth. We define desirable growth to be an increase in the amount of unique functionality provided by a software system. We define undesirable growth to be an increase in the size of a system without an increase in the amount of unique functionality
as provided by a software system. An example of desirable growth is the addition of new methods that provide functionality that was not previously included in the system. An example of undesirable growth is the addition of methods that duplicate functionality which was already present in another place in the system (i.e., copy-and-paste).

We argue that the ultimate goal of software developers should be to achieve long-term sustainable growth of their software systems, meaning that they should aim to reach their optimal productivity in terms of new functionality added. Additionally, changes to their software system or newly added functionality should not erode the existing architecture of the system, decrease system maintainability or make changing the system harder in the future. When the future changeability of a software system is not taken into account when adding new functionality to a system, it becomes increasingly harder to add new functionality to it in the long run. These mechanisms have long been recognized in the software engineering research community [12, 17, 22, 25, 33, 52, 79, 101].

We use the concepts of compression and Kolmogorov complexity to measure desirable and undesirable growth. With compression, the true information content of code can be calculated, and any form of duplication, literal or structural, can be eliminated. The true information content of a system is defined as the minimal amount of information that is needed to represent the functionality present in that system. By measuring the size of the true information content instead of the number of lines of code, the amount of new functionality, instead of simply an increase in LOC, can be measured. Additionally, again by using compression, the amount of undesirable growth can be measured.

We apply a metric known as the Normalized Compression Distance [74] (NCD) to measure the functional growth in software systems. We present a unified model that incorporates these metrics and we test this model on a set of more than 100,000 open-source Java software libraries and a large set of industrial software systems. Finally, we discuss the practical implications of our results for practitioners such as software developers and software architects.

The framework as proposed in this chapter can give developers of software systems insight in processes underlying the evolution of their system and can help them to achieve sustainable software growth over time. This chapter can motivate software developers and architects to think about the software they create in a different light, which can make them more aware of the quality of the systems they create, and could ultimately lead to more changeable, future-ready software systems. While the effects of software evolution as studied in this work have long been recognized, to our knowledge, our study is the first to include a distinction between desirable and undesirable growth, to introduce a measurement model that can measure these concepts and to test it on a large dataset.

In this study, we aim to answer the following research questions:

- How can we measure desirable and undesirable software growth?
- How do the concepts of software growth, software size, software quality and developer productivity relate to each other?
- What is the influence of software size on software growth and vice versa?
- What is the influence of software quality on software growth and vice versa?
- What factors influence the functional growth of a software system?
Our approach to investigate software growth and factors influencing this growth is novel because we use compression to detect functional growth, which has not been applied on this scale before. Using compression, it becomes possible to perform a fully automated source code analysis without resorting to manually counting function points. This enables us to investigate concepts related to software growth in a large dataset of more than 100,000 software libraries.

9.2. Theory and Hypotheses

In this section, we present a theoretical background and hypotheses how we expect that the concepts of software growth, work done, quality and size relate to each other.

We have presented the growth of software in absolute and percentual terms in Chapter 8. We have investigated the absolute growth in industrial software systems and the percentual growth in open-source software libraries. The absolute growth of industrial software systems depends strongly on the size of the software system and peaks at 124.17 LOC per day in software systems between 50,000 and 100,000 LOC. The percentual growth of open-source libraries lies around 10% per year.

As was discussed in Chapter 8, the growth of software in absolute terms (growth in total number of lines of code) has been investigated before, with the goal of obtaining a growth function that expresses the size of the software as a function of time [48, 49, 55, 66, 96]. The goal is often to investigate whether current growth figures can be maintained in the long run, with the underlying idea that the bigger software gets, the harder it must be to maintain the same percentual growth rate. This effect has long been recognized by Lehman’s famous laws of software evolution [69–71]. The law of continuing growth states that “the functional content of an E-type system must be continually increased to maintain user satisfaction over time”. This is often perceived as stating that the size of a software system must increase in order to stay relevant [56].

If the law is assumed to be true, and the average team size is not expected to be scaled up with the size of the software, it must follow that larger software systems grow slower than smaller software systems, since generally speaking, the same number of developers is working on a larger amount of code, thus reducing the time they can spend on any part of the system.

Therefore, we posit H1:

\[ H1. \text{Large software libraries grow slower than small software libraries.} \]

Besides growth in absolute numbers, the compound annual growth rate has been proposed as measurement of software growth[109]. A compound growth rate assumes that software grows exponentially with a constant percentual amount per year. In this study, however, we assume that software does not grow exponentially but rather with a constant amount per unit of time.

Closely related to growth in absolute terms is the amount of work that is performed. An amount of work performed on a software system does not automatically translate to an increase in system size because a system can also be refactored, thus leading to a constant or decreasing amount of code while work is still being performed. For this reason, we consider work done and growth to be two different concepts in this study. Larger systems have more code that needs to be maintained and require more work to keep their level of functional growth constant, as in Lehman’s law.
However, due to diseconomies of scale, we expect that less work can be performed on larger systems because it takes more time to get an understanding of the context of each change and to locate certain features. This could manifest itself through a negative relationship between work done and system size, or a positive relationship with a "diminishing return" of work done. This would mean that the amount of work that is performed on systems initially increases with system size, but after a certain point becomes constant or even decreases.

Therefore, we posit H2:

**H2. Larger systems require more work, but their size inhibits the required higher productivity.**

The effect of software quality on development effort has been investigated extensively [16, 22]. Overall, higher software quality is assumed to influence the productivity of developers positively. In this study, we assume that the effect of system quality on work done can be explained by two mechanisms:

1. An increase in system quality can reduce the amount of work that needs to be performed because of a better architecture, low coupling and high cohesion of classes and packages [22, 33]. Developers only need a small amount of code to make a certain change. This is desirable since less code that needs to be written or changed means less time to implement that change.

2. An increase in system quality can increase the amount of work that can be performed because a better architecture, low coupling and high cohesion can enable developers to be more productive. Each change may need a smaller amount of code, but the overall number of changes a developer can make increases.

The distinction between these mechanisms makes clear that a high developer productivity number is not necessarily something that is desirable: does a large amount of work done indicate that people must perform a lot of work to implement a certain change, or does it mean that they can implement a lot of changes? Developer productivity as measured by changes in LOC or frequency of releases suffer from a lack of distinction between these two concepts. This study introduces compression and Kolmogorov complexity to make this distinction measurable. We expect that higher quality software enables developers to be more productive in terms of implemented functionality as measured by the NCD.

Therefore, we posit H3:

**H3. Higher quality software makes developers more productive in terms of newly implemented functionality.**

In this chapter, we use compression in two different metrics. First, we use it as a measure of software quality in a metric called the compression ratio. The more a library can be compressed, the more duplication and redundancy is present in the library. It indicates that the same amount of functionality could have been written in a more compact form. Second, we use it as a measure of functional growth in a metric called the Normalized Compression Distance (NCD). By compressing two subsequent versions of the same library and normalizing the size difference, we obtain a measure for the amount of code that has been changed between these versions while ignoring any duplication and redundancy. Instead of calculating the number of changed lines or differences in lines
of code, the normalized compression distance can be used to calculate the “functional distance” between system snapshots. We expect that the NCD is positively related to the amount of work performed. The more work (in terms of changed lines of code) is performed in a new version of a system, the more likely it is that there are large functional changes in this work. The NCD has been proven to be a metric that can accurately measure the “distance” between two pieces of information[27, 74, 75, 112]. To our knowledge, it has not been applied before on a large scale to measure software growth.

Therefore, we posit H4:

**H4. The Normalized Compression Distance can be used to measure the amount of new functionality as compared to the previous version of a software system and is positively correlated with the work done in that system.**

Since low quality software typically contains a large amount of duplication, we expect that high compression ratios are associated with lower quality software as measured by the maintainability rating of SIG. The maintainability rating is explained in more detail in Section 9.6.

Therefore, we posit H5:

**H5. Higher compression ratios are associated with lower quality software.**

9.3. **Dataset and Data Collection**

The conceptual model as introduced in this chapter is tested on data of 1,538 snapshots from open-source Java libraries from our Maven Dependency Dataset. For this chapter, we selected complete cases (listwise deletion), leading to the selection of 1,538 snapshots for our analysis (1.1% of all library snapshots). Summary statistics for our dataset are provided in section 9.7.

For performance reasons, data for our analysis was obtained using the DAS-3 Supercomputer, a supercomputer with 50 computing nodes. Jar files for each library version were extracted and source code was parsed using the Eclipse JDT Core API\(^1\). Software was written to extract information such as LOC, the cyclomatic complexity and the number of methods from source code. The maintainability rating was obtained by running the Software Analysis Toolkit (SAT) of the SIG on the supercomputer. Compressed sizes were obtained by running the Unix command-line tool `bzip2`.

Compression ratios for each language were calculated by collecting all files in a system written in a certain language. For each system snapshot, consultants of the SIG have determined which files in the system belong to which programming language. This information is stored in a configuration file, and these files were used to select the proper files for each language. From these files, the total size of the source code was calculated by adding the size of all individual files in bytes. All files belonging to the same language in the same system were added to a zip file using the `bzip2` compression algorithm. The file size of this zip file in bytes was stored.

In the next section, we describe the relationship between new functionality and new code that is assumed to hold in each change made in a system.

\(^1\)https://projects.eclipse.org/projects/eclipse.jdt.core
9.4. Growth Types

When a developer adds new code to a system, the amount of functionality present in that piece of code is not necessarily constant. It is possible that a developer duplicates a piece of existing code, thus effectively increasing the amount of code in the system without adding new functionality to it. The theory of compression and Kolmogorov complexity implicitly assumes that there exists a relationship between new functionality and new code that gets added to a system. Figure 9.1 shows this relationship. For a discussion on situations in which this assumption might not hold, see Section 9.10.

![Figure 9.1: The relationship between new functionality and new code for each change to a software system as assumed in this chapter.](image)

In the upper left quadrant of the figure, a change is introduced that adds new functionality to the system without adding more code to it. According to the theory of Kolmogorov complexity, this situation is impossible since each unique piece of functionality must be represented by a minimum amount of information, in this case code. In our theory, we assume that this situation is desirable from a maintenance perspective since no extra code has to be maintained to serve a larger amount of functionality, but is impossible to achieve.

The lower left quadrant shows a change in which no new functionality is added to a system and also no new code is added. This is a change that does not implement anything new and contains no new code. Refactorings are an example of such changes, assuming the refactoring does not introduce new code but only transforms code. An example of such a change is a method that is moved to another class. If only this type of changes are applied to a system in the long run, the system essentially does not evolve any more.

In the lower right quadrant, a change is shown that adds no new functionality to a system but does add more code to it. This can happen when a developer copies and pastes a piece of functionality into another part of the system. In this case, the total amount of code increases but the total amount of functionality stays the same. We assume that this type of change is always undesirable. Finally, the upper right quadrant shows the most practical and desirable situation all changes should be classified in: the change adds more functionality to the system and also adds more code to it.
Most changes as made to software systems can not be easily categorized in this diagram since in practice, each change likely contains new functionality as well as new code, but in different proportions. This gives rise to the concept of the F/C-ratio, which we define as the ratio of the amount of functionality to the amount of code for a change. The goal of the developer is to find an optimal value for the F/C-ratio.

The higher the F/C ratio is, the less code is needed to express the same concept. The F/C-ratio can also be too high, however, for example when a single line of code contains a lot of functionality but becomes incomprehensible. For this reason, the second concept that has to be optimized by the developer is the understandability of the created code. In this chapter, we assume that code can also become too verbose (low F/C-ratio) and therefore incomprehensible.

To make the amount of functionality present in any amount of code measurable, we use the concept of Kolmogorov complexity and compression. The concept of Kolmogorov complexity is explained next.

9.4.1. **Kolmogorov Complexity**

As was already explained in Chapter 7, the Kolmogorov complexity [67] of a piece of information can be described as the smallest size that piece of information can be compressed to. For instance, consider the strings

```
abababababababababababababababababab
```

and

```
wboijetoiuatpioywlgibvijefiociplnqde.
```

While both strings have the same number of characters (36), the first string can be much shorter described as “18 times ab”, while the second string of characters cannot be described any shorter, since it is a random sequence of characters.

The Kolmogorov complexity of a string of characters is theoretically incomputable, but can be approached from above with zip algorithms such as bzip2 [75]. To obtain the shortest string representation of a software system, all source code in a software system can be concatenated and then be compressed using bzip2.

