The Impact of Code Review on Architectural Changes

Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman

Abstract—Although considered one of the most important decisions in the software development lifecycle, empirical evidence on how developers perform and perceive architectural changes remains scarce. Architectural decisions have far-reaching consequences yet, we know relatively little about the level of developers' awareness of their changes' impact on the software's architecture. We also know little about whether architecture-related discussions between developers lead to better architectural changes. To provide a better understanding of these questions, we use the code review data from 7 open source systems to investigate developers' intent and awareness when performing changes alongside the evolution of the changes during the reviewing process. We extracted the code base of 18,400 reviews and 51,889 revisions. 4,171 of the reviews have changes in their computed architectural metrics, and 731 present significant changes to the architecture. We manually inspected all reviews that caused significant changes and found that developers are discussing the impact of their changes on the architectural structure in only 31% of the cases, suggesting a lack of awareness. Moreover, we noticed that in 73% of the cases in which developers provided architectural feedback during code review, the comments were addressed, where the final merged revision tended to exhibit higher architectural improvement than reviews in which the system's structure is not discussed.

Index Terms—Software Architecture, Code Reviews, Empirical Software Engineering

1 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 architectural design [2]. Recent studies have empirically shown that bug-prone files are more architecturally connected than clean files [3], [4], and that architectural flaws can lead to increased maintenance effort [5].

The notions of cohesion and coupling as guides for software architecture design have been extensively associated with different aspects of software quality, such as, maintainability [6], [7], comprehensibility [8], [9] and code smells [10], [11]. Structural dependencies between code components were the most used assets for cohesion and coupling measurement for many years [12]–[15], where other sources of information have been taken into account more recently, such as semantics [16] and revision history [17], [18]. Nevertheless, recent studies [19], [20] have revealed structural dependencies to be one of the best proxies for developers' perception of cohesion and coupling.

The structural dependencies between source code files alongside the organisation of the code base in its directory structure (or package structure for Java systems) represent the structural view of a software's architecture. In both the original 4+1 architectural model proposed by Kruchten [21] and in the seminal book by Rozanski and Wood [1], the authors indicate that the architecture of a software system cannot be expressed by a single artefact or diagram. Hence, a system's architecture is composed of different views and perspectives, each of which is tailored to a different subset of stakeholders [1]. In this context, the structural view of the architecture is the one with which developers interact most. Moreover, this is the architectural view that practitioners commonly use as the groundwork for the design of the other architectural views [22]. Hence, in this paper, we focus our analysis on the structural view of the software's architecture. In other words, the architectural discussions presented in this paper refer to the structural organisation of source code in its directory structure and the dependencies between source code files.

Despite the large body of work aimed at aiding developers in the structural organisation of systems [23]–[25], we still see evidence of structural erosion as systems evolve [26], [27]. Developers sometimes *choose* to accept suboptimal solutions in order to achieve a desired goal, such as short-term delivery [28]; thereby accruing technical debt [29]. 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 [19], [20], 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 a better *understanding* of how developers deal with structural changes.

In this paper, we extend the body of empirical knowledge regarding structural 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

- *Dr. Matheus Paixao is a research assistant at State University of Ceara, Brazil. Corresponding e-mail: matheus.paixao@uece.br*
- *Dr. Jens Krinke and DongGyun Han are members of the Centre for Research in Search, Testing and Evolution (CREST) at University College London*
- *Dr. Chaiyong Ragkhitwetsagul is a lecturer at the Faculty of Information and Communication Technology, Mahidol University.*
- *Prof. Mark Harman is a professor at the Centre for Research in Search, Testing and Evolution (CREST) at University College London and an Engineering Manager at Facebook London*

Manuscript received xxx xx, xxxx; revised xxx xx, xxxx.

system's structural architecture. Moreover, we also assess whether developers are *aware* of the architectural impact of their changes at the time these changes are being made. Finally, we study how architectural changes evolve between the first proposed version of the change until the last version of the change that is merged into the system's repository.

Quantitative studies evaluating metrics and techniques for structural optimisation [23], [24], [30] 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 [31] or selected past changes [19], [20]. Since surveys are subjective to bias [32] and the questionnaires usually target the software system as a whole, such studies fail to capture details and nuances of each particular architectural change.

