Testing principles, current practices, and effects of change localization

Raemaekers, SBA; Nane, GF; van Deursen, A; Visser, J

DOI
10.1109/MSR.2013.6624037

Publication date
2013

Document Version
Accepted author manuscript

Published in
Proceedings - 10th Working Conference on Mining Software Repositories (MSR)

Delft University of Technology
Software Engineering Research Group
Technical Report Series

Testing Principles, Current Practices, and Effects of Change Localization

Steven Raemaekers, Gabriela F. Nane, Arie van Deursen, and Joost Visser

Report TUD-SERG-2013-004

TU Delft

SERG

TUD-SERG-2013-004

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication in the Proceedings of the 10th Working Conference on Mining Software Repositories 2013, IEEE. http://dl.acm.org/citation.cfm?id=2487136

Testing Principles, Current Practices, and Effects of Change Localization

Steven Raemaekers*†, Gabriela F. Nane‡, Arie van Deursen†, and Joost Visser*

* Software Improvement Group, Amsterdam, The Netherlands
{s.raemaekers,j.visser}@sig.eu
† Delft University of Technology, Delft, The Netherlands
{s.b.a.raemaekers, arie.vandeursen, g.f.nane}@tudelft.nl
‡ Delft Institute of Applied Mathematics, Delft, The Netherlands

***Abstract*—Best practices in software development state that code that is likely to change should be encapsulated to localize possible modifications. In this paper, 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.**

***Index Terms*—Software libraries, encapsulation, ripple effects**

I. Introduction

Encapsulation is an important design principle in modern software development. The famous "Gang of Four" describe design patterns [7] which have the primary goal of encapsulating change and hiding implementation details. Booch [3] 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 paper, 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 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 paper 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 paper is as follows. In Section II, the problem is stated. Section III explains how data was obtained and what techniques have been used to calculate metrics. Section IV discusses our dataset. In Sections VII and VIII, results of our analysis can be found. In Section IX and X, threats to validity and a discussion can be found. Section XI discusses related work and Section XII concludes the paper. An online addendum containing links to websites of software mentioned in this paper is available at `http://www.sig.eu/en/msr2013a`. Source code of the analysis, the complete dataset and a description of this dataset is available at `http://www.sig.eu/en/msr2013b`.

II. Problem Statement

A. Illustrative Example

As an illustrative example, 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 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.

First Version	*Second Version* / *Date*	*1.0.57*	*1.0.68*	*1.0.79*	*1.1.110*	*1.2.121*	*1.2.133*	*1.2.147*	*1.3.158*
1.0.57	26-aug-'07	0	7	13	16	16	26	27	27
1.0.68	15-mar-'08	29	0	7	12	12	30	31	31
1.0.79	26-sep-'08	102	81	0	5	5	23	24	24
1.1.110	03-apr-'09	114	97	25	0	3	23	23	26
1.2.121	11-oct-'09	139	121	63	40	0	21	22	22
1.2.133	10-apr-'10	189	184	127	114	104	0	15	15
1.2.147	21-nov-'10	197	195	142	127	121	24	0	0
1.3.158	17-jul-'11	219	221	166	153	150	68	45	0

Table 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.

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: 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.

B. Research Questions

In this paper, 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 paper, we refer to the H2 database system as a running example to illustrate concepts and models as described in this paper.

III. Conceptual Framework and Methodology

We define *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 changes 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 meaure stability, we use three metrics which we defined in previous work [15]:

- *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 a library version with 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 version with a large amount of used methods removed from its interface.
- *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 a large growth in new methods.

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. In this paper 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 also use *WRM* and *PNM* as dependent variables in our statistical analyses.

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 answer **RQ1** and **RQ3** by computing correlations between properties of individual libraries. 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 [10]) is applied to the linear regression models in this paper, meaning that estimates of standard errors and p-values are robust against violations of normality, homoscedastiscity and independence [6]. 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 that properties are linearly correlated. 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 [1], [18]. To account for the graph structure of dependencies between libraries, more advanced statistical methods are required. Multilevel modeling [17] 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, 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. Second, developers of popular systems might feel limited in their freedom to change existing interfaces because a larger number of other systems depend on it. Either way, this becomes visible by calculating the correlation between library stability and network metrics such as the PageRank [13]. 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 other code 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.

