Are Developers Aware of the Architectural Impact of Their Changes?

Matheus Paixao*, Jens Krinke*, DongGyun Han*, Chaiyong Ragkhitwetsagul*, and Mark Harman*
* University College London, United Kingdom
{matheus.paixao.14, j.krinke, d.han.14, chaiyong.ragkhitwetsagul.14, mark.harman}@ucl.ac.uk

***Abstract*—Although considered one of the most important decisions in a software development lifecycle, empirical evidence on how developers perform and perceive architectural changes is still scarce. Given the large implications of architectural decisions, we do not know whether developers are aware of their changes' impact on the software's architecture, whether awareness leads to better changes, and whether automatically making developers aware would prevent degradation. Therefore, we use code review data of 4 open source systems to investigate the intent and awareness of developers when performing changes. We extracted 8,900 reviews for which the commits are available. 2,152 of the commits have changes in their computed architectural metrics, and 338 present significant changes to the architecture. We manually inspected all reviews for commits with significant changes and found that only in 38% of the time developers are discussing the impact of their changes on the architectural structure, suggesting a lack of awareness. Finally, we observed that developers tend to be more aware of the architectural impact of their changes when the architectural structure is improved, suggesting that developers should be automatically made aware when their changes degrade the architectural structure.**

***Index Terms*—Software Architecture, Code Reviews**

I. Introduction

Architectural decisions are among the most important decisions to be taken by practitioners [1], due to the high risks and costs accrued by poor decisions [2]. Recent studies have empirically shown that bug-prone files are more architecturally connected than clean files [3], and that architectural flaws can lead to increased maintenance effort [4]. The notions of cohesion and coupling as guides for software architecture design have been extensively associated with different aspects of software quality, including, but not limited to, maintainability [5], [6], comprehensibility [7], [8] and code smells [9], [10]. Structural dependencies between code components were the most used assets for cohesion and coupling measurement for many years [11]–[14], where other sources of information have been taken into account more recently, such as semantics [15] and revision history [16]. Nevertheless, recent studies [17], [18] have revealed structural dependencies to be one of the best proxies for developers' perception of cohesion and coupling.

Despite the large body of work aimed at aiding developers in the structural organisation of systems [19]–[21], we still see evidence of architectural erosion as systems evolve [22], [23]. Developers sometimes *choose* to accept suboptimal solutions in order to achieve a desired goal, such as short term delivery [24]; thereby accruing technical debt [25]. Nevertheless, the reasons for a developer to accept a solution that will damage the software architecture or to neglect the refactoring of an eroded architecture are still open for investigation. As pointed out by recent studies with developers [17], [18], different systems and different developers work under different conditions and have different perspectives regarding architectural quality. This diversity indicates the need for studies aimed at better *understanding* how developers deal with architectural changes.

In this paper, we extend the body of empirical knowledge regarding architectural changes in software systems by studying these changes on a day-to-day basis. We investigate the *intent* of developers when performing changes that will impact the system's architecture. Moreover, we also assess whether developers are *aware* of the architectural impact of their changes at the time these changes are being made.

To the best of our knowledge, no existing work addresses developers' intent and awareness when performing architectural changes on a day-to-day basis. Quantitative studies evaluating metrics and techniques for structural optimisation [19], [20], [26] show how much architectural improvement can be achieved in software systems, but the feedback from developers is usually insufficient. Qualitative studies interview developers regarding architectural quality by either using toy systems [27] or selected past changes [17], [18]. Since surveys are subjective to bias [28] and the questionnaires usually target the software system as whole, such studies fail to capture details and nuances of each particular architectural change.

In order to understand the intent and the architectural awareness of developers when performing architectural changes on a day-to-day basis, we mined code review data [29] to extract significant architectural changes and infer the developers' intent and architectural awareness. During the process of code review, a change is only incorporated into the system after an inspection of the patch being submitted. The author of the change submits the patch and a natural language description of the change, where other developers will have the opportunity to review the code and provide feedback. Reviewers may request the author to improve the patch and the author will do so until the change is incorporated in the system or discarded.

The code review process provides detailed information about each change, which enables us to perform the empirical study on which we report here. For each change, we have the source code from which cohesion and coupling metrics can be computed, and a natural language description that was submitted alongside the change from which the intent can be inferred. Based on the change's description and the

978-1-5386-2684-9/17 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

feedback provided by other developers, we can seek evidence of developers' awareness, at the time the change was being made, of the architectural impact of each specific change.

After mining a total of 8,900 code reviews from 4 software systems, we used a metric-based approach to identify reviews that changed the structural architecture of the systems. For 626 architectural changing reviews, we performed a manual analysis and classification of the reviews according to the *intent* of the review and the *architectural awareness* of the developers involved in the review. The inference of each review's intent and architectural awareness is based on the reviews' description and feedback provided by developers (no interviews have been performed).

The main contributions of the paper are listed as follows:

1) We found that developers discuss the architectural impact of their changes in only 38% of the reviews with significant impact to the system's architecture. In addition, reviews in which the architecture is discussed tend to have higher structural improvement than reviews where the architecture is not discussed. This provides evidence and reasoning for the introduction of tools for automated architectural analysis at reviewing time, where the developers would be automatically made aware of the architectural impact of their changes.
2) A dataset of 626 manually classified code reviews that include the intent of each review and the architectural awareness of developers involved in each review.
3) A dataset of 17,800 structural architectures extracted from the source code of 4 open source software systems, which corresponds to a total of 17 years and 2 months of development time.

II. Background

In an object-oriented context, structural metrics of cohesion and coupling assess how the code is organised in terms of its structural dependencies between classes and packages. These dependencies capture compile time dependencies, such as method calls, data access and inheritance. In this paper, the architectural structure of a system is represented as a Module Dependency Graph (MDG) [14]. Once the MDG of a system is computed, structural cohesion and coupling measurements can be used to assess the system's structure. In this paper, we employ a set of structural metrics for cohesion and coupling measurement that have been quantitatively and qualitatively evaluated in a recent study [18].

The structural cohesion of the MDG M of a certain system, consisting of m packages $P_1, ..., P_m$, is assessed by measuring the lack of structural cohesion, which is computed as

$$\mathrm{LStrCoh}(M) = \frac{\sum_{j=1}^{m} \mathrm{LCOF}_{P_j}}{m}, \quad (1)$$

where LCOF_{P_j} represents the Lack of Cohesion of Files for package P_j. LCOF_{P_j} is computed as the number of pairs of files in P_j without a structural dependency between them. Packages with a high amount of unrelated files will be scored a high LCOF, and, accordingly, packages with only a few unrelated files will be scored a low LCOF.