Conceptually, zip algorithms are capable of removing any duplication or redundant information in code and can reduce a system to its true information content. For instance, when a software system contains a method that is duplicated across different places, the zip file will only store one complete reference to the method and point to that reference in other places in the zip file (dictionary compression). The algorithm is also capable of detecting common structure in different places in the source code, such as a complex for-loop that is duplicated in different places, although identifiers may be different. The size of the zipped file thus represents the “essential complexity” which remains after removing any redundancy, duplication, or “structural inflation”.

9.4.2. **Compression ratio**

When a system has grown in size but not in functionality (undesirable growth), zipping the file would deflate the system to a size which should be the same as before the system grew. When a system has grown in functionality and in size (desirable growth), the compressed file would be larger, which would reflect that new functionality was added to the system.
We use compression to calculate the true information content of code, which is defined as the minimal amount of information that is needed to represent the functionality in a software system. Because the same amount of functionality can be encoded in more or less characters in different languages, compression can be used to find the “essential” information present in a software system.

An increase in the compression ratio of a software system as compared to its previous version is considered to be a sign of undesirable growth since it indicates that the relative size of the true information content of the system has reduced, i.e. the source code can be “deflated” further to reduce its size to its true information content. We assume that the higher the compression ratio for a certain system is, the lower its quality will be.

Figure 9.2 shows an example of two versions of a library and their compressed and uncompressed sizes. Version 2 of the system has expanded significantly in size, but when zipping the source code, it turns out that the amount of actual functional growth is much smaller than the total growth in LOC would suggest. The compression ratio of a software system is defined as the size of the uncompressed source code of a certain language in bytes divided by the size of the compressed source code in bytes. It represents the number of times the file size can be reduced when zipping the source code.

![Figure 9.2: An example of two versions of a software system, v1 and v2, and their compressed and uncompressed source sizes.](image)

In this study, we calculate the compression ratio for each software library by calculating the sum of the individual source file sizes in bytes, compressing all source files, and calculating the size of the archive with the compressed source files in bytes. These two numbers are then divided to get the compression ratio.

**9.4.3. Normalized Compression Distance**

The compression ratio can serve as a measure of undesirable growth since a larger compression ratio implies that more code is needed to represent the same information. It
cannot, however, measure the amount of desirable growth between snapshots. Therefore, an additional measure based on Kolmogorov complexity is needed.

We use the Normalized Compression Distance (NCD) [74] as measure for desirable growth, which represents the functional distance between two snapshots. The NCD is defined as follows:

\[ \text{NCD}(x, y) = \frac{C(xy) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}} \] (9.1)

where \( C(x) \) is the compressed size in bytes of the source code of \( x \), and \( C(xy) \) is the compressed size of the concatenated source code of both \( x \) and \( y \).

The metric works as follows. When no work was performed between two snapshots of a system (i.e. the snapshots are identical), a compression algorithm will be able to reduce the compressed size of the combined source code by half, since the second part of the compressed file is simply an exact copy of the first, and the compressed file will eliminate this duplication. In this case, the NCD will be 0. When two snapshots of a system are completely different, the NCD will be 1. A larger NCD indicates a larger amount of desirable growth because a larger amount of new functionality has been implemented since the previous snapshot.

### 9.4.4. OTHER APPLICATIONS OF COMPRESSION

The NCD has been applied in large variety of domains to find the similarity between various sources of data. For instance, it has been applied to cluster music by genre [27, 112], to build an evolutionary tree of different mammals [24, 27] or to classify natural languages [74]. Since the metric is indifferent between the underlying semantics of the information, the metric can be applied to any stream of bytes with a finite character set. In the field of software engineering, the normalized compression distance has been applied before to study the functional distance between software releases [7]. The authors apply the NCD on ArgoUML and they validate the metric on measured evolution against experimental data from other studies. Similarly, Veldhuizen [111] has investigated how well software libraries compress, and found that the extent to which software reuse occurs is an intrinsic property of its problem domain.

Woodside [113] proposed a mathematical model in which concepts are used which resemble desirable and undesirable growth. Woodside proposed that there exists a balance between progressive and anti-regressive work in software development. Progressive work introduces new features in the system, while anti-regressive work is “aimed at maintenance of the software system, keeping it well-structured, documented and evolvable” [15]. Our notion of desirable growth corresponds with the concept of progressive work, but our notion of undesirable growth does not correspond with the concept of anti-regressive work, since anti-regressive work is also desirable. In our model, there does not exist a separate concept that represents anti-regressive work.

In the next section, we describe our conceptual model and the variables that make up the model. After this, we show the results of the model and we investigate relationships in the model separately using regression and correlation analysis.
The model that integrates the concepts that are tested in this chapter is shown in Figure 9.3. The figure shows several concepts, which we do not measure directly but through different metrics (shown in rectangular boxes). The model shows that in our theory, the abstract concept “system growth” consists of both desirable and undesirable growth, which are in turn measured by several metrics. Desirable growth can be measured as the NCD between the two snapshots and the number of new methods since the previous snapshot. Undesirable growth is measured by the increase in the average cyclomatic complexity, the increase in the duplication score and the increase in the compression ratio between the two snapshots. These metrics are explained in detail in Section 9.6. System size is measured by the closely related metrics LOC, number of files and the number of methods in a software system. Work done is measured as the edit script size between two snapshots, which is explained in Section 9.6.4. System quality is measured with the Maintainability rating of the SIG, which is explained in Section 9.6.1.

9.5.1. RELATIONSHIPS

The relationships as shown in the model are explained as follows. The amount of work done influences system growth in a positive way since the more work is done on a system, the faster it will grow. The relationship between system growth and system size is also obvious: the faster a system grows the larger it size will become. The relation-
ship between system size and the amount of work done is negative since we expect that the larger a system becomes, the less work can be done in any single unit of time since it takes more time to find the right place to change the code and it becomes harder to change it. Undesirable growth is expected to influence system quality in a negative way since a large amount of undesirable growth will erode system architecture and will reduce the changeability of the system. System quality is expected to influence work done in a positive way since a good system architecture and a maintainable system are expected to increase the amount of work that can be done on the system in any amount of time.

9.6. Measurement Variables

Besides the concepts of desirable and undesirable growth as described previously, our theoretical framework also defines several other concepts, which are explained in this section.

9.6.1. Maintainability Rating

We use the maintainability rating of the SIG [53] as a proxy for software quality. The maintainability model uses several metrics to measure maintainability, which is, according to ISO 25010 [1], a sub-property of software quality.

The ISO model defines five sub-properties of maintainability: analyzability, modifiability, testability, modularity and reusability. The model does not, however, specify how these properties should be measured. For this reason, the maintainability model defines several metrics to measure these concepts: volume, duplication, method size, method complexity, unit interfacing, module coupling, component balance and component independence. A weighted average for each sub-property is used to aggregate the values into an overall maintainability rating [4, 5].

The rating represents a percentile against a benchmark of other industrial software systems and ranges from 0.5 to 5.5, but as Table 9.1 shows, ranges from 1.91§ to 4.75 with an average of 3.53 in our dataset. When the maintainability rating is 0.5, the system scores has the lowest quality as compared to the benchmark, and when the score is 5.5, it is in the highest percentile of the benchmark. A high score indicates a maintainable software system.

9.6.2. Duplication

In literature, it is generally assumed that code clones are considered harmful to the maintenance effort and quality of a software system [10, 59, 99]. For instance, when a bug is found in a duplicated piece of code and this bug has to be fixed, each instance of the clone must be adapted. This also opens the possibility that subtle errors are introduced [26, 76]. Additionally, if a change must be made to a cloned piece of code, the change must be made to all instances. The maintenance effort must thus be increased for every instance of code that must be changed [110]. In this study, we will assume that an increase in duplication in a software system is a sign of undesirable growth.

To calculate duplication, we detect the number of exact clones in a system. We define an exact clone as a piece of code that is literally copy-and-pasted into another place
in the system. We obtain a single duplication measurement for a software system by applying the following formula:

$$\ln\left(\sum_{i=1}^{nrClones} \text{matches}_i \times \text{lines}_i \times \text{nrDiffFiles}_i\right)$$

(9.2)

with \textit{matches} being the number of times a specific clone is found, \textit{lines} the number of lines of the duplicated piece of code and \textit{nrDiffFiles} the number of different files the clone is found in. We multiply each of these properties because we assume that a clone with more instances in multiple files, a larger number of copies and a larger number of lines should weight heavier than clones with less lines and less instances in a fewer number of files. We take the natural logarithm of the sum of the product over all clones because inspection of the distribution of the metric shows that it is lognormally distributed. The following constraints apply: \textit{matches} \geq 2, \textit{lines} \geq 5 and \textit{nrDiffFiles} \geq 1 for each clone, indicating that we only investigate clones that consist of more than five lines of code.

The metric only takes into account exact clones, but not code that is changed slightly, such as a duplicated method with changed variable names or values. Higher-level clones are not taken into account. However, we assume that the number of exact clones serves as an indicator for the total amount of duplication present in the system. In our model, we use the natural logarithm of change in the duplication score as compared to the previous snapshot of the library in our model. The Copy/Paste Detector (CPD) of the PMD tool\(^2\) was used to calculate the duplication score.

### 9.6.3. AVERAGE CYCLOMATIC COMPLEXITY

We define the average complexity of a system as the sum of the cyclomatic complexity of all methods in the system divided by the number of methods in that system. The cyclomatic complexity (also called the \textit{McCabe} value) is a measure of the number of unique paths through each method [77]. The increase in the average cyclomatic complexity serves as a proxy for undesirable growth since it is commonly understood that large, complex methods decrease the understandability of a system and reduce its changeability [14, 43]. Preferably, smaller, less complex methods that implement the same functionality should be added instead.

In our model, we use the natural logarithm of the increase in the average cyclomatic complexity between snapshots:

$$\ln\left(\frac{\sum_{n=1}^{N_2} \text{cc}_{i2}}{N_2} - \frac{\sum_{n=1}^{N_1} \text{cc}_{i1}}{N_1}\right)$$

(9.3)

where \text{cc}_{i2} is the cyclomatic complexity of the \text{ith} method in the second snapshot, and \text{N}_2 is the number of methods in the second snapshot.

The SIG maintainability model uses quality profiles instead of simple averages as an aggregation method. The first reason for this is to enable root-cause analysis. Another reason is that in power law distribution, which the cyclomatic complexity of a software

\(^2\)http://pmd.sourceforge.net/pmd-5.1.1/cpd-usage.html
system generally follows, the average cyclomatic complexity will almost always be low. This masks problems with the few methods that have a high cyclomatic complexity [53].

In our model, however, we do not need to perform a root cause analysis and we are not interested in the absolute average cyclomatic complexity of a single snapshot, but rather in the difference between two snapshots. The difference will detect an increase in the complexity between snapshots, regardless how the underlying distributions changed or which methods caused the increase.

9.6.4. Work done
The amount of work done is included separately in our model because it is conceptually different from the absolute or relative change in LOC since the previous version of a system. Work done is measured as the edit script size between two versions. The edit script between two versions contains the statements that must be inserted, deleted, moved or updated to transform the first version of the system into the second one. The size of the edit script cannot be directly translated into effort in terms of man-hours since two edit scripts of the same length can each take a different time to implement, but it can nonetheless serve as an indicator for this effort. We calculate the edit script size between each pair of files between different versions of two snapshots separately. We then add these numbers to obtain a total for a pair of system versions. ChangeDistiller was used to calculate the edit script between two system versions.

9.6.5. Number of new methods and ΔLOC
The number of new methods are a measure of desirable growth since we assume that in general, developers add new methods when they are adding new functionality. We also include the change in LOC as measure of desirable growth to increase the number of measurement variables for the latent construct desirable growth. It is expected that by combining NCD and ΔLOC in the same latent construct, any increase in ΔLOC that remains after correcting for NCD can be attributed to desirable growth.

In the next section, we describe the results of the model. We start by performing a confirmatory factor analysis to investigate the validity of the latent constructs in the model and the degree to which the underlying metrics measure these constructs coherently.

9.7. Summary Statistics
Table 9.1 shows summary statistics for the measurement variables in our dataset. The mean size of the analyzed systems in the Maven repository is 4,099.79 LOC, with 403.5 methods and 48.8 files. There are, however, large differences between system sizes: the smallest system is only 42 LOC and the biggest system is 53,857 LOC. All variables show a large degree of skewness and kurtosis, as can be seen in the last two columns of Table 9.1. The skewness values indicates that most variables are strongly positively skewed, with a small chance of extremely large outliers.