IV. Dataset

We use the Maven repository for our analysis, a collection source and binary jar files of third-party libraries. Maven is a build configuration tool in which third-party library dependencies can be specified in the build file of a project. When building a project, Maven automatically retrieves the required libraries from a central repository. We downloaded a snapshot of this repository, dated July 30, 2011. It contains 148,253 separate jar files, approximately 20,000 separate projects and on average 7 versions per project.

We chose the Maven repository as dataset because it is the largest collection of open-source third-party Java libraries publicly available. We assume that programming practices in the Java programming language as investigated in this paper are representative for practices in other object-oriented languages.

For a more detailed description of our dataset and a download location, see [16]. Additional information is also available on the accompanying websites mentioned earlier.

V. 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[1] was used, which was adapted to run in parallel on multiple machines.

Specialized data structures were required to store the large amount of data. Berkeley DB, 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. For example, the package prefix of H2 is `com.h2database`. For each jar file, the number of files that contain an import 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

[1] http://www.sig.eu/en

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 [15], 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.

VI. Descriptive Statistics

In Table 2, descriptive statistics of our dataset 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.

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

Table 2. Descriptive statitistics for libraries in the Maven repository. nM = number of methods, nC = number of classes, LOC = lines of code, inD = indegree, outD = outdegree, 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 SLOC. 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", which only contain configuration files and no Java code. Libraries are used 24.42 times (indegree) on average and use 6.5 other libraries (outdegree) 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.

VII. Individual Library Results

A. The Relationship Between Library Properties

To answer **RQ1**, we inspect relationships between properties of individual libraries. These relationships are shown in Table 4. In the lower left part of the table, below the diagonal, the strength of correlations can be found [4]. 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 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 3 shows the relationships between system size (nM) and the average incoming (inI) and outgoing isolation (outI) rating. 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 4 further shows that network metrics, such as the PageRank and the Hubbiness and Authoritativeness from the HITS-algorithm [9] 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 removing 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.

	PageR.	*Hubb.*	*Auth.*	*WRM*	*CEM*	*RCNO*	*PNM*	*nM*	*nMn*	*nMr*	*InI.*	*OutI.*	*InD.*	*OutD.*
PageRank	1.00	*9.0e-78*	*0*	*9.7e-165*	*6.0e-233*	*1.4e-267*	*5.6e-182*	*0*	*1.2e-267*	*4.8e-162*		*1.5e-135*	*0*	*1.8e-26*
Hubbiness	-0.06	1.00	*6.5e-08*	*2.8e-67*	*8.1e-73*	*3.9e-75*	*3.7e-56*	*0*	*3.4e-75*	*4.0e-64*		*0*	*8.8e-13*	*0*
Authoritativeness	0.96	-0.02	1.00	*6.0e-155*	*1.8e-217*	*3.1e-245*	*2.8e-173*	*0*	*1.3e-247*	*4.8e-152*	*2.5e-125*	*1.0e-133*	*0*	*9.4e-10*
WRM	0.11	0.07	0.11	1.00	*0*	*0*	*0*	*0*	*0*	*0*		*0*	*2.4e-20*	*2.0e-239*
CEM	0.13	0.07	0.13	0.53	1.00	*0*	*0*	*0*	*0*	*0*		*1.8e-290*	*4.6e-25*	*4.3e-245*
RCNO	0.14	0.08	0.14	0.58	0.83	1.00	*0*	*0*	*0*	*0*		*0*	*1.5e-32*	*3.2e-250*
PNM	0.12	0.07	0.12	0.75	0.59	0.70	1.00	*0*	*0*	*0*		*0*	*9.4e-22*	*1.3e-196*
Number of methods (nM)	0.19	0.17	0.17	0.36	0.39	0.43	0.35	1.00	*0*	*0*	*1.9e-14*	*0*	*5.5e-74*	*0*
Number new methods (nMn)	0.14	0.08	0.14	0.78	0.62	0.75	0.98	0.44	1.00	*0*		*0*	*3.9e-36*	*1.7e-267*
Number removed methods (nMr)	0.11	0.07	0.11	1.00	0.52	0.58	0.76	0.36	0.78	1.00		*0*	*2.8e-19*	*7.4e-236*
Avg. Incoming Isolation (InI.)			0.24					-0.11			1.00	*2.0e-11*	*3.8e-61*	*4.5e-16*
Avg. Outgoing Isolation (OutI.)	0.10	0.31	0.10	0.20	0.16	0.20	0.19	0.47	0.24	0.20	-0.08	1.00	*2.0e-09*	*0*
Indegree (InD.)	0.45	0.06	0.33	0.12	0.13	0.15	0.12	0.23	0.16	0.11	0.17	0.06	1.00	
Outdegree (OutD.)	-0.04	0.54	-0.02	0.14	0.14	0.14	0.13	0.24	0.15	0.14	-0.09	0.39		1.00