In order to investigate the developers' intent and awareness when performing architectural changes alongside the evolution of these architectural changes on a day-to-day basis, we used code review data. During the process of code review, a change is only incorporated into the system after an inspection of the patch (code change) 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. Depending on the feedback from the reviewers, the author of the change may need to improve the patch. In these cases, the author submits new revisions to the patch until the change is incorporated into the system or it is discarded.

The code review process provides detailed information about each change and each revision, which enables us to perform the empirical study on which we report here. In this paper, we adopt CROP [33], a recently published open source code review dataset that links code review data to complete versions of software systems at the time of review.

In CROP, we provide data for all code reviews and all revisions of a certain software system. Thus, for each change and each revision, 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 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. Moreover, by studying the different revisions of architectural changes during code review, we can investigate how changes that impact the system's architecture evolve from when they are first proposed to when they were finally merged. We anonymised and protected the developers' names and identities in CROP to the best of our ability. Moreover, all code included in CROP retains its original license and distribution policies.

After analysing a total of 18,400 code reviews and 51,889 revisions from 7 software systems, we used a metric-based approach to identify reviews that changed the structural architecture of the systems. For 731 reviews that significantly changed the architecture, 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).

As well as a framework for the identification of significant architectural changes, this paper made the following specific contributions:

1) We found that developers discuss the architectural impact of their changes in only 31% of the reviews with a noticeable impact on the system's architecture. In addition, reviews in which the architecture is discussed tend to have higher architectural improvement than reviews in which the system's structure is not discussed.
2) When considering the reviews in which we identified architectural discussion, we found that the architectural quality of the patch was decreased in 33% of the cases in which developers provided architectural feedback via comments during the reviewing process.
3) A dataset of 1,139 manually classified code reviews that include the intent of each review and the architectural awareness of developers involved in each review.
4) A dataset of 103,778 structural architectures extracted from the source code of 7 open source software systems.

As this work in an extended version of our conference paper [34], we present the primary novel contributions of this extension as follows.

Expansion of evaluation corpus: In our previous work, only the last merged revision was considered in the empirical study, while for this paper we included in the analysis versions of the system for all revisions submitted during the reviewing process. We now consider 7 open source systems in our study instead of the 4 previously studied. In total, these extensions meant we studied 9,500 more code reviews and 42,989 more revisions than our previous paper.

Sensitivity analysis for threshold selection: Our method for identification of reviews that caused significant changes to the architecture is based on an outlier detection procedure. Instead of selecting the default threshold, as we have done in the previous submission, we now perform a sensitivity analysis to select the best-suited threshold for the study.

The evolution of architectural changes: As previously mentioned, we have now collected data for all revisions submitted during code review instead of only the last merged revision (as in our previous work). This allowed us to ask a new research question: *How do architectural changes evolve during code review?*.

Qualitative analysis of negative refactorings: During the empirical study, we observed cases in which refactorings caused a degradation to the system's architecture. In this paper, we describe a qualitative analysis in which we investigated in details the causes of such phenomena.

2 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

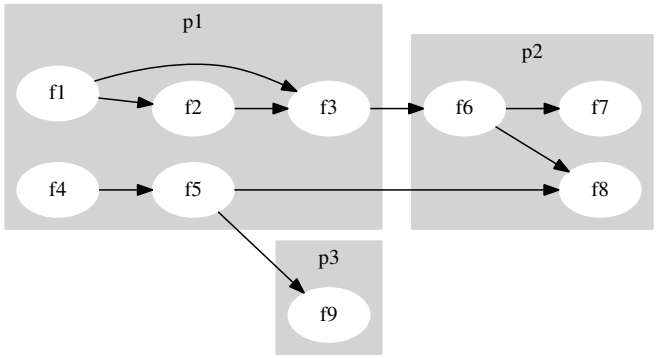


Fig. 1: Example of a Module Dependency Graph that is used to represent the structural modularisation of the Java systems under study. Nodes represent files and edges represent dependencies between files. Clusters of files (grey regions) indicate packages.

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) [15].

Figure 1 shows an example of an MDG. An MDG is a graph $G(F, D)$ where the set of nodes F represents the files in the system and D represents the dependencies between files. The clusters of files in the MDG are indicated by the grey areas in the figure and they represent the modules in the system (i.e. packages in Java systems).

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 structural metrics for cohesion and coupling measurement that have been quantitatively and qualitatively evaluated in a recent study [20].

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.