The average change in LOC is 352, and the average edit script size is 134.5, indicating that on average, 134.5 statements need to be deleted, inserted, added or updated to convert a software system into its next version. There are only 1.98 new methods per re-
lease, on average. The average software system tends to get more complex with a change in average cyclomatic complexity of 0.16. Also, systems tend to get more duplication, on average, with a mean change in duplication score of 0.05. Finally, the average system quality tends to decrease with a change in compression ratio of 1.01, on average. These metrics are explained in more detail in Section 9.6.

Although the theoretical maximum for the NCD is 1, the maximum value of NCD as observed in our dataset is 1.04. This is expected since any compressor adds an additive constant to the NCD, which can be thought of as overhead in the zip algorithm [75].

### 9.7.1. Correlations

Table 9.2 shows the Spearman rank correlations of the measurement variables in our dataset.

All correlations are significant with $p$-values of 0. The correlation between maintainability and edit script size is negative with a value of -0.35, indicating that less work tends to be performed on more maintainable systems ($H4$). The correlation between edit script size and LOC, number of methods and number of files is positive with values of 0.44, 0.37 and 0.35, respectively. This indicates that larger systems require larger amounts of work ($H2$). The table further shows that changes in compression ratio, duplication and average cyclomatic complexity tend to move together, with correlations of 0.66, 0.53 and 0.62. This gives support for the latent construct of undesirable growth that consists of these metrics.

Figure 9.4 shows scatter plots of four correlations from this table. The correlation between maintainability and the compression ratio is negative with a value of -0.35, indicating that less maintainable systems can be compressed further. The correlation between NCD and the number of new methods is positive with a value of 0.407, indicating that new methods are associated with an increase in the compression distance. The correlation between NCD and edit script size is positive with a value of 0.61, and the correlation between the duplication score and the compression ratio is positive with a value of 0.473.

The correlations provide a confirmation for $H4$: Functional distance is positively related to the amount of work performed and negatively related to duplication, maintainability and the compression ratio. The numbers also provide support for our construct undesirable growth, consisting of the increase in the average cyclomatic complexity, the change in duplication score and the change in compression ratio. In Section 9.9, we provide further support that the concepts of desirable and undesirable growth can be represented by the measurement variables as proposed in this chapter.

### 9.8. Individual Relationships

#### 9.8.1. System Size and System Growth

To investigate $H1$, we perform a regression analysis of system growth on system size. Table 9.2 shows that there exists a positive relationship between the number of new methods and the number of existing methods in a system of 0.29. Similarly, between

---

$^3$These correlations are different from Table 9.2 because data was selected to be non-missing for these four variables separately, leading to 26,115 observations instead of the 1,538 of Table 9.2.
<table>
<thead>
<tr>
<th>Variable</th>
<th>( \mu )</th>
<th>( \sigma )</th>
<th>min</th>
<th>p5</th>
<th>p25</th>
<th>p50</th>
<th>p75</th>
<th>p95</th>
<th>max</th>
<th>( \gamma_1 )</th>
<th>( \gamma_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit script size</td>
<td>134.48</td>
<td>268.73</td>
<td>1</td>
<td>5</td>
<td>23</td>
<td>59</td>
<td>141</td>
<td>526.6</td>
<td>4396</td>
<td>7.44</td>
<td>84.47</td>
</tr>
<tr>
<td>Maintainability</td>
<td>3.53</td>
<td>0.58</td>
<td>1.91</td>
<td>2.61</td>
<td>3.08</td>
<td>3.48</td>
<td>3.97</td>
<td>4.52</td>
<td>4.75</td>
<td>0.14</td>
<td>2.3</td>
</tr>
<tr>
<td>LOC</td>
<td>4,099.79</td>
<td>5,373.84</td>
<td>42</td>
<td>259.7</td>
<td>861.25</td>
<td>1,943.5</td>
<td>5,429.25</td>
<td>14,308.6</td>
<td>53,857</td>
<td>3.02</td>
<td>17.99</td>
</tr>
<tr>
<td># methods</td>
<td>403.46</td>
<td>544.36</td>
<td>2</td>
<td>20</td>
<td>69</td>
<td>180.5</td>
<td>519.5</td>
<td>1578.25</td>
<td>5172</td>
<td>2.68</td>
<td>14.16</td>
</tr>
<tr>
<td># files</td>
<td>48.81</td>
<td>62.06</td>
<td>1</td>
<td>3</td>
<td>10</td>
<td>23</td>
<td>66.75</td>
<td>168.15</td>
<td>489</td>
<td>2.51</td>
<td>11.36</td>
</tr>
<tr>
<td>NCD</td>
<td>0.42</td>
<td>0.22</td>
<td>0.17</td>
<td>0.21</td>
<td>0.25</td>
<td>0.33</td>
<td>0.51</td>
<td>0.95</td>
<td>1.04</td>
<td>1.26</td>
<td>3.45</td>
</tr>
<tr>
<td># new methods</td>
<td>1.98</td>
<td>1.43</td>
<td>0</td>
<td>0</td>
<td>0.69</td>
<td>1.79</td>
<td>2.94</td>
<td>4.53</td>
<td>8</td>
<td>0.5</td>
<td>2.86</td>
</tr>
<tr>
<td>( \Delta \text{LOC} )</td>
<td>352.94</td>
<td>1,530.61</td>
<td>1</td>
<td>15</td>
<td>46</td>
<td>115</td>
<td>302.75</td>
<td>1,128.55</td>
<td>42,008</td>
<td>20.35</td>
<td>493.93</td>
</tr>
<tr>
<td>( \Delta \text{avg mccabe} )</td>
<td>0.16</td>
<td>0.39</td>
<td>0</td>
<td>0</td>
<td>0.01</td>
<td>0.04</td>
<td>0.15</td>
<td>0.7</td>
<td>9</td>
<td>10.5</td>
<td>197.04</td>
</tr>
<tr>
<td>( \Delta \text{dup. score} )</td>
<td>0.05</td>
<td>0.12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.01</td>
<td>0.05</td>
<td>0.24</td>
<td>1.48</td>
<td>5.17</td>
<td>40.02</td>
</tr>
<tr>
<td>( \Delta \text{comp. ratio} )</td>
<td>1.01</td>
<td>2.38</td>
<td>0</td>
<td>0.01</td>
<td>0.07</td>
<td>0.27</td>
<td>0.91</td>
<td>4.51</td>
<td>44.66</td>
<td>7.25</td>
<td>92.7</td>
</tr>
</tbody>
</table>

Table 9.1: Summary statistics for the variables in the structural model. \( \gamma_1 = \text{skewness} \), \( \gamma_2 = \text{kurtosis} \).
the number of new methods and the number of existing files a correlation of 0.26 exists.

This indicates that a slightly positive relationship between system size and system growth exists. To take into account the possible effect of “diminishing return” of system size on system growth, i.e. whether there exists a quadratic relationship between system size and system growth, we include the square of the number of methods in the regression model [21]. If there indeed exists a quadratic relationship between system size and system growth, the coefficient of the original \( \ln(\# \text{ methods}) \) and the squared \( \ln(\# \text{ methods})^2 \) will both be significant. Table 9.3 shows the results of this model.

### Table 9.3: Regression of \( \ln(\# \text{ new methods}) \) on the number of methods

<table>
<thead>
<tr>
<th>Independent</th>
<th>coeff</th>
<th>Std. err</th>
<th>lower</th>
<th>upper</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ln(# \text{ methods}) )</td>
<td>0.200</td>
<td>0.084</td>
<td>0.035</td>
<td>0.365</td>
<td>0.017</td>
</tr>
<tr>
<td>( \ln(# \text{ methods})^2 )</td>
<td>-0.007</td>
<td>0.008</td>
<td>-0.022</td>
<td>0.008</td>
<td>0.406</td>
</tr>
<tr>
<td>constant</td>
<td>-0.176</td>
<td>0.216</td>
<td>-0.600</td>
<td>0.249</td>
<td>0.417</td>
</tr>
</tbody>
</table>

The model shows that a significant relationship between system size and system growth can be found, although the influence is small with an adjusted \( R^2 \) of 8.14%. This indicates that the larger a library is, the more new methods are added to it. This is the opposite of our expectation and means that larger software libraries tend to grow faster than smaller ones. The squared coefficient, however, is not significant, indicating that there does not exist a quadratic relationship between system size and system growth. Thus, no support for a reduced increase in the number of methods can be found for larger libraries.

We can therefore conclude that no support for \( H1 \) is found in our dataset of Java libraries. **Larger software libraries do not grow slower than smaller software libraries.**

This is in direct contrast with the findings from Chapter 6, in which the growth speed
9.8. INDIVIDUAL RELATIONSHIPS

Figure 9.4: Scatterplots of the relationship between a selection of different variables in our dataset.

of industrial software systems was compared for different system sizes. The growth speed of industrial software systems does become slower as system size increases. A possible explanation for this difference is that the industrial software systems in our dataset are much larger on average ($\mu_{LOC} = 102,514$, $\sigma_{LOC} = 235,866$) than the libraries from the Maven Dependency Dataset ($\mu_{LOC} = 4,099.79$, $\sigma_{LOC} = 5,373.84$), and that the diminishing effect of size on growth only manifests itself after a certain number of lines of code, which is 100,000 in the case of our industrial dataset.

9.8.2. SYSTEM SIZE AND WORK DONE

Table 9.2 shows that the correlation between system size and LOC is positive with a value of 0.44, ($p = 0$), which shows that on average, more work is done on larger systems. To further investigate the relationship between system size and work done, we perform a regression analysis of LOC on edit script size which again includes the squared log-transformed LOC variable. If this variable is significant, this would provide support for an effect of “diminishing return”: more work is performed in larger systems but the amount decreases with the size of the system.

The regression model and the correlation coefficients provides support for our second hypothesis: More work is performed on larger systems. However, the squared coefficient is not significant, indicating that there is no evidence for a diminishing return to scale of system size and work done: the amount of work done on larger systems is linear.

To answer $H2$, Larger systems require more work, but there does not exist a diminish-
### 9.8.3. Maintainability and Work Done

In this section, we investigate the relationship between maintainability and work done. The Spearman correlation coefficient between the SIG maintainability rating and the amount of work done as measured by the edit script size is negative with a value of -0.3465 \((p = 0)\), indicating that the amount of work done on libraries with a higher maintainability rating is less than on libraries with a low rating.

The problem with the correlation between maintainability and the amount of work done is that these variables are both influenced by system size. At SIG, it is known that higher quality software systems are generally smaller. Also, less work needs to be done on smaller software systems. Therefore, a confounding effect of system size exists between system quality and work done. For this reason, we correct the correlation coefficient between the two properties by calculating the partial correlation coefficient between maintainability and work done with the number of methods as correcting factor. The diagram below shows the correlations between these properties:

![Diagram showing correlations between system quality, maintainability rating, work done, edit script size, system size, and number of methods.](image)

The partial correlation coefficient between \(X\) and \(Y\) while correcting for the possible effect of \(Z\) on both variables can be calculated with the following formula \([40]\):

\[
\rho_{x'y'z'} = \frac{\rho_{xy} - \rho_{xz}\rho_{zy}}{\sqrt{(1 - \rho_{xz}^2)(1 - \rho_{zy}^2)}}
\]

\[(9.4)\]

---

**Table 9.4: Regression of \(\ln(\Delta LOC)\) on work done.**

<table>
<thead>
<tr>
<th>Independent</th>
<th>coeff.</th>
<th>Std. err.</th>
<th>lower</th>
<th>upper</th>
<th>(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\ln(LOC))</td>
<td>0.503</td>
<td>0.026</td>
<td>0.453</td>
<td>0.553</td>
<td>0</td>
</tr>
<tr>
<td>(\ln(LOC)^2)</td>
<td>-0.005</td>
<td>0.167</td>
<td>-0.380</td>
<td>0.027</td>
<td>0.749</td>
</tr>
<tr>
<td>constant</td>
<td>0.181</td>
<td>0.197</td>
<td>-0.206</td>
<td>0.567</td>
<td>0.360</td>
</tr>
</tbody>
</table>
In the case of maintainability, work done and system size, \( x \) is maintainability, \( y \) is the edit script size and \( z \) is the number of methods in a system. The result of this equation is -0.19, indicating that a negative correlation between maintainability and work done remains after correcting for the effect of library size. This means that less work needs to be done on systems that have higher code quality, and this effect cannot be attributed to system size.

<table>
<thead>
<tr>
<th>Model statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependent</td>
<td>( \ln(\text{ess}) )</td>
</tr>
<tr>
<td># Observations</td>
<td>1,538</td>
</tr>
<tr>
<td>Adjusted ( R^2 )</td>
<td>22.23%</td>
</tr>
</tbody>
</table>

Table 9.5: Regression of maintainability and system size on work done.