Table 4. Spearman rank correlation matrix for jar file properties. All shown correlations are significant at the 0.0005 (0.05 / 105) level. In the upper right part of the table, above the diagonal, p-values can be found. PageR. = PageRank, Hubb. = hubbiness, Auth. = authoritativeness.

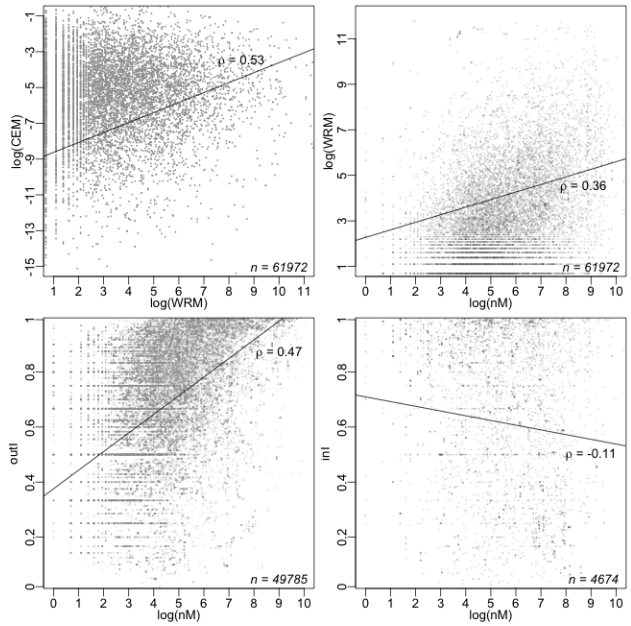


Fig. 3. Scatterplots and Spearman rank correlations between, CEM, WRM, number of methods (nM) and average incoming (inI) and outgoing isolation rating (outI). The axis limits have been adjusted and logarithmic transformations have been applied to demonstrate the relationships more clearly.

B. 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. 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 7394 observations and the results of this model are shown in Table 5.

Independents	*Coeff.*	*Beta*	*Std. Err.*	*p-value*	*95% C.I.*
log(CEM)	0.094	0.099	0.011	0.000	0.073 - 0.115
log(nM)	0.939	0.674	0.016	0.000	0.907 - 0.970
log(PNM)	0.694	0.487	0.017	0.000	0.661 - 0.726
outI	-0.439	-0.042	0.106	0.000	-0.647 - -0.230
outD	0.007	0.029	0.002	0.003	0.003 - 0.012
constant	0.352	-	0.093	0.000	0.169 - 0.534

Table 5. A regression model performed on the number of removed methods from library interfaces.

Expressed as a formula, the model looks as follows:

$$\begin{aligned} \textit{log(WRM)} = 0.352 &+ 0.094\textit{log(CEM)} + 0.939\textit{log(nM)} \\ &+ 0.694\textit{log(PNM)} - 0.439\textit{outI} + 0.007\textit{outD} \end{aligned} \quad (1)$$

The effect of all predictors in our model is significant with p-values close to 0. The standardized coefficients ("Beta") in Table 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(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 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 incuded

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 $2.1 * 10^{-4}$, 7058 units, a PNM of $8.8 * 10^{-3}$, 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.