Consider a review that changed the system's structural architecture. LStrCoh is used to measure the cohesion of the system both before (M_i) and after (M_{i+1}) the review. In this case, LStrCoh is an inverse metric, where a positive difference in $\mathrm{LStrCoh}(M_{i+1}) - \mathrm{LStrCoh}(M_i)$ indicates higher lack of cohesion, and therefore, a *degradation* in structural cohesion as a result of the review. Similarly, a negative difference in LStrCoh indicates an *improvement* in structural cohesion.

The structural coupling of M, StrCop, is computed as

$$\mathrm{StrCop}(M) = \frac{\sum_{j=1}^{m} \mathrm{FanOut}_{P_j}}{m}, \quad (2)$$

where FanOut_{P_j} indicates the number of files outside package P_j that depend on files inside P_j. Similarly to LStrCoh, a positive difference in $\mathrm{StrCop}(M_{i+1}) - \mathrm{StrCop}(M_i)$ after a review indicates a *degradation* in structural coupling, and a negative difference after a review indicates an *improvement* in structural coupling.

III. Experimental Design

The goal of this paper is to study the intent and the architectural awareness of developers when performing architectural changes on a day-to-day basis. To this end, we ask the following research questions:

RQ1: *What are common intents when developers perform significant changes to the architecture?* This research question investigates architectural changes and identifies common intents behind these changes. Thus, we classify architectural changes regarding their intent at the time the change was reviewed, such as *New Feature*, *Bug Fixing* and so on. Using this approach we can perform our analysis on the most recurrent intents, thereby achieving a better understanding of the conditions under which architectural changes were performed.

RQ2: *How often are developers aware of the architectural impact of their changes on a day-to-day basis?* Given the large number of ramifications of an architectural change, this research question investigates how often developers are aware of the impact of their changes on the system's structure. To answer it, we inspect changes that had an impact on the architectural structure to identify whether developers discuss the system's architecture during the review of that change.

RQ3: *How does awareness and intent influence architectural changes on a day-to-day basis?* Considering changes with common intents, we assess how the architectural awareness of developers influences the improvement or degradation of cohesion and coupling for each change.

The rest of this section reports the experimental methodology we used to answer the research questions presented above.

A. Code Reviews Data Mining

Code review in modern software development is a lightweight process in which changes proposed by developers are first reviewed by other developers before incorporation in the system. In this paper, we focus on Gerrit [30], one

of the most popular code review systems currently in use by large open source communities, such as Eclipse [31] and Couchbase [32]. Although we focus on Gerrit in this paper, the methodology presented here is adaptable and extensible for other code review systems.

In Gerrit, a developer submits a new patch for review in the form of a git commit, where the commit message is used as the review's description and the commit id is stored for future reference. For each new submission, Gerrit creates a *Change-Id* to be used as a unique identifier of that review throughout its reviewing cycle. Other developers of the systems will then inspect the patch, and provide feedback in the form of comments. Improved patches are submitted according to the feedback until the review is *merged* or *abandoned*, where the first indicates the patch was incorporated to the system and the latter indicates the patch was rejected. For the rest of this paper, we use review and change interchangeably to indicate a code submission that was manually inspected by developers and later merged or abandoned.

After selecting the subject systems for our empirical study (Section III-B), we mined the systems' code review data from Gerrit and downloaded the systems' git repositories. Since the goal of this study is to better understand how architectural changes are performed, we focus our analysis on the merged reviews. Hence, with the aid of the review's original commit id and Gerrit's *Change-Id*, we linked each merged review to the *commit* in the git repository where the review was incorporated[1]. Moreover, we also identify the commit that immediately preceded each review. Finally, for each review, we collected the author's description of the change, the feedback provided by other developers and the commits that represent the state of the system *before* and *after* the review.

B. Subjects Selection for Empirical Investigation

As we needed systems with a rich review history, the number of *Merged* reviews was one of the criteria used when selecting the subject systems for investigation. Moreover, since our static analysis framework can only be applied to Java systems (see Section III-C), we used the proportion of Java code in the system as the other selection criteria.

Thus, we selected the two systems with the most merged reviews and the highest proportion of Java code in two open source communities, i.e. `egit` and `linuxtools` for Eclipse, and `couchbase-java-client` and `couchbase-jvm-core` for Couchbase. For brevity, the Couchbase systems will be abbreviated as `java-client` and `jvm-core`, respectively. The consideration of these 4 systems yielded a manual inspection and classification of 628 code reviews, highlighting the high level of manual analysis involved in this study. This high level of painstaking manual analysis is required to form a ground truth, which will assist other researchers in subsequent studies, facilitating greater automation. Table I reports the number of *Merged* reviews for each system and the time span of the system's history we are investigating. Moreover, we also report the proportion of Java code for each system and size metrics. Since the proportion of Java code and the size of the systems change throughout their history, we additionally report median, maximum and minimum values for these statistics.

Both `egit` and `linuxtools` are plugins for Eclipse, where the first is aimed at providing git support in Eclipse, and the second provides a C/C++ IDE for Linux developers. Couchbase as a whole is a NoSQL database solution for both server side and mobile, where `java-client` is the official driver to access the Couchbase database using Java, and the `jvm-core` is a low-level API mostly used by the `java-client` implementation. Considering the 4 systems, we have access to a rich review history, with a combined total of 17 years and 2 months of development time.

C. Computing the Difference in Structural Cohesion and Coupling for Reviewed Changes

For each system selected to participate in our empirical study, we computed the difference in structural cohesion and coupling for the changes that have undergone a process of code review as described in Section III-A, where the formal definitions of the metrics being computed are presented in Section II.

The computation of the difference in structural cohesion and coupling for all code reviews we mined is depicted in the first steps of the framework presented in Figure 1. For each code review, we revert the system to the states before and after the review took place, creating two snapshots where the difference between them was induced by the merged review.

We subsequently filter all the test code in the system's code base. Although part of the project, test code is not included in the end product, and so we chose not to include it as part of the structural architecture. In this paper, we employ a two-stage procedure for test code filtering. In the first stage, every file under a `test/` folder is filtered. Next, all remaining files with `Test` or `test` in the file name are manually analysed by two of the authors, where a decision is reached to either include or filter the file from the structural architecture analysis.

After filtering test code, we extract the MDG representing the structural architecture of the system for the snapshots before and after the review. Previous studies that performed architecture analyses in Java systems relied on bytecode analysis for structural architecture extraction [18], [20], [33]–[37]. However, building and compiling the systems for each commit is a time consuming and error prone activity. Hence, for this investigation, we extract the architectural structure of a system directly from its *source code* by using Understand [38], a commercial tool for static code analysis whose set of features include dependencies extraction and visualisation.