A regression of maintainability and the number of methods on the edit script size shows the same picture, as can be seen in Table 9.5. The relationship between maintainability and \( \ln \) ess is negative with a coefficient of -0.403 (\( p = 0 \)) after correcting for system size. This means that the higher system quality is as measured by maintainability rating, the less work is performed in each new library version.

As stated in Section 9.2, work done in terms of edit script size is not a good measure to investigate the actual amount of new functionality implemented in a snapshot. For this reason, we investigate the relationship between maintainability and NCD in the next section.

### 9.8.4. Maintainability and NCD

To find out if more maintainable software enables the implementation of a larger amount of functionality, we investigate the relationship between maintainability and NCD. In Table 9.2 we see that there exists a negligible negative correlation between maintainability and NCD of -0.09. As we have seen in Section 9.8.3, the correlation between maintainability and work done must be corrected for the confounder of system size. The partial correlation coefficient between maintainability and NCD while correcting for the number of lines of code is 0.05. This means that a negligible positive correlation remains between maintainability and NCD after correcting for system size.

To further investigate this relationship, we perform a regression analysis of maintainability on NCD. The number of methods is included in the regression analysis to correct for the confounding effect of system size. The results of this analysis can be found in Table 9.6.

The table shows that a positive relationship between maintainability and NCD exists after correcting for system size. This indicates that the compression distance increases between snapshots when the system has a higher quality as measured by the maintainability rating. Since the relationship between work done and maintainability was nega-
Table 9.6: Regression of maintainability on NCD.

<table>
<thead>
<tr>
<th>Independent</th>
<th>coeff.</th>
<th>Std. err.</th>
<th>lower</th>
<th>upper</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>0.040</td>
<td>0.010</td>
<td>0.0210</td>
<td>0.059</td>
<td>0</td>
</tr>
<tr>
<td>ln # LOC</td>
<td>0.970</td>
<td>0.005</td>
<td>0.088</td>
<td>0.106</td>
<td>0</td>
</tr>
<tr>
<td>constant</td>
<td>-0.461</td>
<td>0.059</td>
<td>-0.577</td>
<td>-0.344</td>
<td>0</td>
</tr>
</tbody>
</table>

9.8.5. NCD AND DESIRABLE GROWTH

To provide further support that the NCD can be used as an indicator for the functional distance between snapshots, we perform an analysis which relates NCD to edit script size and changes in the number of lines of code. We perform a nonlinear least-squares regression [21] with NCD as dependent and edit script size as independent variable. The reason a nonlinear least-squares regression is performed is that the relationship between increases in LOC and NCD is of an exponential nature, as can be seen in Figure 9.6. The picture also shows the best fitting regression line.

Figure 9.6: The exponential relationship between ln(ΔLOC) and NCD.

The results of the regression of edit script size on NCD can be found in Table 9.7.
The regression is significant with an adjusted $R^2$ of 82.53%. The predicted model is as follows:

$$NCD = 0.1979 \times 1.1910^{\ln(ess)}$$ (9.5)

The model predicts, for instance, that a snapshot with an edit script size of 200 as compared to its previous snapshot, has an NCD of $0.1979 \times 1.1910^{\ln(200)} = 0.50$.

<table>
<thead>
<tr>
<th>Model statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependent</td>
<td>NCD</td>
</tr>
<tr>
<td># Observations</td>
<td>1,538</td>
</tr>
<tr>
<td>Adjusted $R^2$</td>
<td>82.53%</td>
</tr>
</tbody>
</table>

Table 9.7: Nonlinear exponential regression of $\ln(ess)$ on NCD.

Table 9.8 shows the relationship between NCD and $\Delta$LOC.

<table>
<thead>
<tr>
<th>Model statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependent</td>
<td>NCD</td>
</tr>
<tr>
<td># Observations</td>
<td>1,538</td>
</tr>
<tr>
<td>Adjusted $R^2$</td>
<td>85.46%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Independent</th>
<th>coeff.</th>
<th>Std. err.</th>
<th>lower</th>
<th>upper</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b1$</td>
<td>0.1476</td>
<td>0.006</td>
<td>0.135</td>
<td>0.160</td>
<td>0</td>
</tr>
<tr>
<td>$b2$</td>
<td>1.2287</td>
<td>0.009</td>
<td>1.211</td>
<td>1.246</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 9.8: Nonlinear exponential regression of $\ln(\Delta LOC)$ on NCD.

This regression is also significant with an adjusted $R^2$ of 85.46%. The predicted model is as follows:

$$NCD = 0.1476 \times 1.2287^{\ln(\Delta LOC)}$$ (9.6)

When the difference in LOC as compared to the previous snapshot is 1,000, the expected NCD is $0.1476 \times 1.2287^{\ln(1,000)} = 0.61$.

Table 9.9 shows the predicted values of NCD based on edit script size and changes in LOC. The third column shows the expected NCD’s for different values of edit script size in the first column.

We can conclude that the NCD is a proper indicator for newly implemented functionality in a snapshot. Thus, we can confirm $H4$: The Normalized Compression Distance can be used to measure the amount of new functionality as compared to the previous version of a software system and is positively correlated with the work done in that system.
### Table 9.9: Predicted values of NCD for different values of edit script size and change in LOC.

<table>
<thead>
<tr>
<th>ln(x)</th>
<th>x</th>
<th>NCD (x = ess)</th>
<th>NCD (x = ΔLOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.7</td>
<td>0.236</td>
<td>0.181</td>
</tr>
<tr>
<td>2</td>
<td>7.4</td>
<td>0.281</td>
<td>0.223</td>
</tr>
<tr>
<td>3</td>
<td>20.1</td>
<td>0.334</td>
<td>0.274</td>
</tr>
<tr>
<td>4</td>
<td>54.6</td>
<td>0.398</td>
<td>0.336</td>
</tr>
<tr>
<td>5</td>
<td>148.4</td>
<td>0.474</td>
<td>0.413</td>
</tr>
<tr>
<td>6</td>
<td>403.4</td>
<td>0.565</td>
<td>0.508</td>
</tr>
<tr>
<td>7</td>
<td>1,096.6</td>
<td>0.673</td>
<td>0.624</td>
</tr>
<tr>
<td>8</td>
<td>2,981.0</td>
<td>0.801</td>
<td>0.767</td>
</tr>
<tr>
<td>9</td>
<td>8,103.1</td>
<td>0.955</td>
<td>0.942</td>
</tr>
<tr>
<td>9.27</td>
<td>10,614.8</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

#### 9.8.6. Compression Ratio and Undesirable Growth

To provide support for our fifth hypothesis that higher compression ratios are associated with lower quality software, we again inspect Table 9.2 and Figure 9.4. The correlation between the compression ratio and the maintainability rating is -0.35, based on 26,115 library snapshots. The correlation between the compression ratio and the duplication score is 0.473, based on the same number of snapshots. The correlation between an increase in compression ratio between two snapshots (Δcomp. ratio) and an increase in the average cyclomatic complexity between two snapshots (Δavg. mccabe) is 0.53. The correlation between Δcomp. ratio and the difference in duplication score (Δdup. score) is 0.62.

These results indicate that higher compression ratios can be found in systems where methods have a higher cyclomatic complexity, more code duplication, and lower system quality as measured with the maintainability rating. Together, these correlations provide solid support for our statement that higher compression ratios are associated with lower quality software.

Thus, to answer H5: Higher compression ratios are associated with lower quality software.

To further provide support that our chosen measurement variables properly measure their underlying latent constructs as a group, we perform a factor analysis in the next section.

#### 9.9. Factor Analysis

Our theoretical model states that the change in average cyclomatic complexity, the change in duplication score and an increase in compression ratio are all proxies for the underlying latent variable undesirable growth. Similarly, we have stated that the NCD, the number of new methods and an increase in the number of lines of code are proxies for the underlying latent variable desirable growth. In previous sections, we have provided statistical support for the proposed relationships in our theoretical model based on individual correlations or regression analyses. In this section, we provide support that the measurement variables properly measure their underlying latent constructs as a group,
i.e. whether these variables “measure the same thing”. To test this, we use factor analysis, which is a commonly used method in the social sciences [51].

Table 9.10 shows measures of construct validity for the latent variables in the model.

<table>
<thead>
<tr>
<th>Latent variable</th>
<th>α</th>
<th>λ</th>
<th>AVE</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desirable growth</td>
<td>0.74</td>
<td>2.22</td>
<td>0.66</td>
<td>0.74</td>
</tr>
<tr>
<td>Undesirable growth</td>
<td>0.80</td>
<td>2.15</td>
<td>0.58</td>
<td>0.75</td>
</tr>
<tr>
<td>System size</td>
<td>0.97</td>
<td>2.83</td>
<td>0.92</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Table 9.10: Cronbach’s α, Eigenvalues (λ), Average Variance Explained (AVE) and Composite Reliability (CR) for all latent variables in the model.

The values in Table 9.10 show that the grouping of measurement variables into latent constructs is supported by the data. The latent variable desirable growth has a Cronbach’s α of 0.74 and an Eigenvalue of 2.22. The Cronbach’s α measures whether a group of metrics collectively measures the same concept, and a value between 0.7 and 0.9 is generally considered to be a good fit [84]. Additionally, desirable growth has an Eigenvalue of 2.22, which is also considered to be a good fit. The Kaiser criterion states that factors with Eigenvalues larger than 1 should be maintained [51]. Similarly, the Cronbach’s α for undesirable growth (0.80) and the Eigenvalue (2.15) also satisfy these criteria. The concept of system size has a very high Cronbach’s α of 0.97, indicating that the size metrics group very well together.

The Average Variance Explained (AVE) for each of the three latent variables are larger than 0.5, indicating that more than half of the variance in the latent variable is due to the measurement variables, and not due to the errors [42]. The Composite Reliability for each latent construct is larger than 0.70, as suggested by [51].

Table 9.11 shows factor loadings and uniqueness for the measurement variables in the model that are grouped into latent constructs.

<table>
<thead>
<tr>
<th>Latent variable</th>
<th>Measurement variable</th>
<th>Factor loading</th>
<th>Uniqueness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desirable growth</td>
<td>NCD</td>
<td>0.71</td>
<td>0.49</td>
</tr>
<tr>
<td></td>
<td># new methods</td>
<td>0.92</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>ln ΔLOC</td>
<td>0.93</td>
<td>0.13</td>
</tr>
<tr>
<td>Undesirable growth</td>
<td>ln Δavg McCabe</td>
<td>0.84</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>ln Δdup. score</td>
<td>0.88</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>ln Δcomp. ratio</td>
<td>0.82</td>
<td>0.33</td>
</tr>
<tr>
<td>System size</td>
<td>ln LOC</td>
<td>0.97</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>ln # methods</td>
<td>0.98</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>ln # files</td>
<td>0.96</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Table 9.11: Factor loadings, uniqueness and Cronbach’s α for latent and measurement variables. α = Cronbach’s alpha, λ = Eigenvalue, AVE = Average Variance Extracted, CR = Composite Reliability.

Factor loadings specify the “relative importance” of a certain measurement variable in a certain factor. There exist several guidelines for thresholds of factor loadings, but overall, values above 0.70 are considered to be acceptable [51]. The factor loading of
The uniqueness of each variable indicates the degree to which each variable varies uniquely as compared to the other variables in the construct. The greater the uniqueness, the lower the relevance of the variable in the factor model. Lower values for uniqueness are thus preferred from a standpoint of factor analysis [51]. The metrics for system size have a small degree of uniqueness, which is expected given the high values for Cronbach’s α (0.97) and the Eigenvalue of 2.83. NCD, on the other hand, has a larger degree of uniqueness as compared to the other variables in the construct desirable growth, which corresponds to its lower factor loading than the other variables.

We can thus conclude that the factor analysis of our measurement variables and latent constructs provides support for our grouping of measurement variables in the latent constructs system size, undesirable growth and desirable growth.

9.10. DISCUSSION
In this chapter, we have investigated the growth of software libraries and different factors influencing that growth. Our first hypothesis states that larger software libraries grow slower than small software libraries. This was proven to be the case for industrial software systems in Chapter 8, where larger systems tended to grow slower and very large systems even decrease in size. For software libraries, however, this does not turn out to be the case. In fact, quite the opposite was true: larger software libraries tend to grow faster than smaller ones. We assume that the discrepancy between our findings in Chapter 8 and in this chapter is caused by the fact that the analyzed libraries are relatively small compared to the industrial software systems that we have analyzed. The diminishing effect of size on growth is expected to only manifest itself after a certain size, which was 100,000 LOC in the case of industrial software systems.