VIII. 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.

A. 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 6. Expressed as a formula, the model looks as follows:

$$log(WRM_{from}) = 3.24 + 0.037 log(WRM_{to}) - 1.17 is \quad (2)$$

Independents	*Coeff.*	*Std. Err.*	*p-value*	*95% C.I.*
$log(WRM_{to})$	0.037	0.011	0.001	0.015 - 0.059
isolation	-1.17	0.109	0.000	-1.38 - -0.957
constant	3.24	0.061	0.000	3.12 - 3.36

Table 6. A regression model with stability in libraries and their dependencies, and the isolation rating between them.

The model is based on 6813 cases, is significant with a p-value of 0.000 but has an R^2 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 thus shows that there indeed exists a positive effect of dependency instability on library instability while correcting for isolation (*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 VIII-C.

B. 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 ρ 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.*

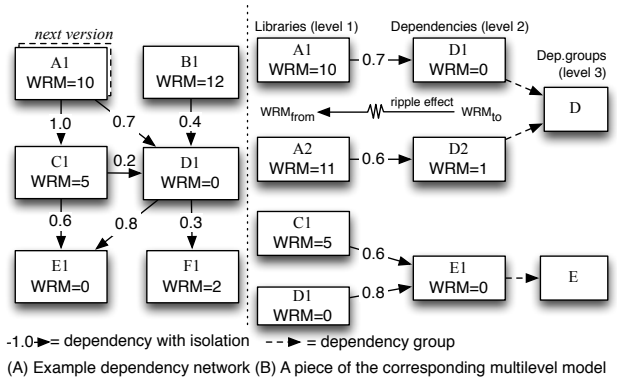
C. Multilevel Model for Interface Instability

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 nonnormality 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 also 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 [1], [18].

The same principles apply in our dataset, of which Figure 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 7A can be found in Table 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 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 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.



(A) Example dependency network (B) A piece of the corresponding multilevel model

Fig. 7. An example of libraries and their dependencies. On the right side, the corresponding multilevel representation of the left side is shown. Not all nodes and relationships from panel A are included in panel B.

from	*to*	WRM_{from}	WRM_{to}	*isolation*	Lib_{from}	Lib_{to}
A1	C1	10	5	1.0	A	C
A1	D1	10	0	0.7	A	D
A2	D2	11	1	0.6	A	D
B1	D1	12	0	0.4	B	D
C1	D1	5	0	0.2	C	D
C1	E1	5	0	0.6	C	E
D1	E1	0	0	0.8	D	E
D1	F1	0	2	0.3	D	F

Duplicated measurement | Time dependency | Same Artifact | Independent observations

Table. 8. Tabular form of the diagram presented in Figure 7A. Multiple sources of data dependencies can be identified in this table.

D. Model Specification

Figure 7B shows a piece of the corresponding multilevel model. In this paper, 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 VIII-A, 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 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.

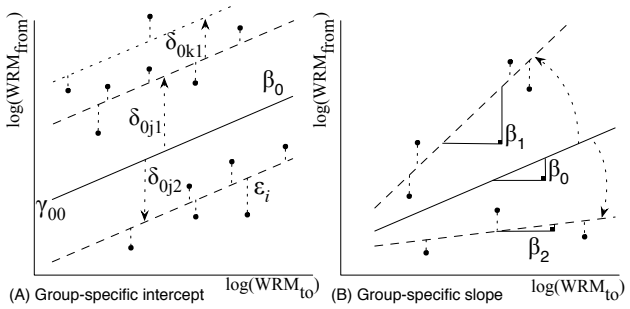


Fig. 9. Visual representation of group-specific intercept and slope. In figure A, the intercept is allowed to vary but regression lines stay parallel; the total group-specific intercept is $\gamma_{00} + \delta_{0j} + \delta_{0k}$. In figure B, slopes are allowed to vary, leading to a different β per group.