Given the system's MDG before and after the review, we compute the structural cohesion and coupling as defined in Equations 1 and 2 and compare the cohesion and coupling of the system before and after the review. The measurements of structural cohesion and coupling are separately computed for each package in the structural architecture, and then aggregated in an overall score. Hence, when comparing the cohesion and coupling for before and after the review, we store not only the

[1]In rare cases, a link cannot be established and we discarded such reviews.

TABLE I: DESCRIPTIVE STATISTICS FOR THE SYSTEMS UNDER STUDY. WE REPORT THE NUMBER OF MERGED REVIEWS IN EACH SYSTEM AND THE TIME SPAN OF OUR INVESTIGATION. WE REPORT THE MEDIAN, MAXIMUM AND MINIMUM VALUES OF SIZE METRICS.

Systems	No. of Reviews	Time Span (Months)	Proportion of Java Code (%)			kLOC			Number of Packages			Number of Files			Number of Dependencies		
			Med	Max	Min	Med	Max	Min	Med	Max	Min	Med	Max	Min	Med	Max	Min
egit	3983	9/09 to 5/16 (80)	91	99	84	67	99	16	57	80	19	602	801	137	2508	3779	347
linuxtools	3633	6/12 to 5/16 (47)	90	94	85	178	215	90	351	435	214	1825	2199	1083	6765	8777	3103
java-client	716	11/11 to 5/16 (54)	100	100	97	8	24	1	14	39	3	125	420	13	392	1612	22
jvm-core	568	4/14 to 5/16 (25)	100	100	100	13	18	2	38	50	17	286	386	74	1106	1639	230

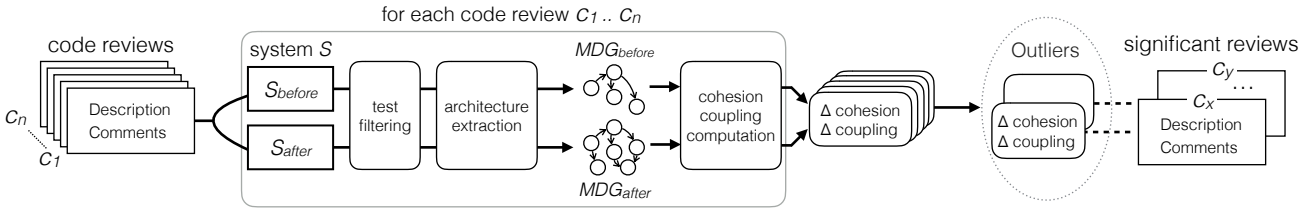

Fig. 1: Framework for the identification of code reviews with significant changes to the system's architecture. Given a set of code reviews, our automated framework identifies significant reviews in terms of the impact to system's architectural structure.

overall difference, but also the biggest difference in a single package. We thus expand our analysis to consider not only changes to the overall structural architecture, but also changes that highly affect a single package.

At the end of this process, four different values are stored for each review, each of which indicate the difference in overall cohesion/coupling and the biggest difference in cohesion/coupling for a single package, respectively. In this paper, we computed the differences in cohesion and coupling for 8,900 code reviews, which generated a dataset of 17,800 structural architectures automatically extracted from source code. The dataset of all extracted structural architectures and the respective cohesion and coupling values computed for each review is available at our supporting webpage [39].

D. Identification of Reviews with Significant Architectural Changes

In order to identify the reviews that performed significant changes to the system's architecture, we employed an outlier-based approach. At first we grouped the set of code reviews according to the following criteria. We identified all reviews that showed an improvement in overall cohesion, followed by all reviews with an improvement in overall coupling. We then identified all reviews that showed a cohesion improvement for a certain package, followed by all reviews with a coupling improvement for a certain package. Similarly, we identified all reviews that showed a degradation in the cohesion and coupling measurements presented above. In total, we grouped the reviews on 8 different subsets, which stand for the reviews that improve or degrade the cohesion and coupling of either the overall structural architecture or a single package.

Next, for each of the 8 subsets, we identified the outliers using Tukey's method [40] and defined the upper outlier "fence" as $1.5 \times$ IQR (interquartile range) from the third quartile (Q3) over the distribution of measurements in the specific subset. The outliers indicate the reviews with "significant" differences in cohesion and coupling relative to the overall distribution. Table II presents the number of reviews identified as outliers for each subset discussed above, and for each system under study. Additionally, since reviews can be identified as outliers in more than one subset, we also report the number of unique reviews identified as outliers when considering all subsets.

TABLE II: NUMBER OF REVIEWS IDENTIFIED AS OUTLIERS ACCORDING TO DIFFERENT ARCHITECTURAL ASPECTS. WE REPORT THE NUMBER OF OUTLIERS FOR THE REVIEWS WITH AN OVERALL IMPROVEMENT ($\oplus$) OR DEGRADATION ($\ominus$) IN COHESION AND/OR COUPLING. WE ALSO REPORT THE NUMBER OF OUTLIERS FOR A SINGLE PACKAGE. THE NUMBER OF UNIQUE OUTLIERS CONSIDERS ALL ASPECTS DISCUSSED ABOVE.

System	Cohesion				Coupling				Unique Outliers
	Overall		Single P.		Overall		Single P.		
	$\oplus$	$\ominus$	$\oplus$	$\ominus$	$\oplus$	$\ominus$	$\oplus$	$\ominus$	
egit	5	33	15	34	4	63	13	40	122
linuxtools	30	34	17	31	24	49	37	63	164
java-client	5	10	2	11	3	12	2	11	27
jvm-core	0	4	0	7	1	17	1	9	25
All	40	81	34	83	32	141	53	123	338

As one can see from the table, 338 reviews were automatically identified as the ones presenting the biggest changes in structural cohesion and coupling, indicating that these reviews are the ones that performed significant changes to the systems' architecture. The subset of 338 unique reviews identified as outliers stand for 15% of all reviews with architectural change.

E. Manual Inspection and Classification of Reviews

Following the automated process described in the previous section, we considered all 338 outlier reviews, and performed a manual inspection and classification inspired by the work of Tufano et al. [41]. The manual classification process consisted of two authors analysing each review and providing values for a set of *tags*. Each tag can assume *true* or *false*, and aim at describing a review in two dimensions: *intent of change*

and *architectural awareness*. In order to identify the reviews' intent, we performed an open coding classification process. As a starting point, we considered the set of tags originally proposed by Tufano et al. During the open coding classification, we augmented the set of tags with different intents that emerged from the reviews' data in a bottom-up fashion. The final set of tags used in the reviews' classification is presented in Table III, alongside a short description of each tag.