The second hypothesis expressed our expectation that larger systems require more work but that their size inhibits the required higher productivity. We have found that indeed more work is performed on larger software libraries, but there was no proof found for an inhibiting effect of size on productivity. This indicates that, at least for software libraries, the amount of work that is performed increases linearly with the size of the library. Similar to the previous hypothesis, we hypothesize that a nonlinear relationship between work and system size was not found because the average size of software libraries is too small to detect a reduced productivity in larger libraries. Future work could further investigate whether this hypothesis is valid by measuring the NCD through time similar to the model in Chapter 8, and then finding out if the NCD decreases on average for a project.

The third hypothesis states that higher quality software makes developers more productive in terms of newly implemented functionality. This hypothesis was confirmed using two different analyses. First, we have demonstrated that less work is done on software systems of higher quality. As we have pointed out in Section 9.2, all metrics based on lines of code, including edit script size, have the problem that it is unclear whether a low value means that developers are limited in the amount of work they can do, or that they simply need to do less work to implement the same amount of functionality.

For this reason, another analysis was performed that demonstrated that the NCD
between two snapshots is generally larger in systems with higher quality code. This indicates that developers can implement larger amounts of functionality in systems of higher quality. Translating this result into time and monetary terms, this result suggests that higher quality software is expected to cost less to maintain than lower quality software. It also indicates that there does not seem to be a trade-off between quality and costs, but instead, higher quality software costs less to build and maintain than lower quality software.

The fourth hypothesis states that the NCD can be used to measure the amount of newly implemented functionality. Since the metric is based on solid foundations in information theory, we expect that NCD is a good proxy for new functionality. The measurements we have performed also indicate that edit script, changes in LOC and the number of new methods in a library correspond to changes in NCD. This provides support for our hypothesis that the NCD is a metric that can be used for this purpose.

A point of concern might be that we apply the NCD to source code instead of binary code. As was discussed in Chapter 7, future work could investigate whether using binary code instead of source files gives better NCD values. However, in order to make a valid comparison with applying NCD to source code, an investigation on the amount of new functionality present for given values of NCD has to be performed first. After this, the same analysis can be performed with binary code and NCD values.

The fifth hypothesis states that the compression ratio can be used to measure undesirable growth. We have shown that the compression ratio correlates with multiple different metrics that we use as an indicator for undesirable growth, such as the duplication score, the average cyclomatic complexity, and the maintainability rating. Similar to the NCD metric, the compression ratio metric is based on a solid rationale rooted in information theory. This rationale, combined with the provided evidence, indicates that the compression ratio (and differences in the compression ratio) can be used as an indicator for low software quality and undesirable growth.

### 9.10.1. Impact for Practitioners

Our analyses show that the concept of compression can be used to measure the amount of new functionality or changes in software quality. By maximizing the NCD, the amount of new functionality implemented in a system can be optimized. By minimizing the compression ratio, software quality can be optimized. The NCD and the compression ratio could be used by software developers in practice and provide an alternative to metrics that are currently in use, which are often based on lines of code. It is impossible to use lines of code to optimize the amount of newly implemented functionality, which was argued in this chapter, because an increase in LOC does not necessarily correspond to an increase in the amount of functionality a system provides.

We assume that software developers want to optimize the amount of functionality they implement in a given time interval. The best environment to optimize the NCD metric is a software system with a low compression ratio. The analyses we have performed in this chapter support this argument. This also implies that it pays off to invest in the quality of a software system to achieve a certain level of “functional” productivity.
9.11. Threats to Validity

In this chapter, we have introduced several new variables that measure different concepts related to software growth. A potential threat of construct validity is that these variables do not measure what was intended. An example of this is the increase in the duplication score and the increase in the average cyclomatic complexity to measure undesirable growth. We have chosen these variables because we expect that they measure two important aspects of undesirable software growth: an increase in the average complexity of methods in a software system and an increase in the amount of duplication in that system. This set of variables was not chosen to measure the concept of undesirable growth exhaustively. Other variables could also have been chosen, which would lead to a different definition of undesirable growth. In this chapter, we have chosen that undesirable growth is represented by these variables. Future work could investigate other metrics for undesirable growth.

There exists a possibility for selection bias in our dataset because we have only analyzed systems with complete data, which are 1,538 cases in most analyses. This means that the vast majority of systems in the Maven Dependency Dataset was not included in our analysis. It is unknown to what degree this leads to a bias in our results. Similarly, we have only investigated software libraries written in Java. However, due to the large size and the diversity of the Maven Dependency Dataset, we believe that the obtained results are not only representative of software development in open-source Java libraries in the specific subset we have analyzed, but also to software development in general. The observed relationships are of such a fundamental nature that we expect them to be present in other programming languages and repositories as well. Future work could test this expectation by replicating the same experiments in other languages and repositories.

We did not test the NCD metric against a software system with a known amount of function points. This would result in a table similar to Table 9.9 and would show the relationship between increases in NCD and increases in the amount of functionality as provided by a system. However, Table 9.9 contains a comparison between edit script size and NCD, which provides a similar validation assuming that edit script size and the amount of new functionality implemented correlate with each other. Table 9.2 shows that the correlation between these variables is 0.40. We assume that this correlation has a relatively low value because edit scripts also contain changes to systems which do not necessarily contain new functionality, such as refactorings. Future work could further investigate the relationship between NCD and newly implemented functionality.

9.12. Conclusion

We have introduced a new theoretical model that connects the concepts of system quality, desirable and undesirable growth, system size and work done. To make these concepts measurable, we have introduced several metrics. Compression was used to measure the functional distance between snapshots and to measure changes in software quality. We have demonstrated that higher quality software increases the amount of functionality a software developer can implement and reduces the amount of work needed to implement it. We have also introduced the NCD as measure for the amount of functionality that was implemented in a snapshot. The compression ratio can be used to
detect changes in software quality.
The contributions of our analysis are the following:

- A theoretical model that links the concepts of desirable growth, undesirable growth, system quality, system size and work done;
- A metric to measure changes in software quality;
- A metric to measure the amount of new functionality implemented in a system;
- A validation of our theoretical model using correlations and regression analyses.
Software libraries are frequently used in modern software development. As we have shown in this thesis, interfaces in these libraries change frequently and cause rework in client systems. We have investigated to what degree library developers take backward compatibility into account, what the impact of changing interfaces is and how ripple effects caused by these changes can be mitigated. In a broader context, we have investigated change in software systems in general by investigating desirable and undesirable software growth and we have used compression to measure the amount of functional growth of software systems.

10.1. Contributions

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

- The Maven Dependency Dataset and a database containing several metrics and measurements from the Maven repository (Chapter 2).
- An assessment of the current level of interface stability and backward compatibility in open-source libraries (Chapter 3).
- Four new metrics to assess the degree of stability of a library and its historical releases (Chapter 4).
- An investigation in the relationship between stability and encapsulation of software libraries (Chapter 5).
- A new method to assess the impact of changing interfaces by injecting changes in interfaces (Chapter 6).
- A software growth model for industrial software systems and open-source software libraries (Chapter 7).
- A software productivity estimation and comparison method using compression (Chapter 8).
• A structural model that relates the concepts of software growth, size, quality and work done with each other (Chapter 9).

10.2. **Answers to Research Questions**

**RQ1: How stable are software interfaces in practice?**

The answer to this research question was given in Chapter 2, where the adherence to the Semantic Versioning guideline was tested in the Maven repository. This shows that software interfaces turn out to be unstable in practice. Mechanisms to signal this instability to library users are not used properly, leaving users of these interfaces with little guidance on changes in these interfaces. Even though not all developers may be aware of the existence of this standard, it is assumed to encode rules that have been applied implicitly by the open-source community before the existence of the standard.

The Semantic versioning guideline prescribes that version strings should have the format “MAJOR.MINOR.PATCH”, where the major version number should be increased when backward-incompatible changes are made, the MINOR version number should be increased when backward-compatible features are implemented and the PATCH version number should be increased when backward-compatible bug fixes are implemented. Version strings are the primary mechanism to signal changes in the public interface of a library, but one-third of open-source libraries contain backward incompatible changes in releases which are marked non-major. Similarly, deprecation tags are not applied properly. Public methods that are marked as deprecated are never deleted afterwards, and sometimes public methods are deleted in the next release without first applying a deprecation tag.

This indicates that mechanisms currently available to signal backward compatibility and changes in interfaces are not applied properly. The adherence to the Semantic versioning standard appears to improve marginally, however, since the number of breaking changes in minor and patch releases slowly decreases over time.

**RQ2: How can we measure interface instability?**

We have introduced four new metrics to measure interface instability. The metrics look at the number of removed public methods (WRM), the amount of change in existing methods (CEM), the percentage of new methods added to public interfaces (PNM) and the ratio of change in old and new methods (RCNO). Together, these metrics can provide insight in the stability of a library. Because the metrics are weighed with the number of times a method is used in the Maven repository, methods that are more frequently used and thus have more impact on client systems get a higher weighting in the final rating for a library.

A higher value for the WRM metric indicates that a library has a higher number of removed methods from public interfaces. This is a signal of interface instability. A higher value for the CEM metric indicates that a large amount of change is made to implementations of existing methods, indicating that the behavior of existing methods may have
changed since the previous version of a library. A higher value for the PNM metric indicates that the interface of a library grows over time. Adding methods to an interface preserves backward compatibility, but the PNM metric is included in our metric suite because it can signal the growth speed of a library interface. When the RCNO metric has a value larger than 1, it indicates that more changes are being made in newly implemented methods than in existing methods. If the metric is smaller than 1, the system is in a state of maintenance.

The CEM and the RCNO metric take implementation changes into account in terms of the amount of changed code. These metrics thus do not measure interface instability directly. These metrics were created because the code of a library can be changed while keeping its interface constant, leading to a change in behavior of the library, which can also have impact on client systems using these libraries.

**RQ3: What is the impact of interface instability?**

To measure the impact of interface instability in a controlled environment, we have introduced a method to automatically inject changes from a library update in its old version and collect the compilation errors in client libraries. The number and distribution of these errors can provide library users with information on the amount of effort it will take to update to a new version of a library interface, since it is expected that both a larger number of compilation errors as well as a larger number of different places in which these errors appear cause an increase in the time it takes to fix them.

By applying this method to the Maven repository, we have found that there are 595,158 breaking changes in the Maven repository and that these cause a total of 11,139,014 compilation errors in client libraries when applying these changes one by one to the previous version of the library, for an average of 18.72 compilation errors per breaking change. On average, when the number of breaking changes increases by 1%, the number of compilation errors is expected to increase by 1.68%. These numbers show that breaking changes cause rework in terms of compilation errors that need to be fixed in client libraries.

We have also investigated characteristics of libraries that cause a large dispersion of compilation errors in client systems. As it turns out, larger libraries cause a larger dispersion of compilation errors in client systems. A possible explanation for this is that larger libraries tend to contain more functionality which is harder to localize in client systems.

To find out how large a typical library update is in terms of edit script size, we have analyzed the structure of an average library update and the relative size of 10 of the most frequently occurring breaking change types. The result of this analysis indicates that a method removal and a parameter type change are typically the largest types of updates performed.

**RQ4: How can interface instability be mitigated?**

By using the metrics as defined above, we have first investigated factors influencing interface instability in a single library, as measured by the WRM metric. As it turns out, interface instability of a library can be predicted to a large degree by the implementation
changes in that library, the size of that library, the percentual growth of its interface, the amount of encapsulation of outgoing dependencies and the number of outgoing dependencies. Change in existing methods, library size, the percentage of new methods and the number of outgoing dependencies are positively related to $WRM$, indicating that an increase in any of these factors is associated with an increase in interface instability as measured by $WRM$. The average outgoing isolation rating is negatively related to interface instability, indicating that a larger isolation rating of external dependencies reduces the instability of a library interface.

We have also investigated the relationship between encapsulation, ripple effects and interface instability between library dependencies. As it turns out, an increase in library instability in dependencies is associated with an increase in library instability in a client library. This effect can be offset by better isolation of dependencies in client systems. Thus, instability in interfaces in library dependencies causes a ripple effect in client libraries, which can be offset by better encapsulation of these dependencies. Furthermore, the amount of dampening of ripple effects coming from library dependencies also depends on the specific library dependency. This means that it may be easier to limit ripple effects coming from, for instance, JUnit, than the ripple effects coming from a library as the Spring framework. A possible explanation for this can be that certain dependencies contain functionality which can be more easily encapsulated in single places in a client system, while others might be more difficult to encapsulate because it contains more cross-cutting concerns.