Dependent variable	log(WRM*from*)
Level 1 Group variable	Dependency versions
Level 2 Group variable	Dependencies
Number of cases	6813
Number of level 1 groups	1277
Number of level 2 groups	641
Cases per level 1 group	min 1, avg 5.3, max 523
Cases per level 2 group	min 1, avg 10.6, max 649
Wald χ^2	65.31
P($> \chi^2$)	0.000

Single library effects parameter estimates				
Independents	*Coeff.*	*Std. Err*	*p-value*	*95% C.I.*
γ_{00}	3.269	0.099	0.000	3.074 - 3.463
γ_{01}	0.037	0.018	0.045	0.000 - 0.073
γ_{02}	-1.079	0.183	0.000	-1.437 - -0.721

Group effects parameter estimates			
	Estimate	*Std. Err*	*95% C.I.*
$\sigma(\delta_{0j})$	0.317	0.080	0.194 - 0.518
$\sigma(\delta_{0k})$	0.636	0.112	0.450 - 0.897
$\sigma(\delta_{2k})$	1.203	0.358	0.671 - 2.156
σ(*Residual*)	5.515	0.100	5.323 - 5.715

Fig. 10. A multilevel model for removals of interface methods.

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

$$log(WRM_{ijk}) = \beta_{0jk} + \beta_1 log(WRM_{jk}) + \beta_2 is + \epsilon_{ijk} \quad (3)$$
$$\beta_{0jk} = \gamma_{00} + \delta_{0j} + \delta_{0k}$$
$$\beta_1 = \gamma_{01}$$
$$\beta_{2k} = \gamma_{02} + \delta_{2k}$$

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 γ_{00}, each dependency version adds a version-specific intercept δ_{0j} and each dependency adds a dependency-specific intercept δ_{0k}. Comparing this model to formula 2, β_0 is 3.24, β_1 is 0.037 and β_2 is -1.17. The individual error term is not shown in formula 2.

E. Multilevel Model Explanation

The results of this model are shown in Table 10. The overall model is significant with a p-value of 0.000. The model yields a grand mean γ_{00} of 3.27, which is the average *log(WRM)* in libraries. This is comparable to our earlier result of 3.24 in formula 2. The ripple effect γ_{01} attributable to dependencies is exactly the same, namely 0.037. This is expected since there are no group-specific parameters for β_1; the estimation method for single libraries (the middle table in Figure 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.*

IX. Threats to Validity

A. 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 erronerous 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.

B. 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.

X. Discussion

A. 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 paper does not immediately provide actionable advice, it is worth investigating the practical consequences of the analysis. Looking at the linear regression in figure 5, 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 VIII-A 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 VIII-A. This means that the results found in Section VIII-A 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.

B. Relationship with Aspect-oriented Programming

The isolation rating as defined in this paper 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 features, 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 paper, 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.

C. Model Misspecification Risk

In applying complex statistical techniques such as the multilevel modeling technique in this paper, 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 paper 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 VIII-C are expected to be common.

D. Multiple Hypotheses Testing

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 therefore focus more on the strength of correlations.

We applied a Bonferroni adjustment factor on the correlations in table 4, but there exist strong 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 [14] and [12]. Nonetheless, even with a conservative Bonferroni adjustment factor of 105 as applied in this paper, the largest part of tested correlations is significant at the 0.0005 level.

XI. Related Work

Similar to the kind of relationships investigated in this paper, Mohagheghi [11] 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 extensively, for example by Herzig [8], 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 [2] 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 paper.

Cossette et al. [5] 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 VIII-A.

In previous work [15] 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.

XII. Conclusion

In this paper, we made the following contributions:

- A large-scale experimental setup to process a large number of source files;
- 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.

References

[1] M. Aitkin, D. Anderson, and J. Hinde. Statistical modelling of data on teaching styles. *Journal of the Royal Statistical Society. Series A (General)*, 144(4):419–461, 1981.