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

RQ4: *How do architectural changes evolve during code review?* By comparing the last merged revision to all the other previous revisions of a certain architectural change, we study how the code review process influences the evolution of changes that cause a significant impact on the system's architecture.

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

3.1 Code Review Data

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 [35], one of the most popular code review systems currently in use by large open source communities, such as Eclipse [36] and Couchbase [37]. 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 (code change) 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 in the form of revisions according to the feedback until the review is *merged* or *abandoned*, where the first indicates the code change was incorporated to the system and the latter indicates the code change was rejected. For the rest of this paper, we use review and (code) change interchangeably to indicate a code submission that was manually inspected by developers and later merged or abandoned. In addition, we use revisions to indicate intermediate patches submitted during the reviewing process of a single review according to the feedback from other developers.

In this paper, we make use of CROP [33], an open source dataset that links code review data with their respective software changes. We designed CROP as an extended and more comprehensive version of the dataset we employed in our previous paper [34]. Given a software system, CROP provides a complete reviewing history that includes not only the code review data such as descriptions and comments from developers, but also versions of the code base that represent the software system at the time of review. In CROP, we collected the Gerrit code review data from both Eclipse and Couchbase communities. For each of these communities, CROP provides data for the software systems with most reviewed changes. We made CROP publicly available to support other researchers, where we carefully handled developers anonymisation and software license compliance to ensure the CROP dataset meets data protection and licensing policies. For the interested reader, we recommend CROP's website[1] and original paper [33] for additional information on the dataset.

For this particular paper, we adopt all the Java systems included in the CROP dataset. For the Eclipse community, we study egit, jgit, linuxtools and platform.ui. For the Couchbase community, we adopt couchbase-java-client, couchbase-jvm-core and spymemcached. For brevity, the Couchbase systems will be abbreviated as java-client and jvm-core, respectively.

The consideration of these 7 systems yielded a manual inspection and classification of 731 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. Table 1 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 have changed throughout their history, we additionally report median, maximum and minimum values for these statistics.

Both egit and jgit are aimed at providing git support in Eclipse. While jgit is a full Java implementation of the git version control system, egit integrates jgit into the Eclipse IDE. Linuxtools provides a C/C++ IDE for linux developers, and platform.ui provides the basic building blocks for user interfaces built with Eclipse.

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 jvm-core is a low-level API mostly used by java-client. Spymemcached is a lightweight Java implementation of a memory caching system that later became the groundwork for the development of java-client.

1. https://crop-repo.github.io

3.2 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 each review and revision that have undergone a process of code review as described in Section 3.1, where the formal definitions of the metrics being computed are presented in Section 2. All the metrics considered in this paper were recently validated by developer studies [19], [20], in which they were found to be good proxies for the developers' perceptions of cohesion and coupling.

The computation of the difference in structural cohesion and coupling for all code reviews and revisions we collected is depicted in the first steps of the framework presented in Figure 2. For each submitted revision, we use CROP to access the versions of the system before and after the revision took place, guaranteeing that the observed difference between them was solely induced by the code change in the revision.

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, 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 versions before and after the revision. Previous studies that performed architectural analyses in Java systems relied on bytecode analysis for structural architecture extraction [20], [24], [38]–[42]. However, building and compiling the systems for each revision 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 [43], 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 revision, 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 revision. 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 before and after the revision, we store not only the 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 revision, where each of which indicates the difference in overall cohesion/coupling and the biggest

TABLE 1: Descriptive statistics for the systems under study. We report the number of merged reviews and revisions in each system followed by the time span of our investigation. In addition, we report the median, maximum and minimum values of size metrics.

Systems	No. of Reviews	No. of Revisions	Time Span	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	4,502	11,430	9/09 to 11/17	91	98	84	70	1071	16	59	81	19	641	839	137	2,720	4,017	356
jgit	4,463	11,891	10/09 to 11/17	99	99	98	84	114	34	47	71	19	776	990	338	5,650	7,304	1,965
platform.ui	3,802	12,005	2/13 to 11/17	98	99	98	460	472	453	393	404	380	4,386	4,520	4,265	31,237	32,375	30,593
linuxtools	3,695	10,892	6/12 to 11/17	90	93	85	170	205	89	346	434	214	1,776	2,197	1,082	6,473	8,773	3,310
java-client	798	2,394	11/11 to 11/17	100	100	97	9	29	0	16	45	3	184	467	10	704	1,898	14
jvm-core	785	2,184	4/14 to 11/17	100	100	100	13	24	1	43	55	17	328	457	70	1,317	2,093	204
spymemcached	383	1,098	5/10 to 7/17	98	99	98	10	13	7	14	17	11	192	235	133	917	1,113	606