$RQ5$: **How can we calculate the costs of interface instability?**

To translate the impact of interface instability into a measure in time or monetary terms, we have used the concepts of Kolmogorov complexity and compression to obtain a measure for the amount of information present in source code or a change. This method makes it possible to approximate the amount of functionality present in source code by removing all redundancy and (structural) duplication from source code. What remains is an estimate of the true information content of a software system. Additionally, the Normalized Compression Distance (NCD) can be used to approximate the amount of new functionality present in a new release of a library as compared to its previous release.

An example of the application of the NCD is to monitor performance of agile teams. The NCD can be used to find out what the functional distance of the software is as compared to the start of a sprint. It is a better indicator of performance than lines of code written since it more closely measures the actual amount of new functionality implemented. The advantage of this method as compared to manually counting function points or inspecting backlogs is that it is fully automated.

We have demonstrated that the compressed size of the source code of a software systems correlates to a large degree with function point density estimates from other software cost estimation methodologies. This demonstrates that compression can be used to calculate the costs of a software change. We have also presented a benchmark for hourly churn figures per developer for a number of widely used programming languages. This benchmark can be used to check whether the performance of a software development team is in line with the industry as a whole.
**RQ6: What factors influence software growth in libraries?**

On a more general level, changing interfaces are a manifestation of changing requirements of a software system. For this reason, we have presented a structural model that investigates the relationship between software growth, size, quality and work done. We have introduced the distinction between “undesirable” and “desirable” software growth, which can be quantified with compression. We have argued that the ultimate goal of developers should be to obtain long-term sustainable software growth. When developers are able to reach long-term sustainable software growth, the amount of new functionality that they can implement in any moment in time is optimized while current architecture is not degraded.

We started by investigating actual growth in software systems. As it turns out, software libraries in the Maven repository tend to grow with approximately 10% per year. Industrial software systems grow with 17 LOC per day on average, but this number depends to a large degree on the size of the software system. Systems between 50 and 100 KLOC grow the fastest with 124 LOC per day. After 100 KLOC, growth diminishes and after 500 KLOC, growth even becomes negative with -41 LOC per day, indicating that these systems tend to decrease in size. Possible explanations for this are that systems are simply finished after a certain size, or it becomes more difficult to add functionality to a bigger system. The analysis suggests that there exists an optimal system size with respect to growth which lies between 50 and 100 KLOC.

In the case of software libraries, we have found that more work is done on larger libraries but no diminishing effect of size on the amount of work done is found. The relationship between maintainability and work done is negative, indicating that less work needs to be performed on systems with higher quality code. Furthermore, an increase in system quality increases the amount of functionality that can be implemented in a library. Thus, higher quality code enables developers to implement more functionality with less work.

The relationship between NCD, work done and changes in lines of code demonstrate that NCD is a proper indicator of newly implemented functionality. Factor loadings of the concepts of undesirable and desirable growth indicate that the increase in the average cyclomatic complexity, an increase in the compression ratio and an increase in the duplication of a system are a proper indicator for the concept of undesirable growth. Similarly, the NCD, the number of new methods added to a library and the increase in lines of code are a good indicator for the concept of desirable growth.

**10.3. IMPACT AND VALIDITY**

In this thesis, we have shown that interface instability is a real problem for software developers and leads to a measurable amount of rework. This was demonstrated using the Maven repository as dataset. Furthermore, we have demonstrated that it is possible to estimate the amount of work that needs to be performed in order to implement a change of a certain size.

There are, however, several limitations to our study and results. First, we have only investigated third-party, open-source Java libraries available in the Maven repository.
Although this is a large repository and it is frequently used by software engineers, it is still only a subset of the actual libraries software developers use in their software projects. Libraries other than those written in Java, libraries which are not open-source, libraries which are never published to an external repository (which are thus not “third-party”), and software systems which are not libraries are outside of the scope of this analysis.

To what degree our findings generalize to languages other than Java is unknown, but we expect that the underlying phenomena that are studied in this thesis can be found in other languages as well. An interface is a concept that the majority of programming languages support. In fact, the principle is so general that it can also be found in areas outside of software engineering, as mentioned in the introduction of this thesis. For example, a large building on which several different teams work has standardized interfaces, such as standardized electricity outlets, standardized wall sizes, and a layout on which all teams have agreed. If any of the information in these interfaces would change, the impact on other teams working on that building would be large.

In software engineering, the impact that interface instability has may take on different forms depending on the constructs available to mitigate this instability in a certain language. For instance, Java offers the opportunity to make methods private, but does not have a way to make complete packages private. Thus, there is no way to encapsulate changes in complete packages.

We expect that the same principles apply for libraries which are not open-source, since these libraries typically also have a published interface which other software developers can use. Software systems which are not libraries are also expected to suffer from the same problems as investigated in this thesis, as long as they have an interface which is used by other people than the ones who created the interface.

Our investigation on cost estimation using compression uses a large dataset of industrial software systems written in a large set of programming languages, which increases the generalizability of our results. However, the software still represents a subset of industrial, typically administrative, software systems. The productivity figures we have obtained may not apply to substantially different types of systems such as, for example, games, operating systems, hardware drivers and client-side browser code. Other benchmarks may have to be established to apply the same methods in these industries.

Our cost estimation methodology is based on large-scale averages of compression ratios for different languages. As was stated before, differences in company, culture and country may need to be taken into account before the results can be translated to other contexts. These differences were not taken into account in our investigation, for a number of reasons. First, we are only interested in large-scale averages and we assume that we can use these averages to obtain reasonable estimates with. Second, we have not investigated the programming style or productivity of individual programmers since we have limited the scope of our analysis to automated techniques, and we can therefore only apply the obtained averages instead of a more specific programmer-specific number. Third, we have demonstrated the validity of our method in this thesis, and we have provided the formulas for performing the compression-based calculations. When desired, numbers specific to companies or programmers can be used as input for these formulas, which will lead to a more specific estimation. This was, however, outside the scope of this thesis.
10.4. Lessons for Practitioners

Our research leads to several insights and recommendations for practitioners such as software engineers and architects. We have shown that backward incompatibilities are common in the open-source community. On one hand, this is understandable since it may be almost impossible to design a library from the start in such way that its public interface never needs to change, but changing it after it is being used by others creates rework for these developers. Developers should therefore be more aware of the impact of changing interfaces on systems using these interfaces. Mechanisms to signal interface instability should be used more frequently, given the large amount of breaking changes in the Maven repository. Deprecation tags should be applied correctly to methods, and deprecated methods should be thrown away after a number of releases.

We have demonstrated that changes in interfaces have measurable impact on client systems. From our analyses, a couple of design principles can be distilled. As is already widely known in the software engineering community and as we have demonstrated in this thesis as well, encapsulation is an effective way to prevent ripple effects from changes in software. Reducing the number of places where external code is used in a system can reduce the impact of changes in this code. This is expected to be true not just for library dependencies but for code in general as well. The amount of changes that need to be made to code can be reduced when the number of external dependencies of that code are reduced to a minimum. Thus, ideally, any piece of code should only do what it needs to do without any external dependencies. A design in which this principle is incorporated properly is expected to have a reduced overall long-run maintenance burden.

Our analysis on software growth in industrial systems of different sizes show that systems bigger than 100 KLOC generally have a diminished growth speed. We assume that a higher growth speed is desirable since it means that new features can be implemented faster. The “diminishing returns on scale” in software systems can be interpreted as an indication that systems bigger than a certain size should be split up into smaller parts or even multiple systems, which is again a widely known software engineering principle.

The application of the theory of compression and Kolmogorov complexity to source code implies that there may be multiple ways to implement the same feature in different programming languages, but in the end, programming language constructs can be abstracted away and source code can be reduced to the true information content of a software system. Programming languages may be more or less suitable for certain tasks, and the resulting code can be more or less verbose, but ultimately, writing software is just a means to implement desired functionality, regardless of the form it takes.

Since most software will be written once but maintained multiple times, possibly by other developers, it is important that developers choose the right level of abstraction and expressiveness to clearly communicate intent to developers maintaining that code. When writing down code that is either too verbose or not verbose enough, this is expected to lead to a larger maintenance burden for maintenance developers, thus making the system more expensive to maintain in the long run. To achieve long-term sustainable software growth, developers should therefore choose a programming language and level of abstraction that is most capable of expressing their intention to other developers.
10.5. **Future Work**

The investigations in this thesis have left several areas to future work, which are identified below.

### 10.5.1. Developer Behavior

In this thesis, we have assumed that anything that does not have direct effect on the source code of a system or which cannot be directly measured in that source code is outside the scope of this thesis. This does not mean, however, that inspecting developer behavior with regards to breaking changes cannot deliver valuable insights. Future work could try to understand why developers introduce breaking changes in their software. Or, more generally, why do developers apply changes to existing code? What are the decisions made by developers to change an existing piece of code? It could be that because of bad design of the system, the applied change would have been unnecessary if the system had been better designed from the start. Why did a developer choose to remove a certain method from an interface, and why did he add it in the first place? Understanding these design decisions of a developer could help in understanding the mechanisms involved in interface instability, and in a broader sense, changing software in general.

### 10.5.2. Change Injection

The controlled experiment that we have performed when injecting changes into source code could be further refined by also inspecting the effect of encapsulation on ripple effects from breaking changes. The experimental setup could be extended in such way that the system is automatically able to detect certain “encapsulation patterns”, and see to what degree these help to prevent the mitigation of ripple effects.

The manipulation of source code by changing its evaluated structure (the abstract syntax tree) has the potential to provide insight by answering several “what-if” scenarios. For example, what is the impact if a certain interface is completely removed, or moved to another package? What is the impact of applying a certain patch to a system on other parts of that system? This could go further than simply counting compilation errors. Unit tests could automatically be executed, source code adapted to include certain changes, and source code could then be re-executed to see what goes wrong.

### 10.5.3. Applying Compression

We have only scratched the surface of applying compression to source code. Our results give an indication that compression can be a useful tool that can be used in several different metrics which can help software engineers to build better quality software in practice. To better understand the behavior of compression-related metrics in practice, more research is needed.

We expect that applying compression to binary code instead of source code leads to more reliable results since the used “language” in an executable file is the same (for instance, bytecode). Since the Java bytecode compiler applies optimizations to source code, we expect that this might give better results when using our compression-based metrics, since differences in verbosity between programmers have already been optimized away to a large degree by the compiler. There are also compiler optimizations
which increase the amount of code produced, such as function inlining or loop unfolding, but we expect that differences between developers nevertheless become smaller in bytecode when the same compiler is used. In this thesis, we have not applied compression to binary code because for a lot of programming languages in our industrial dataset, only source code was available.

To gather more insight in the behavior of compression-related metrics, the same piece of functionality could be implemented in different programming languages, compressed and then compared to each other. This results in accurate compression ratios for languages for a given piece of functionality. In this thesis, however, we have relied on averages across a benchmark to obtain a compression ratio for a certain language. Because a large number of different systems and companies are represented in a single language, we have assumed that differences in programming style or verbosity cancel each other out. What remains are the differences between programming languages. In future work, actual differences between programming languages with respect to compression-related metrics could be further investigated.

With respect to the Normalized Compression Distance metric, the amount of newly implemented functionality as reported by an agile team or as counted by function points could be compared to the NCD metric. This would provide support for the usage of the NCD metric in practice. We believe, however, that the theory underlying the NCD metric is solid and that the metric could already be used in practice directly.


A

MAVEN DEPENDENCY DATASET
INSTALLATION INSTRUCTIONS

This chapter contains information on the Maven Dependency Dataset and how to install and use it.

A.1. DATASET EXPLANATION

A.1.1. MySQL database
Tables A.1 to A.9 show tables and columns present in our MySQL database.

A.1.2. Neo4j database
The Neo4j database consists of a collection of tuples of the following form:

\(<\text{unitId}_1, \text{unitId}_2, \text{type}>\)

where \(\text{unitId}_1\) is a 64-bit integer referring to an object in the Berkeley DB database and \(\text{unitId}_2\) is a 64-bit integer referring to another object in the Berkeley DB database. These two objects can have one of four types of relationships, which is stored as an integer in \(\text{type}\) and is one of the following:

1. Next version
   \(\text{unitId}_1\) is the next version of \(\text{unitId}_2\). For instance, when two methods are present in two library versions, one \(\text{unitId}_1\) would point to the method in the first version and the other \(\text{unitId}_2\) would point to the method in the second version of the library.

2. Extends/Implements
   \(\text{unitId}_1\) extends or implements \(\text{unitId}_2\). When there is an extends/implements relationship, both identifiers refer to classes or interfaces.