[2] S. Black. Deriving an approximation algorithm for automatic computation of ripple effect measures. *Information & Software Technology*, 50(7-8):723–736, 2008.

[3] G. Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Redwood City, USA, 1994.

[4] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences, Second Edition*. Lawrence Erlbaum Associates, Publ., 1988.

[5] B. Cossette and R. J. Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *SIGSOFT FSE*, page 55, 2012.

[6] J. Fox. *Applied Regression Analysis and Generalized Linear Models, Second Edition*. SAGE Publications, Inc., 2008.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] K. S. Herzig. Capturing the long-term impact of changes. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 393–396, New York, NY, 2010. ACM.

[9] J. M. Kleinberg. Hubs, authorities, and communities. *ACM Comput. Surv.*, 31(4es), Dec. 1999.

[10] G. Li. Robust regression. In D. Hoaglin, F. Mosteller, and J. Tukey, editors, *Exploring Data Tables, Trends, and Shapes*, pages 281–343. John Wiley & Sons, New York, 1985.

[11] P. Mohagheghi, R. Conradi, O. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, ICSE'04, pages 282–291, may 2004.

[12] S. Nakagawa. A farewell to Bonferroni: the problems of low statistical power and publication bias. *Behavioral Ecology*, 15(6):1044–1045, 2004.

[13] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the web. Technical report, November 1999.

[14] T. V. Perneger. What's wrong with Bonferroni adjustments. *British Medical Journal*, 316:1236–1238, 1998.

[15] S. Raemaekers, A. v. Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *28th IEEE International Conference on Software Maintenance (ICSM)*, 2012.

[16] S. Raemaekers, A. v. Deursen, and J. Visser. The maven repository dataset of metrics, changes, and dependencies. In *10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[17] T. Snijders and R. Bosker. *Multilevel Analysis: An Introduction to Basic and Advanced Multilevel Modeling, Second Edition*. SAGE Publications, Inc., 2011.

[18] N. Spencer. Combining modelling strategies to analyse teaching styles data. *Quality and Quantity*, 36:113–127, 2002.

Addendum

To the paper *"Testing Principles, Current Practices and Effects of Change Localization"*

I. Mentioned websites

The following software is mentioned in the paper:

H2 Database	http://www.h2database.com
Clirr	http://clirr.sourceforge.net
DAS-3 Supercomputer	http://www.cs.vu.nl/das3
Berkeley DB	http://www.oracle.com/technetwork /products/berkeleydb
Neo4j Graph Database	http://www.neo4j.org
Apache Maven	http://maven.apache.org
Apache Ant	http://ant.apache.org

II. Correlations with maintainability

The following correlations have been calculated between the SIG maintainability rating and other code properties:

Property	*Spearman's ρ*	*p-value*
PageRank	-0.10	5.9e-165
Hubbiness	-0.08	7.4e-107
Authoritativeness	-0.09	3.7e-138
WRM	-0.23	0
CEM	-0.34	0
RCNO	-0.33	0
PNM	-0.22	0
dUn	-0.26	0
dUo	-0.34	0
nU	-0.57	0
nUn	-0.27	0
nUr	-0.22	0
OutI.	-0.20	0
OutD.	-0.18	0

Fig. 1. dUn = change in new methods, dUo = change in old methods, nU = number of methods, nUn = number of new methods, nUr = number of removed methods, OutI. = average outgoing isolation rating, OutD. = number of external dependencies.

The SIG Maintainability rating shows to be negatively correlated with all metrics in this table, such as network metrics (PageRank and Hubbiness and Authoritativeness), stability metrics (WRM, CEM, RCNO and PNM) and system size (nU). This indicates that code that is larger, changes faster and changes more tends to be less maintainable. The Maintainability rating is also negatively correlated with the number of incoming and outgoing dependencies, indicating that systems with a larger number of dependencies are less maintainable and libraries which are used more frequently by other libraries also tend to be less maintainable.