To assess architectural awareness, we rely on the review's description and/or comments to ascertain the developers' awareness of the architectural impact of the change. When developers discuss the structural architecture in the review's description or comments, we can be certain of the developer's awareness. However, when the architecture is not discussed, two scenarios are possible. In the first scenario, developers do not discuss the architecture because they are not aware of the impact of their changes. In the second scenario, developers are aware of the architectural impact, but *choose* not to discuss it during code review. We are therefore careful to couch over scientific conclusions in the conduct of our analysis which is a conservative, safe, under approximation of developer awareness.

In this paper, our analysis is focused on reviews that performed significant changes to the system's structural architecture. In this case, when the author do not discuss the architecture in the review's description, reviewers who are not familiar with the change might not be able to understand its impact in the architecture. Similarly, if a reviewer do not raise the architecture discussion during the reviewing process, the author of the change might not perceive the ramifications of the change being performed. In both cases, the lack of discussion in regard to the system's architecture during code review will lead to a lack of awareness of some developers involved in the review, which will ultimately lead to a poor reviewing process. Therefore, the (lack of) discussion of structural architecture during code review can be used as proxy for the developers' awareness regarding the impact of their changes.

In order to mitigate threats to internal validity during the classification process, we employed a two stages classification. In the first stage, each author solely inspected and classified the reviews according to a guideline that was discussed, reviewed and agreed by all authors. In the second stage, the authors discussed all the reviews for which there was a disagreement in the classification. For this paper, there was no disagreement in any reviews after the second stage of classification. The set of manually classified code reviews is available at our supporting webpage [39].

TABLE III: TAGS BEING USED IN THE MANUAL CLASSIFICATION OF CODE REVIEWS.

Intent of Change	
New Feature	Developer is adding a new feature to the system
Enhancement	Developer is enhancing an existing feature or code
Feature Removal	Developer is removing an obsolete feature
Platform Update	Developer is updating the code for a new platform/API
Refactoring	Developer is refactoring the system
Bug fixing	Developer is fixing a bug
Not clear	There's no evidence to suggest any of the previous
Architectural Awareness	
In Description	Architectural impact is discussed in the description
In Comments	Architectural impact is discussed in the comments
Never	Architectural impact is never discussed

F. Pre-study for Validation of Experimental Design

We focus on the reviews with significant changes as identified by the outliers. We assume that reviews with no significant impact on the structural architecture will have fewer discussion as developers may not expect a significant change (i.e. they are aware of the low impact and see no need to discuss it). Moreover, we also assume that the structural metrics of cohesion and coupling are indeed capturing changes in structural architecture and not only fluctuations in the system's size. To validate these assumptions, we performed both a qualitative and a quantitative pre-study.

First, we assessed whether an analysis of only outliers is suitable to answer the research questions we proposed. Thus, we collected all reviews from the Couchbase projects, including reviews with significant changes (see Section III-D) and reviews with no significant change (as identified by the outliers). On the set of 287 non significant changes, we performed an inspection as described in Section III-E and identified 45 reviews in which the architecture is discussed (15%). The ratio is much higher for the reviews with significant changes: developers discuss the architecture in 19 out of 52 (36%) reviews. The difference in the two groups of reviews is statistically confirmed by a two-tailed pooled test in which the statistical difference is confirmed at the 0.01 α level. It is therefore safe to assume that developers tend to discuss the structural architecture when they expect a significant change.

Next, we performed an analysis to assess whether structural metrics of cohesion and coupling correlate with size and churn. In terms of size metrics, we used LOC, number of packages, number of files and number of dependencies. For churn, we used number of files changed, number of lines changed and number of hunks. We employed the Kendall-τ correlation test [42], and the correlation coefficients were interpreted as proposed by Cohen [43]. For all systems under study, structural metrics presented either *no* or *small* correlation to both size and churn metrics, where most of correlation coefficients lie below 0.3. An exception was observed when considering structural coupling and number of dependencies, where the correlation coefficients for these metrics varied from 0.65 to 0.7 between systems. This correlation was expected as structural coupling is directly computed from dependencies. Nevertheless, structural coupling performs a qualified assessment of the system's structural cohesion as it evaluates not only the number of dependencies as it is but also how dependencies affect each other in an overall fashion.

IV. EXPERIMENTAL RESULTS

This section describes the results we found for each of our research questions.

A. RQ1: What are common intents when developers perform significant changes to the architecture?

Table IV reports the number of reviews identified under different intents for the 338 outliers. Most of the reviews that caused a significant change to the system's structural architecture were introducing a *new feature* to the system, followed by *enhancement*, *refactoring*, *bug fixing*, *feature removal* and *platform update*, respectively. An interesting observation is that most architecturally significant changes introduce a new feature, even though we have found weak correlation between the metrics we used for architectural change and metrics of size and churn (see Section III-F). This is expected because new code usually has dependencies to existing code, which affects the structural architecture of the system, where changes that add/modify several lines of code, but that do not affect the dependencies will have no effect in the architecture.

A surprising result is that 9% of architectural reviews are classified as bug fixing, as one would expect that bug fixing would not alter the system's architectural structure. After an in depth analysis, we noticed that the majority of bugs being fixed in these reviews are bugs that affect the behaviour of the system, instead of bugs that simply cause an error or throw an exception. For these kind of bugs, developers had to rework the code so that the system would exhibit the correct behaviour, which in turn would result in significant architectural changes.

We found few reviews that performed a feature removal or a platform update in comparison to the other intents. In fact, only 4% and 3% of architecturally significant changes updated the platform or removed a feature, respectively. Note that we found a non-negligible number of reviews where we could not infer the intent behind the change. In most of the not clear changes, the review had a very short description and no comments from which we could infer the intent.

When considering the most common intents behind the architectural changes, i.e. new feature, enhancement, refactoring and bug fixing, we noticed that 28% of the reviews have more than one intent. This is also an expected finding since architectural changes are usually large and touch several files at once. Figure IV-A presents the number of reviews for each of the most common intents, including the number of reviews that share more than one intent. The biggest intersection occurs between new feature and enhancement. This happens due to the incremental nature of software development, where a system is developed in an iterative fashion, and existing features are improved by small increments of new functionality. According to our manual inspection, 72% of the reviews that enhance an existing feature are doing so by introducing new features, and 39% of reviews introducing a new feature also have the intent of enhancing an existing feature.