3. Contains
   \(\text{unitId}_1\) contains \(\text{unitId}_2\). Containment can have different meanings depending on
the types of units referred to. For instance, a class can contain a method. If this is the case, unitId1 refers to a class and unitId2 refers to a method. Other types of containment are a class that contains another class or a package that contains a class.

4. **Calls**

unitId1 calls unitId2, which are both methods.

As an example of queries that can be answered using the Neo4j database, consider the following examples:

Count the type of relationships present in the database:

```sql
START n=node(*)
MATCH n-[r]-m
RETURN type(r), count(*);
```

### A.1.3. **Berkeley DB Database**

The Berkeley DB database can be used to obtain information on specific methods, classes and packages in the repository. The script ./getunits.sh can be used to extract information from this database. Below is an example of a query that can be executed:

```
./getunits.sh
-j <fullmaven.jar path>
-b <Berkeley DB path>
-g "tv.bodil"
-n "tv.bodil.testlol.Testlol.startTimer()"
-v "1.2.2"
```

The results of this query are as follows:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>unitId</td>
<td>135108785976600</td>
</tr>
<tr>
<td>name</td>
<td>tv.bodil.testlol.Testlol.startTimer()</td>
</tr>
<tr>
<td>groupId</td>
<td>tv.bodil</td>
</tr>
<tr>
<td>artifactId</td>
<td>maven-testlol-plugin</td>
</tr>
<tr>
<td>version</td>
<td>1.2.2</td>
</tr>
<tr>
<td>unitType</td>
<td>5 (method)</td>
</tr>
<tr>
<td>fileId</td>
<td>12</td>
</tr>
<tr>
<td>snapshotId</td>
<td>3</td>
</tr>
<tr>
<td>next version</td>
<td>-</td>
</tr>
<tr>
<td>LOC</td>
<td>2</td>
</tr>
<tr>
<td>McCabe</td>
<td>1</td>
</tr>
<tr>
<td>nrParams</td>
<td>0</td>
</tr>
<tr>
<td>usageCount</td>
<td>9</td>
</tr>
</tbody>
</table>

To get help using this script, type ./getunits.sh -h.
### Table A.1: Columns in the files table, as stored in MySQL

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileId</td>
<td>The unique file ID of the file in this table.</td>
</tr>
<tr>
<td>fullName</td>
<td>The relative path to the file on disk.</td>
</tr>
<tr>
<td>groupId</td>
<td>The groupId of the artifact.</td>
</tr>
<tr>
<td>artifactId</td>
<td>The artifactId of the artifact.</td>
</tr>
<tr>
<td>version</td>
<td>The version of the artifact.</td>
</tr>
<tr>
<td>reservedNodeId</td>
<td>The node at which code was processed on the Supercomputer. Ranges from 1 to 60.</td>
</tr>
<tr>
<td>snapshotId</td>
<td>The version number when ordering all library versions from first to latest. Starts with 1.</td>
</tr>
<tr>
<td>hasSource</td>
<td>Whether the binary jar has a source jar of the same name in the same directory.</td>
</tr>
<tr>
<td>PageRank</td>
<td>(Network metric) The PageRank of the library when representing libraries and their dependencies as a graph.</td>
</tr>
<tr>
<td>Betweenness</td>
<td>(Network metric) The betweenness of the library (see [45]).</td>
</tr>
<tr>
<td>Hubbiness</td>
<td>(Network metric) The hubbiness of the library (HITS-algorithm, see [65]).</td>
</tr>
<tr>
<td>Authoritativeness</td>
<td>(Network metric) The authoritativeness of the library (HITS-algorithm, see [65]).</td>
</tr>
<tr>
<td>WRM</td>
<td>The Weighted number of Removed Methods compared to the previous version of this library (see [89]).</td>
</tr>
<tr>
<td>CEM</td>
<td>The Change in Existing Methods compared to the previous version of this library (see [89]).</td>
</tr>
<tr>
<td>RCNO</td>
<td>The Ratio of Change of New and Old methods measured with McCabe differences compared to the previous version of this library (see [89]).</td>
</tr>
<tr>
<td>PNM</td>
<td>The Percentage of New Methods compared to the previous version of this library (see [89]).</td>
</tr>
<tr>
<td>nrUnits</td>
<td>The total number of methods in this library version.</td>
</tr>
<tr>
<td>nrNewUnits</td>
<td>The number of new methods compared to the previous version of this library.</td>
</tr>
<tr>
<td>nrOldUnits</td>
<td>The number of methods that are both in this and the previous version of this library.</td>
</tr>
<tr>
<td>nrRemovedUnits</td>
<td>The number of methods that have been removed from the library compared to the last version of this library.</td>
</tr>
<tr>
<td>deltaUn</td>
<td>The sum of McCabe values in newly added methods as compared to the previous version.</td>
</tr>
<tr>
<td>deltaUo</td>
<td>The difference in McCabe values of existing methods as compared to the previous version.</td>
</tr>
<tr>
<td>hws</td>
<td>The weight of this library version as used in summing metric differences over all versions of a library (see [89]).</td>
</tr>
<tr>
<td>maintainability</td>
<td>The SIG Maintainability rating of a library (see [53]).</td>
</tr>
<tr>
<td>CRS</td>
<td>The Commonality Rating of a System as defined in [90].</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>RL</td>
<td>The Rating of a Library (indegree, see [90]).</td>
</tr>
<tr>
<td>updated</td>
<td>The date that this library version was uploaded to the central repository.</td>
</tr>
<tr>
<td>status</td>
<td>Whether the library version has been processed by the Supercomputer.</td>
</tr>
<tr>
<td>enabled</td>
<td>Whether the file should be processed. Is false when hasSource = 0 or when there are other reasons this file should be excluded from analysis.</td>
</tr>
<tr>
<td>packagePrefix</td>
<td>The “greatest common denominator” of package prefixes as found in the library when scanning for package statements. When multiple package prefixes have been found they are separated with a comma. Is used to detect dependencies in other files since these are then imported with import statements.</td>
</tr>
</tbody>
</table>

Table A.2: Columns in the `files` table, as stored in MySQL (continued).

<table>
<thead>
<tr>
<th>deps</th>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callId</td>
<td>The unique ID of the dependency as stored in this table.</td>
<td></td>
</tr>
<tr>
<td>fromFileId</td>
<td>The library that specified the dependency.</td>
<td></td>
</tr>
<tr>
<td>toFileId</td>
<td>The library that <code>fromFileId</code> depends upon.</td>
<td></td>
</tr>
<tr>
<td>isolation</td>
<td>The percentage of files in <code>fromFileId</code> that contains an import statement starting with the <code>packagePrefix</code> of <code>toFileId</code>. For this <code>packagePrefix</code>, see the <code>files</code> table.</td>
<td></td>
</tr>
</tbody>
</table>

Table A.3: Columns in the `deps` table, as stored in MySQL.
Table A.4: Columns in the *units* table, as stored in Berkeley DB.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unitId</td>
<td>The unique unit identifier as stored in this table.</td>
</tr>
<tr>
<td>name</td>
<td>The fully qualified name of the unit.</td>
</tr>
<tr>
<td>groupId</td>
<td>The groupId of the library this unit belongs to.</td>
</tr>
<tr>
<td>artifactId</td>
<td>The artifactId of the library this unit belongs to.</td>
</tr>
<tr>
<td>version</td>
<td>The version of the library this unit belongs to.</td>
</tr>
<tr>
<td>unitType</td>
<td>The unit type of this unit (1 = jar file, 2 = package, 3 = java file, 4 = class, 5 = method)</td>
</tr>
<tr>
<td>fileId</td>
<td>The fileId this unit belongs to.</td>
</tr>
<tr>
<td>snapshotId</td>
<td>The snapshot number of this unit.</td>
</tr>
<tr>
<td>nextVersion</td>
<td>The <em>unitId</em> of the next version of this unit.</td>
</tr>
<tr>
<td>LOC</td>
<td>The number of lines of source code for this unit.</td>
</tr>
<tr>
<td>McCabe</td>
<td>McCabe value for this unit (only when unitType = 5).</td>
</tr>
<tr>
<td>nrParams</td>
<td>The number of parameters of this unit (only when unitType = 5)</td>
</tr>
<tr>
<td>usageCount</td>
<td>The number of times this unit is being used in the repository (only when unitType = 5)</td>
</tr>
<tr>
<td>Column name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>changeId</td>
<td>The unique change ID as stored in this table.</td>
</tr>
<tr>
<td>changeType</td>
<td>The type of change as determined by Clirr. For an overview of change types, see changeTypes.</td>
</tr>
<tr>
<td>fileIdv1</td>
<td>The fileId of the first file involved in the change.</td>
</tr>
<tr>
<td>fileIdv2</td>
<td>The fileId of the second file involved in the change.</td>
</tr>
<tr>
<td>packageUnitIdv1</td>
<td>the Berkeley DB unitId of the first version of the package involved in the change. Can be null when the change does not involve a package.</td>
</tr>
<tr>
<td>packageUnitIdv2</td>
<td>the Berkeley DB unitId of the second version of the package involved in the change. Can be null when the change does not involve a package.</td>
</tr>
<tr>
<td>methodUnitIdv1</td>
<td>the Berkeley DB unitId of the first version of the method involved in the change. Can be null when the change does not involve a method.</td>
</tr>
<tr>
<td>methodUnitIdv2</td>
<td>the Berkeley DB unitId of the second version of the method involved in the change. Can be null when the change does not involve a method.</td>
</tr>
<tr>
<td>classUnitIdv1</td>
<td>the Berkeley DB unitId of the first version of the class involved in the change. Can be null when the change does not involve a class.</td>
</tr>
<tr>
<td>classUnitIdv2</td>
<td>the Berkeley DB unitId of the second version of the class involved in the change. Can be null when the change does not involve a class.</td>
</tr>
<tr>
<td>fieldUnitIdv1</td>
<td>the Berkeley DB unitId of the first version of the field involved in the change. Can be null when the change does not involve a field.</td>
</tr>
<tr>
<td>fieldUnitIdv2</td>
<td>the Berkeley DB unitId of the second version of the field involved in the change. Can be null when the change does not involve a field.</td>
</tr>
</tbody>
</table>

Table A.5: Columns in the changes table, as stored in MySQL.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>changeTypeId</td>
<td>The unique change type ID as stored in this table.</td>
</tr>
<tr>
<td>description</td>
<td>A description of the type of change.</td>
</tr>
<tr>
<td>breaking</td>
<td>Whether the change is breaking, i.e. whether it causes a binary incompatibility in systems using it and which thus have to be recompiled.</td>
</tr>
</tbody>
</table>

Table A.6: Columns in the changeTypes table, as stored in MySQL.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unitTypeId</td>
<td>The unique unit type ID as stored in this table.</td>
</tr>
<tr>
<td>parentType</td>
<td>The unitTypeId of the parent of this unitTypeId.</td>
</tr>
<tr>
<td>description</td>
<td>A description of the unit type.</td>
</tr>
</tbody>
</table>

Table A.7: Columns in the unitTypes table, as stored in MySQL.
### A.1. Dataset Explanation

#### depTypes

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>depTypeId</td>
<td>The unique dependency type ID as stored in this table.</td>
</tr>
<tr>
<td>description</td>
<td>A description for this type of dependency.</td>
</tr>
</tbody>
</table>

Table A.8: Columns in the `depTypes` table, as stored in MySQL.