	min	*p5*	*p25*	*p50*	*p75*	*p95*	*max*	*avg*	*sd*
vol	2.78	5.18	5.46	5.49	5.50	5.50	5.50	5.41	0.36
dup	0.51	2.10	3.96	5.30	5.50	5.50	5.50	4.62	1.20
us	0.50	1.29	2.22	3.40	4.53	5.50	5.50	3.41	1.44
uc	0.50	1.34	2.37	4.07	5.50	5.50	5.50	3.91	1.64
ui	0.50	1.33	2.37	3.95	5.50	5.50	5.50	3.76	1.52
mc	0.61	2.40	4.82	5.50	5.50	5.50	5.50	4.96	1.02

Table 2. Descriptive statitistics for libraries in the Maven repository. vol = volume, dup = duplication, us = unit size, uc = unit complexity, ui = unit interfacing, mc = module coupling

Table 2 contains SIG star ratings, ranging from 0.5 to 5.5. A rating of 0.5 to 1.5 means that the system scores the same as the worst 5% of systems in our industrial benchmark of more than 500 software systems, and a score of 4.5 to 5.5 means that the system scores the same as the best 5%. Scores in between are evenly distributed, meaning that 30% of systems in our benchmark score between 1.5 and 2.5, 30% score between 2.5 and 3.5 and 30% score between 3.5 and 4.5. The reason that the 5th percentile of volume in Table 2 is not at 1.5 is that apparently, systems in the Maven repository are relatively small compared to our industrial benchmark. The same is true for duplication, which means that systems in the Maven repository generally have less duplication than systems in our benchmark. For an explanation of the other star ratings see [1].

Figure 3 and 4 show the relationship between the Maintainability rating and system size and change. Figure 3 shows that bigger systems (log(nrUnits)) tend to be less maintainable (Maintainability / volume), indicated by a negative Spearman's correlation coefficient of -0.51. In this figure, Maintainability is corrected for system volume since system size is a component of maintainability. The figure shows that even after correcting for volume, bigger systems still tend to be less maintainable. Figure 4 show that systems that change more (log(CEM)) tend to be less maintainable (Maintainability), as indicated by a correlation coefficient of -0.34.

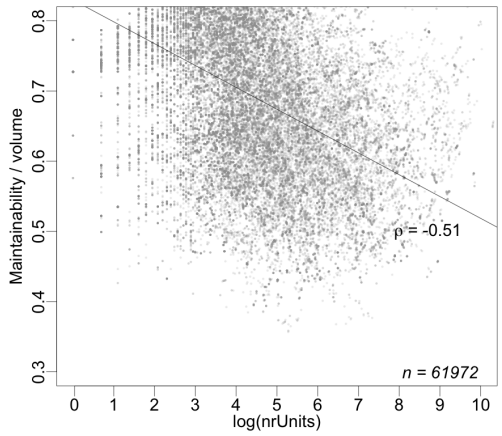


Figure 3. The correlation between the SIG Maintainability rating and system size as measured in number of units. Logarithmic transformations have been applied to demonstrate the relationships more clearly. This does not affect the strength of the correlation.

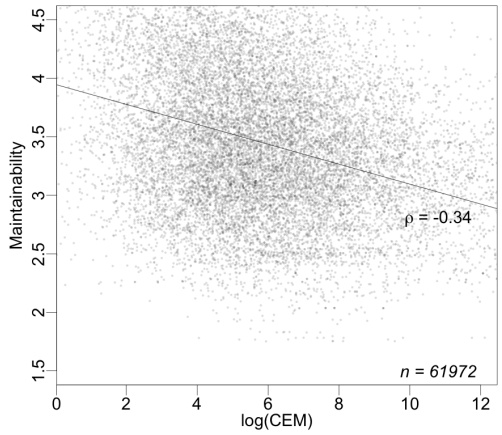


Figure 4. The correlation between the SIG Maintainability rating and log(CEM). Logarithmic transformations have been applied to demonstrate the relationships more clearly. This does not affect the strength of the correlation.

REFERENCES

[1] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pages 30–39, Washington, DC, USA, 2007. IEEE Computer Society.

TUD-SERG-2013-004
ISSN 1872-5392