As an answer to RQ1, we found that new feature, enhancement, refactoring and bug fixing are the most common intents behind architectural changes, accounting for 90% of the significant architectural changes we collected and inspected. Moreover, 28% of these changes have more than one intent,

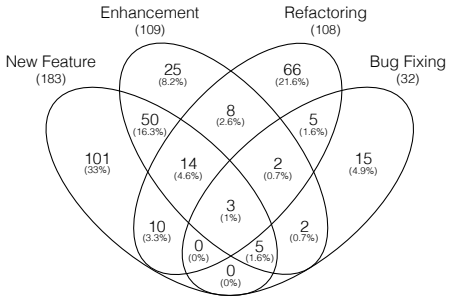

Fig. 2: Classification of reviews with significant architecture changes for each of the most common intents.

and 72% of changes enhancing an existing feature do so by adding a new feature.

B. RQ2: How often are developers aware of the architectural impact of their changes on a day-to-day basis?

Considering the intents behind architectural changes described in RQ1, Table V reports the number of reviews with different levels of architectural awareness according to our inspection and classification. Reviews for which the intent is not clear were left out of the analysis. In total, the number of reviews where the architecture is never discussed is higher than the number of reviews where the architecture is discussed in the description, comments or both. This indicates a substantial lack of architectural awareness from developers when performing changes with significant impact to the system's architecture.

For reviews where developers are adding a new feature, only in 10%, 17% and 7% of the time the architecture was discussed in the description, comments or both, respectively. Considering enhancements of existing feature, the architecture was discussed 15% of the time in the description, 16% of the time in comments and 9% of the time in both. Given that these are the most common intents when developers are performing architectural changes (see RQ1), these results point to an alarming lack of architectural awareness from developers during the changes where the architectural impact is the greatest. Finally, for all 338 architecturally significant reviews, we could find evidence of architectural awareness in the reviews' description, comments and both in only 18%, 12% and 8% of the reviews, respectively.

For the reviews which performed a refactoring to the system, the total number of reviews where the architecture is discussed either in the description, comments or both is higher than the number of reviews where the architecture is not discussed. Developers were aware of the architectural impact of their refactorings in 65% of the cases. We noticed that most of the reviews with a refactoring intent but no architectural awareness were removing dead code. Dead code removal is indicated as an architecturally significant change because of the amount of apparent static dependencies usually removed by such operations. However, this is a straightforward operation in which its impact to the system as a whole is minimum and only apparent dependencies are removed, by definition.

As an answer to RQ2, by inspecting and classifying 338 reviews that performed significant architectural changes, we

TABLE IV: NUMBER OF REVIEWS THAT PERFORMED ARCHITECTURALLY SIGNIFICANT CHANGES GROUPED BY DIFFERENT INTENTS.

Systems	New Feature	Feature Enhancement	Feature Removal	Platform Update	Refactoring	Bug Fixing	Not Clear
egit	76 (62%)	39 (32%)	5 (4%)	4 (3%)	27 (22%)	14 (11%)	6 (5%)
linuxtools	70 (43%)	57 (35%)	7 (4%)	5 (3%)	69 (42%)	15 (9%)	9 (5%)
java-client	22 (81%)	11 (41%)	1 (4%)	1 (4%)	6 (22%)	1 (4%)	0 (0%)
jvm-core	15 (60%)	2 (8%)	0 (0%)	1 (4%)	6 (24%)	2 (8%)	3 (12%)
All Systems	183 (54%)	109 (32%)	13 (4%)	11 (3%)	108 (32%)	32 (9%)	18 (5%)

TABLE V: NUMBER OF REVIEWS, FOR EACH INTENT, WHERE THE ARCHITECTURE IS NOT DISCUSSED, IS DISCUSSED ONLY IN THE REVIEW'S DESCRIPTION, ONLY IN ITS COMMENTS, OR IN BOTH.

Intent	Discussion (Awareness)			
	None	Description	Comments	Both
New Feature	120 (66%)	18 (10%)	32 (17%)	13 (7%)
Enhancement	64 (59%)	17 (15%)	18 (16%)	10 (9%)
Feature Removal	9 (69%)	3 (23%)	1 (8%)	0 (0%)
Platform Update	4 (36%)	5 (45%)	2 (18%)	0 (0%)
Refactoring	37 (34%)	43 (40%)	8 (7%)	20 (18%)
Bug Fixing	24 (75%)	6 (19%)	2 (6%)	0 (0%)
Total	211 (62%)	60 (18%)	41 (12%)	26 (8%)

found that developers were aware of the impact of their change in only 38% of the time. Although being one of the most common intents when performing architectural changes, reviews that add a new feature or enhance an existing feature present a poor level of architectural awareness. Finally, developers present a high level of awareness when refactoring the systems, where the architecture is discussed in the reviews' description, comments or both in 65% of the cases.

C. RQ3: How does awareness and intent influence architectural changes on a day-to-day basis?

Table VI reports the number of reviews that either improved or degraded the cohesion and coupling of each system under study for different intents. In RQ1 we showed that there is a considerable overlap of reviews introducing a new feature and reviews enhancing existing features. Therefore, since both these intents are concerned with augmenting and improving the system's features, we combined these two intents under *Feature* in Table VI. Finally, we consider under *Awareness* all reviews in which the structural architecture was discussed in the review's description or comments (as absolute numbers and as percentage of the total number of reviews).

Consider the coupling degradation of `egit`, for example. When the intent was to add a new feature and/or enhance a feature, we found 57 reviews where the change led to a degradation of either the overall coupling of the system or the coupling of a single package. For 6 reviews, corresponding to 10%, the architecture was discussed during the review. Similarly, we identified a total of 24 reviews that improved the coupling of `linuxtools` through refactoring. However, in only 14 (59%) of these the architecture was discussed.

When one compares the reviews that improve cohesion/coupling to the reviews that degrade cohesion/coupling, developers tend to be more aware of the architectural impact when their changes have a positive effect. This indicates that developers are often not aware of the impact of their changes when they are degrading the structural architecture. Moreover, most of the reviews identified as performing significant archiectural changes caused a degradation in the systems' structural cohesion and coupling. This is arguably the moment which developers should be mostly aware of the architectural impact of their changes since poor architectural decisions might lead to bug proneness [3] and increased maintenance effort [4].

Considering only the reviews in which a Refactoring was performed, this behaviour is not so pronounced. Based on our inspection, developers tend to have a similar level of awareness when the cohesion/coupling of the system is both improved and degraded. As an example, we found that developers of `linuxtools` are aware of the architectural impact in 61% and 62% of the refactorings that improved and degraded the system's cohesion, respectively. This is a counterintuitive finding as one expects that refactorings should lead to improvements instead of degradations.