#### stats

<table>
<thead>
<tr>
<th>Column name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileId</td>
<td>The fileId to which the statistics belong.</td>
</tr>
<tr>
<td>vol</td>
<td>The SIG star rating for volume on a 0.5 - 5.5 scale. 5% of systems has a score between 0.5 and 1.5, 30% has a score between 1.5 and 2.5, 30% has a score between 2.5 and 3.5, 30% has a score between 3.5 and 4.5 and 5% has score between 4.5 and 5.5.</td>
</tr>
<tr>
<td>dup</td>
<td>The star rating for duplication on a 0.5 - 5.5 scale.</td>
</tr>
<tr>
<td>us</td>
<td>The star rating for unit size (lines of code per method) on a 0.5 - 5.5 scale.</td>
</tr>
<tr>
<td>uc</td>
<td>The rating for unit complexity (McCabe) (0.5 - 5.5).</td>
</tr>
<tr>
<td>ui</td>
<td>The star rating for unit interfacing (number of parameters per method) (0.5 - 5.5).</td>
</tr>
<tr>
<td>mc</td>
<td>The star rating for module coupling (number of incoming dependencies per file) (0.5 - 5.5).</td>
</tr>
<tr>
<td>cb</td>
<td>The star rating for component balance (0.5 - 5.5).</td>
</tr>
<tr>
<td>ci</td>
<td>The star rating for component independence (0.5 - 5.5).</td>
</tr>
<tr>
<td>nm</td>
<td>The number of methods in the system.</td>
</tr>
<tr>
<td>nc</td>
<td>The number of classes in the system.</td>
</tr>
<tr>
<td>np</td>
<td>The number of packages in the system.</td>
</tr>
<tr>
<td>loc</td>
<td>The number of source lines of code in the system.</td>
</tr>
</tbody>
</table>

Table A.9: Columns in the `stats` table, as stored in MySQL.
<table>
<thead>
<tr>
<th>Clirr type</th>
<th>Description</th>
<th>Binary compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Increased visibility of class</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>1001</td>
<td>Decreased visibility of class</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>2000</td>
<td>Changed from class to interface</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>2001</td>
<td>Changed from interface to class</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>3001</td>
<td>Removed final modifier</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>3002</td>
<td>Added final modifier to class, but class was effectively final anyway</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>3003</td>
<td>Added final modifier</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>3004</td>
<td>Removed abstract modifier</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>3005</td>
<td>Added abstract modifier</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>4000</td>
<td>Added to the set of implemented interfaces</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>4001</td>
<td>Removed from the set of implemented interfaces</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>5000</td>
<td>Added to the list of superclasses</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>5001</td>
<td>Removed from the list of superclasses</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>6000</td>
<td>Added field</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>6001</td>
<td>Removed field</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>6002</td>
<td>Value of field is no longer a compile-time constant</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>6003</td>
<td>Value of compile-time constant has been changed</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>6004</td>
<td>Changed type of field</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>6005</td>
<td>Field is now non-final</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>6006</td>
<td>Field is now final</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>6007</td>
<td>Field is now non-static</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>6008</td>
<td>Field is now static</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>6009</td>
<td>Accessibility of field has been increased</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>6010</td>
<td>Accessibility of field has been weakened</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>6011</td>
<td>Field has been removed, but it was previously a constant</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>7000</td>
<td>Method now implemented in superclass</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>7001</td>
<td>Abstract method is now specified by implemented interface</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>7002</td>
<td>Method has been removed</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>7003</td>
<td>Method has been removed, but an inherited definition exists</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>7004</td>
<td>Number of arguments changed</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>7005</td>
<td>Parameter has changed its type</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>7006</td>
<td>Return type of method has been changed</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>7007</td>
<td>Method has been deprecated</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>7008</td>
<td>Method is no longer deprecated</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>7009</td>
<td>Accessibility of method has been decreased</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>7010</td>
<td>Accessibility of method has been increased</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>7011</td>
<td>Method has been added</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>7012</td>
<td>Method has been added to an interface</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>7013</td>
<td>Abstract method has been added</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>7014</td>
<td>Method is now final</td>
<td>Breaks compatibility</td>
</tr>
<tr>
<td>7015</td>
<td>Method is no longer final</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>8000</td>
<td>Class added</td>
<td>Binary compatible</td>
</tr>
<tr>
<td>8001</td>
<td>Class removed</td>
<td>Breaks compatibility</td>
</tr>
</tbody>
</table>

Table A.10: Detected binary compatibilities and incompatibilities by Clirr.
CURRICULUM VITAE

PERSONAL DATA
Full name: Steven Bernardus Alexander Raemaekers
Date of birth: August 1st, 1984
Place of birth: Eindhoven

WORK EXPERIENCE
September 2014 - present
Platform architect streaming analytics at ING, Amsterdam.

April 2011 - August 2014
Researcher at the Software Improvement Group, Amsterdam.

July 2010 - March 2011
Analyst at Avanade, Almere.

September 2008 - August 2009
Junior Software Developer (part time) at TRAG Performance Intelligence Group, Zeist.

EDUCATION
April 2011 - April 2015
PhD student at Delft University of Technology.

September 2005 - September 2007
Master student at University of Amsterdam, Master program “Software Engineering”.

September 2006 - June 2009
Bachelor student at University of Amsterdam, Minor program “Medical Informatics”.

September 2002 - August 2006
Medicine at Vrije Universiteit, Amsterdam.

September 1996 - June 2002
High school (VWO, “Natuur en Gezondheid”) at “Pleincollege Eckart”, Eindhoven.
Third-party libraries are used frequently in modern software development. A large number of websites, industrial software and open-source programs use third-party libraries frequently. These libraries have the advantage that they save development time and effort for the users of the library. Additionally, these libraries are often tested extensively and used in practice by other software developers.

Besides these advantages, externally developed software such as third-party libraries also has a number of disadvantages. One of these disadvantages is the potential instability of the public interface on which client systems rely. When the public interface of a library changes, all places in client systems which use this interface must be changed as well.

One mechanism for library developers to signal interface instability is through the use of version numbers and deprecation tags. Conventions have been established on the proper use of version strings and how to update a version of a library given changes in that library. Semantic versioning is an example of such a convention, which states that the major version of a library should be increased when backward-incompatible changes are made, the minor version should be increased when backward-compatible functionality has been added and the patch version should be increased when backward-compatible bug fixes have been made.

In practice, however, properly formatted version strings are often used but without the underlying semantics. Often, backward-incompatible API changes are included in non-major releases. Deprecation tags are another mechanism to provide signals about methods that are expected to be deleted in future versions. However, these tags are also often not used properly, leaving library users with little guidance on the stability of public interfaces.

The impact of changing interfaces, whether signaled properly through version strings or not, is measurable in software that uses these interfaces. If a new library version is released that contains a backward-incompatible change and a client system uses the changed functionality, a change in the client system will be required before the new version of the library can be used. These ripple effects caused by changes in a public inter-
face are quantifiable and are significant. One commonly used mechanism in software engineering to localize the effect of changes in code is encapsulation. Proper design of classes with encapsulation can prevent ripple effects of changes in third-party libraries.

In practice, instead of running extensive analyses on the differences between two versions of a library interface, developers of client systems often simply recompile their code and find out where compilation errors appear. This approach can be used to determine the impact of changing library interfaces. Each change in library interfaces can be injected in the old version of that interface and the compilation errors caused by this change can be captured. This leads to an overview of all code that need to be changed because of a change in an interface.

The evolution of libraries can also be measured in other ways than inspection of their interface. Changes in the implementation of a library may not have a direct impact on client systems in terms of compilation errors that need to be fixed, but can have a large impact on the behavior of a system. To make changes in behavior or functionality measurable, we use compression to measure the functional distance between two versions of a library. The more new functionality is added in a library, the larger the functional distance between two versions is. It can be demonstrated that software written in other programming languages has different compression characteristics, and that these differences correspond to the expressiveness and verbosity of a programming language.

Besides interface instability, software growth is another manifestation of software evolution. Software growth can be divided into desirable and undesirable growth. Desirable growth is an increase in the amount of unique functionality provided by a software system. Undesirable growth is an increase in the size of a system without an increase in the amount of unique functionality as provided by that system. By using compression, the amount of new functionality present in a new version of a system can be measured. These concepts can be used to demonstrate that higher quality software increases the amount of functionality a software developer can implement and reduces the amount of work needed to implement it. This means that higher quality software is less expensive to develop. In order to achieve long-term sustainable software growth, developers should aim to maximize desirable growth and minimize undesirable growth.
Een groot aantal websites en open-source systemen maken gebruik van softwarebibliotheeken. Deze bibliotheken hebben het voordeel dat ze de ontwikkelaar van een systeem tijd en geld besparen. Ook zijn deze bibliotheken vaak uitgebreid getest en worden ze vaak al in de praktijk gebruikt door andere developers.

Afgezien van deze voordelen heeft het includeren van extern ontwikkelde software ook een aantal nadenlen. Een van deze nadenlen is de mogelijke instabiliteit van de publieke interface waar gebruikers van een bibliotheek gebruik van maken. Wanneer de publieke interface van een bibliotheek verandert dan worden alle gebruikers van deze bibliotheek gedwongen om hun code aan te passen wanneer zij gebruik maken van deze interface.

Developers van een bibliotheek kunnen een signaal afgeven aan gebruikers dat hun interface instabiel is en dat deze in een nieuwe versie is veranderd. Dit kan onder andere door middel van versienummers. Er bestaan conventies voor het juiste gebruik van versienummers en het ophogen van dit versienummer gegeven veranderingen in een nieuwe release. Semantic versioning is een voorbeeld van een dergelijke conventie. Semantic versioning stelt dat de major versie van een bibliotheek moet worden opgehoogd wanneer er veranderingen zijn doorgevoerd die breken met de vorige versie van de interface, dat de minor versie moet worden opgehoogd wanneer er veranderingen zijn doorgevoerd die niet breken met de vorige versie van de interface en dat de patch versie moet worden opgehoogd wanneer er enkel bug fixes zijn uitgevoerd.

In de praktijk blijkt echter vaak dat het formaat van de versienummers juist is maar dat de onderliggende semantiek ontbreekt. Vaak worden er veranderingen doorgevoerd die breken met de vorige versie van de interface terwijl het major versienummer niet wordt opgehoogd. Deprecation tags zijn een ander mechanisme om een signaal over de continuïteit van de interface af te geven. Een deprecation tag wordt toegevoegd aan een methode waarvan verwacht wordt dat hij wordt verwijderd in de volgende major versie van de bibliotheek. Deze deprecation tags worden echter vaak onjuist gebruikt waardoor gebruikers van een bibliotheek weinig aanwijzingen hebben over de stabilititeit van een interface.
De impact van veranderingen in interfaces, al dan niet juist gecommuniceerd door middel van versienummers, kan meetbaar worden gemaakt in systemen die van deze interfaces gebruik maken. Wanneer er een nieuwe versie van een bibliotheek is ontwikkeld die veranderingen bevat die breken met de vorige versie van de interface, dan zal er een verandering nodig zijn in alle systemen die van deze bibliotheek gebruik maken. Deze ripple effects zijn kwantificeerbaar en blijken een significante hoeveelheid werk te veroorzaken voor developers. Een manier om de hoeveelheid werk door ripple effects te beperken is het gebruik van encapsulatie, wat er voor zorgt dat veranderingen gelokaliseerd worden. Een goed ontwerp van klassen met encapsulatie kan ripple effects door veranderingen in interfaces van bibliotheken voorkomen.

In de praktijk zullen de meeste developers geen uitgebreide analyses doen naar de verschillen tussen twee versies van een bibliotheek, maar simpelweg de nieuwe versie inclueren, de code hercompileren en kijken waar er compilatiefouten ontstaan. Deze aanpak kan ook worden gebruikt om de impact van veranderingen in de interface van een bibliotheek te bepalen. Individuele wijzigingen in een interface kunnen worden geïnjecteerd in de oude versie van een bibliotheek en de ontstane compilatiefouten door deze verandering kunnen worden opgeslagen. Hierdoor kan een overzicht van alle code die moet worden gewijzigd door een verandering in een interface worden verkregen.

De evolutie van bibliotheken kan ook op andere manieren worden gemeten dan enkel via inspectie van hun interface. Veranderingen in de implementatie van een bibliotheek hebben wellicht niet direct impact op systemen die gebruik maken van deze interface maar kunnen het gedrag van een bibliotheek wel drastisch wijzigen. Om veranderingen in gedrag of functionaliteit meetbaar te maken kan compressie worden gebruikt, waarmee de functionele afstand tussen twee versies van een bibliotheek kan worden bepaald. Hoe meer nieuwe functionaliteit aan een nieuwe versie van een bibliotheek is toegevoegd, hoe groter de functionele afstand zal zijn. Verschillende programmeertalen bezitten verschillende compressie-eigenschappen. Deze verschillen corresponderen met verschillen in expressiviteit en verbositeit van programmeertalen.

De evolutie van software manifesteert zich naast instabiliteit van interfaces ook door de groei van software. Groei van software kan worden onderverdeeld in ongewenste en gewenste groei. Gewenste groei is een toename van de hoeveelheid unieke functionaliteit die een systeem bevat. Ongewenste groei is een toename in de grootte van software zonder toename in de hoeveelheid unieke functionaliteit in een systeem. Door middel van compressie kan de hoeveelheid nieuwe functionaliteit in een nieuwe versie van een systeem vergeleken met de oude versie worden gemeten. Compressie kan worden gebruikt om aan te tonen dat developers meer functionaliteit kunnen opleveren met software van hogere kwaliteit, terwijl de hoeveelheid code die hiervoor nodig is juist afneemt. Dit betekent dat software van hogere kwaliteit goedkoper is om te ontwikkelen. Om duurzame groei op de lange termijn te bereiken zouden developers ongewenste groei van hun systeem moeten minimaliseren en gewenste groei maximaliseren.