In order to shed light into this issue, we present two examples of refactorings in which developers were aware of the architecture, but the change resulted in a degradation of the system's structural architecture. In review 23478 of `linuxtools`, the author of the change describes the refactoring as *"Internalize remaining classes. They were not exported so not a real change"*, which clearly indicates an attempt of structural improvement. For this purpose, a set of 9 files were moved from their original package to an internal package. As a result, the cohesion of the original package was improved, but the amount of dependencies the files had to other files in the system caused a degradation of overall coupling and overall cohesion. This is an example of a developer who, although being aware of the system's structural architecture and having the intent of refactoring it, failed to do so. This is an indication that even with the intent of improving the system's architecture, developers sometimes are not able to see all the ramifications of their architectural changes.

Review 11670 of `linuxtools` in another example, where the author described the refactoring as *"Move input validators to where they are actually used"*. In this case, the developer is also performing an improvement to the code base, but this time the reasoning behind the change seems to be semantical rather than structural. Hence, the author moved two `Validators` files from the `validators` package to a package with files that use the the `Validators`' functionality. Due to the number of dependencies the `Validators` have to their original package, this change caused a degradation in both the structural cohesion

TABLE VI: NUMBER OF REVIEWS THAT EITHER IMPROVED OR DEGRADED THE SYSTEMS' COHESION AND COUPLING FOR DIFFERENT INTENTS AND THE SUBSET OF REVIEWS WITH SIGNS OF AWARENESS.

System	Intent	Cohesion						Coupling					
		Improvement			Degradation			Improvement			Degradation		
		Total	Awareness		Total	Awareness		Total	Awareness		Total	Awareness	
egit	**Feature**	10	6	60%	30	40	13%	9	5	55%	57	6	10%
	Refactoring	6	4	66%	7	5	71%	7	5	71%	13	6	46%
	Bug Fixing	0	0	0%	5	1	20%	2	1	50%	9	2	22%
linuxtools	**Feature**	12	7	58%	28	15	53%	26	15	57%	52	26	50%
	Refactoring	18	11	61%	16	10	62%	24	14	58%	32	25	78%
	Bug Fixing	2	2	100%	4	0	0%	5	3	60%	5	0	0%
java-client	**Feature**	4	3	75%	13	5	38%	3	2	66%	15	9	60%
	Refactoring	2	2	100%	1	1	100%	1	1	100%	4	4	100%
	Bug Fixing	1	0	0%	0	0	—	0	0	—	0	0	—
jvm-core	**Feature**	0	0	—	6	2	33%	0	0	—	12	2	16%
	Refactoring	0	0	—	2	2	100%	1	0	0%	4	3	75%
	Bug Fixing	0	0	—	0	0	—	1	0	0%	1	1	100%

and structural coupling of the system. With this example, we provide evidence that developers consider not only structural cohesion and coupling, but also other aspects when carrying out architectural changes. Such observation is aligned with findings reported in previous empirical studies [17], [18].

In order to assess the effect that architectural awareness has on the improvement and degradation of structural cohesion and coupling, we report in Figure 3 the distribution of cohesion and coupling for reviews we found evidence of architectural awareness and for reviews where we did not. We report box-plots for both `egit` and `linuxtools` because these are the systems with the largest number of architecturally significant reviews. For each system, we computed 8 box-plots. First, we report the distribution of cohesion and coupling for the reviews that improved or degraded the overall cohesion and coupling of the system. Next, we report cohesion and coupling for the reviews that improved or degraded the cohesion and coupling of a single package in the system. In all box-plots, smaller values of cohesion and coupling are more desirable for the system's structural architecture.

Consider the box-plots that depict the distribution of cohesion and coupling for the reviews that improved either the system's overall cohesion/coupling or the cohesion/coupling of a single package. In every case (a, b, e, f, i, j, m, n), the reviews in which the architecture was discussed presented larger improvements in structural cohesion and coupling. When looking at `egit` in particular, reviews with evidence of architectural discussion presented considerably larger improvements to the system's overall cohesion and overall coupling, as can be seen in boxplots (a) and (b), respectively.

When considering the reviews that degraded the system's cohesion and coupling, we found two cases in which the reviews with evidence of architectural discussion caused less degradation than reviews in which the architecture was not discussed. In the first one (g), reviews with architectural discussion caused less degradation to single packages of `egit` than their counterparts with no architectural discussion. A similar behaviour was identified for the reviews that degraded the overall cohesion of `linuxtools`. However, this did not replicate to the other cases, where both reviews with and reviews without architectural discussion had a similar degradation in cohesion and coupling.

The observations from the box-plots provide evidence that architectural awareness has a positive effect in the cohesion and coupling of the systems for the reviews in which the structural architecture was improved. However, for reviews that degrade the system's architecture, apart from specific cases, architectural awareness does not have a noticeable effect in the actual degradation caused by the review.

In summary, we found that the architecture is more often discussed in the reviews that improve the cohesion and coupling of the system when compared to reviews that degrade the cohesion/coupling. Differently, the architecture is similarly discussed in reviews that perform a refactoring. Moreover, we found evidence that refactorings that degrade the systems' cohesion/coupling are not caused only by poor implementation, but also by the fact that developers may consider other properties when performing refactorings. Finally, by assessing the distribution of cohesion and coupling of the reviews we studied, we noticed that reviews in which we found evidence of architectural awareness tend to present larger improvements in cohesion and coupling when compared to reviews where the architecture was not discussed.

As an answer to RQ3, architectural awareness is mostly found in reviews that improve the system's architecture, where the architecture discussion often leads to larger improvements in cohesion and coupling in these reviews.

The results for RQ1–3 lead to actionable findings for tool builders, practitioners and researchers. We have observed that architectural awareness during code review leads to better architectural changes. Thus, tool builders should provide plugins for code review systems that will automatically identify significant architectural changes and make the reviewers aware of the architectural impact of the change. Practitioners should understand the importance of discussing the system's architecture when performing architectural changes, and more importantly, adopt tools to assist in this dicussion. Finally, researchers can use code review not only to study architectural changes, but also as technique to perform architectural improvements and prevent architectural degradation.

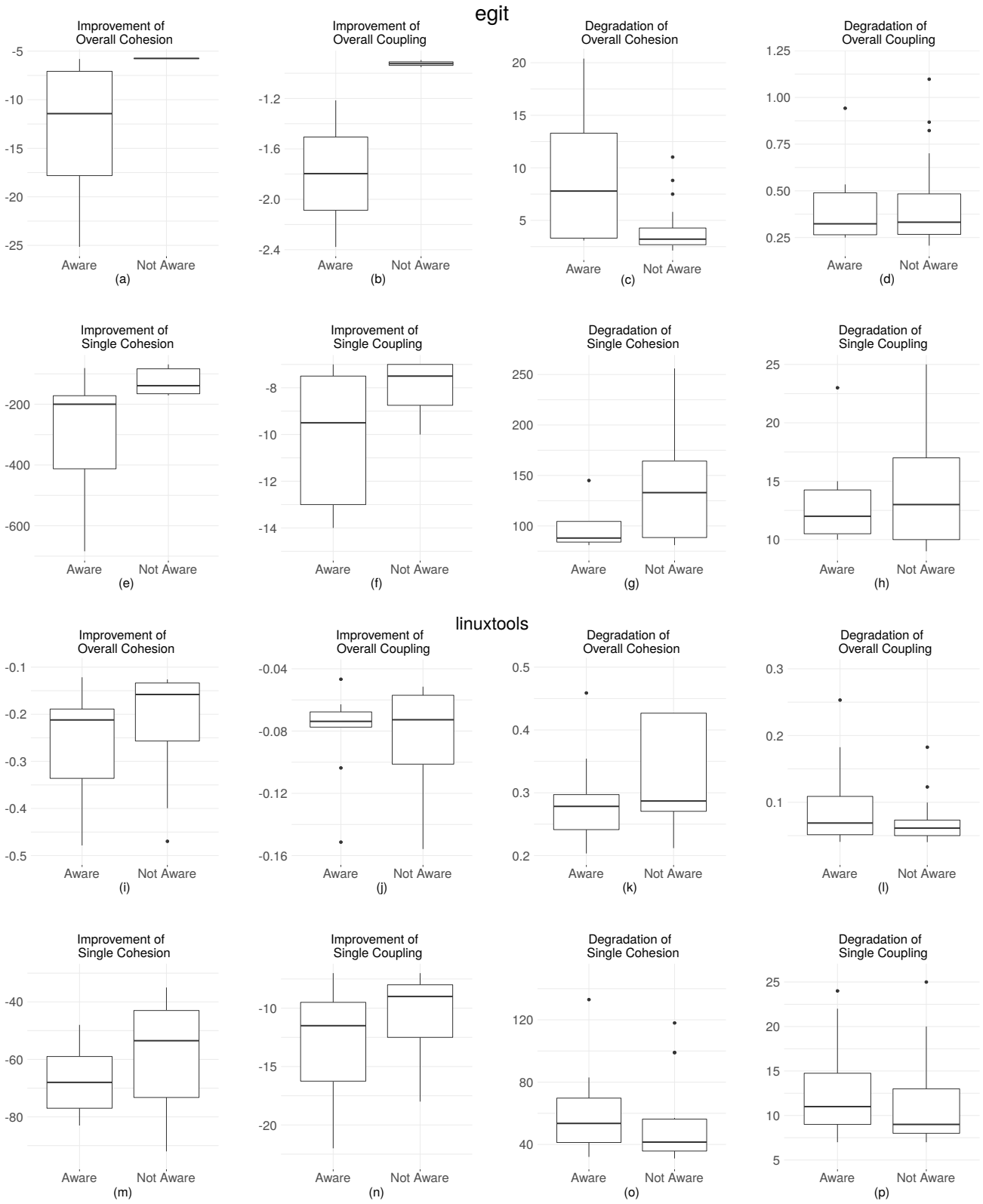

Fig. 3: Distribution of cohesion and coupling for reviews where we found evidence of architectural awareness and for reviews where we did not. We report box-plots for the reviews that improve and degrade the overall cohesion/coupling of the system and also for the reviews that improve and degrade the cohesion/coupling of a single package in the system.

V. Threats to the Validity

Internal validity: We use a metric-based approach to automatically identify reviews that performed significant changes to the system's structural architecture. Using this approach, one cannot guarantee all architecturally significant reviews were inspected. To alleviate this threat, we performed a pre-study in which we inspected all reviews from the Couchbase systems that presented any change to structural cohesion and coupling. With statistically significance at the 0.1 α confidence level, we showed that the number of reviews discussing the architectural impact among the reviews with significant changes is higher than among those with negligible changes.

The metrics of structural cohesion and coupling we used are based on structural dependencies between files, in which differences in size might affect the cohesion and coupling measurement. In our pre-study, we collected size and churn metrics of all systems and performed a correlation analysis with the cohesion and coupling metrics we employed. Most of the correlation coefficients were identified as low or medium, which is aligned with what is usually expected from object-oriented metrics computed from source code [44]. The low and medium correlation indicates that the cohesion and coupling metrics we employed are indeed capturing changes in structural architecture of the system and not only size fluctuations.

Manual classifications are naturally subjective to bias. To mitigate this threat, we employed a two-stage manual classification procedure. In the first stage, all reviews were separately classified by two authors following a strict guideline previously discussed and agreed by all authors. In the second stage, for all reviews in which a disagreement was found, both authors discussed the review until a unified classification was reached.

External validity: Our study focuses on four Java projects which we carefully chose so that we could study them in detail both quantitatively and qualitatively. However, the results may not be generalisable to projects in other languages.

The analysis of the systems' architecture is based on structural metrics of cohesion and coupling. One might expect different results using different metrics. However, we rely on structural cohesion and coupling since they are widely-adopted for architecture analysis and have been thoroughly evaluated in previous studies [18], [20], [26].

VI. Related Work

Tufano et al. [41] performed an empirical study to understand the lifecycle of code smells in software projects. They manually inspected and classified commits in regard to commit goal, project status, and developer status. While their classification is mostly based on commit messages and patches, the code review process adopted in our analysis provides a richer set of artefacts for each software change. Besides having access to each commit and patch, a review also includes feedback provided by other developers, and often links to tickets in the issue tracking system and links to related reviews performed in the past. As such, during our manual inspection, we extend Tufano et al.'s classification of the commit goal to include a wider set of intents we found during our open coding analysis.

Several studies have been performed to qualitatively evaluate the developer's perception of cohesion and coupling metrics. Simons et al. [27] prepared a set of toy examples and surveyed developers to assess whether metrics represent the developer's perception of quality. Bavota et al. [17] and Candela et al. [18] also surveyed developers with the same purpose, where in this case the questionnaire was focused on selected past changes. By inspecting code reviews, we are able to assess developers intent and awareness on a day-to-day basis, focusing on how developers perceive the architectural changes at the time these changes are being reviewed. As a result, we can study the developers' behaviour for each different architectural change in particular, without the bias of interviews that involve toy systems or past changes.

Recent studies have evaluated different metrics of structural cohesion and coupling as suitable measurements for architectural quality. In a context of search based software modularisation, Paixao et al. [20] compared the modularisation developers implemented in their systems against baselines generated by different search procedures. In a similar setting, Ó Cinnéide et al. [26] evaluated a set of structural cohesion metrics for automated refactoring. Although providing quantitative evidence on how structural cohesion measurement can be used to improve software systems, these work lack a qualitative analysis to better understand how developers perform architectural changes on a day-to-day basis.

VII. Conclusion

Architectural decisions have large implications on the development and evolution of software systems. In this context, a better understanding of how developers perform architectural changes on a day-to-day basis is required as a foundation for the improvement of automated decision support tools.

Thus, we performed an empirical study that involved the inspection and classification of 628 architectural changes mined from 4 software systems. We limited our study to reviews with good quality description and feedback provided by developers. After analysing 338 reviews that performed significant changes to the system's structural architecture, we found that the architecture is only discussed in 38% of the reviews we studied, which indicates a lack of architectural awareness when performing significant architectural changes. Nevertheless, developers tend to be more often aware of the architecture when the change is actually improving the system in terms of cohesion and coupling.

Finally, we noticed that changes in which developers are aware of the architectural impact tend to present larger improvements in cohesion and coupling than changes where the architecture is not discussed. Such observation indicates that architectural awareness during code review has the potential to aid developers on their day-to-day activities. Hence, we lay as future work the automation of architectural analysis during code review time, where authors and reviewers would automatically be made aware of the architectural impact of their changes.

REFERENCES

[1] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, 2nd ed. Addison Wesley, 2011.

[2] B. J. Williams and J. C. Carver, "Characterizing software architecture changes: A systematic review," *Information and Software Technology*, vol. 52, no. 1, pp. 31–51, Jan 2010.

[3] R. Schwanke, L. Xiao, and Y. Cai, "Measuring architecture quality by structure plus history analysis," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE, May 2013, pp. 891–900.

[4] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," in *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. New York, USA: ACM Press, 2016, pp. 488–498.

[5] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111–122, Nov 1993.

[6] Y. Zhou and H. Leung, "Predicting object-oriented software maintainability using multivariate adaptive regression splines," *Journal of Systems and Software*, vol. 80, no. 8, pp. 1349–1361, Aug 2007.

[7] Zhifeng Yu and V. Rajlich, "Hidden dependencies in program comprehension and change propagation," in *Proceedings of the 9th International Workshop on Program Comprehension (IWPC '01)*. IEEE, 2001, pp. 293–299.

[8] S. Counsell, S. Swift, and A. Tucker, "Object-oriented cohesion as a surrogate of software comprehension: an empirical study," in *Proceedings of the 5th International Workshop on Source Code Analysis and Manipulation (SCAM '05)*. IEEE, 2005, pp. 161–172.

[9] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[10] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[11] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," in *Proceedings of International Symposium on Applied Corporate Computing*, 1995.

[12] L. Briand, J. Daly, and J. Wust, "A unified framework for cohesion measurement in object-oriented systems," in *Proceedings of the 4th International Software Metrics Symposium (Metrics '98)*, vol. 3. IEEE, 1998, pp. 43–53.

[13] L. Briand, D. J.W., and J. Wust, "A unified framework for coupling measurement in object-oriented systems," *Transactions on Software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.

[14] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98)*. IEEE, 1998, pp. 45–52.

[15] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, Feb 2009.

[16] F. Beck and S. Diehl, "On the impact of software evolution on software clustering," *Empirical Software Engineering*, vol. 18, no. 5, pp. 970–1004, Oct 2013.

[17] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. San Francisco, CA: IEEE, 2013, pp. 692–701.

[18] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization : Is it enough ?" *Transactions on Software Engineering and Methodology*, vol. 25, no. 3, pp. 1–28, 2016.

[19] B. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, Mar 2006.

[20] M. Paixao, M. Harman, Y. Zhang, and Y. Yu, "An empirical study of cohesion and coupling: Balancing optimisation and disruption," *IEEE Transactions on Evolutionary Computation*, pp. 1–21, 2017.

[21] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level," in *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Austin, Texas: ACM Press, 2016, pp. 499–510.

[22] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural change in open-source software systems," in *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE, May 2015, pp. 235–245.

[23] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, Jan 2012.

[24] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '15)*. Bergamo, Italy: ACM Press, 2015, pp. 50–60.

[25] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, Nov 2012.

[26] M. Ó Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. Hemati Moghadam, "Experimental assessment of software metrics using automated refactoring," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM '12)*. Lund, Sweden: ACM Press, 2012, p. 49.

[27] C. Simons, J. Singer, and D. R. White, "Search-based refactoring: Metrics are not enough," in *Proceedings of the 7th International Symposium on Search Based Software Engineering (SSBSE '15)*, 2015, vol. 9275, pp. 47–61.

[28] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, Apr 2009.

[29] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE, May 2013, pp. 712–721.

[30] S. Pearce, "Gerrit code review for git," https://www.gerritcodereview.com, 2006, accessed in: April 2017.

[31] E. Projects, https://eclipse.org/projects, 2017, accessed in: April 2017.

[32] Couchbase Projects, https://developer.couchbase.com/open-source-projects, 2017, accessed in: April 2017.

[33] M. Hall, M. A. Khojaye, N. Walkinshaw, and P. McMinn, "Establishing the source code disruption caused by automated remodularisation tools," in *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 466–470.

[34] F. Beck and S. Diehl, "On the congruence of modularity and code coupling," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (SIGSOFT/FSE '11)*. Szeged, Hungary: ACM Press, 2011.

[35] M. Hall, N. Walkinshaw, and P. McMinn, "Supervised software modularisation," in *Proceedings of the 28th International Conference on Software Maintenance (ICSM '12)*. IEEE, 2012, pp. 472–481.

[36] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*. IEEE, 2013, pp. 901–910.

[37] M. d. O. Barros, F. d. A. Farzat, and G. H. Travassos, "Learning from optimization: A case study with apache ant," *Information and Software Technology*, vol. 57, no. 1, pp. 684–704, Jan 2015.

[38] Scitools, https://scitools.com/features, 2017, accessed in: April 2017.

[39] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman. Supporting material for: Are developers aware of the architectural impact of their changes? [Online]. Available: https://mhepaixao.github.io/architecture_awareness/

[40] J. W. Tukey, *Exploratory data analysis*. Addison-Wesely, 1977.

[41] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *Transactions on Software Engineering*, vol. 1, pp. 1–27, May 2017.

[42] M. G. Kendall, *Rank correlation methods*. Griffin, 1948.

[43] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Routledge, 1988.

[44] K. El Emam, S. Benlarbi, N. Goel, and S. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *Transactions on Software Engineering*, vol. 27, no. 7, pp. 630–650, Jul 2001.