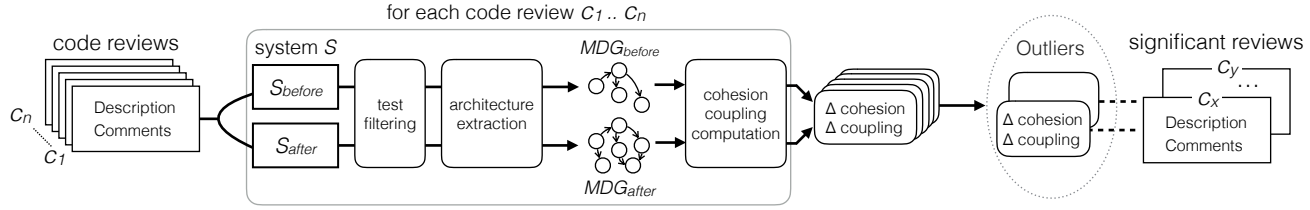


Fig. 2: Framework for the identification of code reviews with significant changes to the system's architecture. Given a set of code reviews, our automated framework identifies significant reviews in terms of the impact to system's architectural structure.

difference in cohesion/coupling for a single package, respectively. In this paper, we computed the differences in cohesion and coupling for 18,400 code reviews and 51,889 revisions, which generated a dataset of 103,778 structural architectures automatically extracted from source code. The data used as input to our analysis is publicly available [33]. We also make available all extracted structural architectures and the respective cohesion and coupling values computed for each revision [44].

3.3 Identification of Reviews with Significant Architectural Changes

A code review is formed by a collection of revisions that were sequentially submitted for review until the code change was merged or abandoned. In this context, the intermediate revisions of a certain code review can be seen as iterations of a code change that is not yet ready to be introduced in the code base. Hence, the final merged revision is the version of the code change that incorporates all the feedback from the reviewing process and represents the code review as a whole. Therefore, when identifying the code reviews that performed significant changes to the structural architecture of the system, we rely on the last merged revision of each code review.

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.

TABLE 2: Number of reviews identified as significant cohesion and coupling outliers according to Tukey's method'. 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	Coupling				Cohesion				Unique Outliers
	Overall		Single P.		Overall		Single P.		
	$\oplus$	$\ominus$	$\oplus$	$\ominus$	$\oplus$	$\ominus$	$\oplus$	$\ominus$	
egit	5	65	13	78	7	36	18	35	148
jgit	16	93	15	92	14	38	32	33	192
platform.ui	40	57	26	57	12	11	25	17	147
linuxtools	25	49	28	50	20	47	22	48	160
java-client	3	14	2	14	5	12	2	11	32
jvm-core	1	20	2	18	1	6	1	10	34
spymemcached	0	6	1	14	2	2	3	5	18
All	90	304	87	323	61	152	103	159	731

Next, for each of the 8 subsets, we identified the outliers using Tukey's method [45], and defined the 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 2 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.

As one can see from the table, 731 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 731 unique reviews identified as outliers consists of 17.5% of all reviews with

architectural change.

We motivate, discuss and validate this methodology for identification of reviews with significant architectural change in a study described in Section 4.

3.4 Manual Inspection and Classification of Reviews

Following the automated process described in the previous section, we considered all 731 outlier reviews, and performed a manual inspection and classification inspired by the work of Tufano et al. [46]. 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 first column of Table 3 presents the final set of tags used in the reviews' classification. A short description of each tag is presented in the second column of Table 3.

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 developers' 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 developers' awareness.

Our analysis is focused on reviews that performed significant changes to the system's structural architecture. In this case, when the author does 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 on the architecture. Similarly, if a reviewer does 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 the 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 a 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, two authors of the paper 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 review after the second stage of classification. Nevertheless, to make the classification process as transparent as possible, we present in the third column of Table 3 excerpts from the description and/or comments of a subset of code reviews as examples of relevant discussions we found during our study. Finally, the complete set of manually classified code reviews is available at our supporting webpage [44].

4 Validation of Experimental Design

In this section, we discuss and validate the experimental design we propose to study code reviews that performed significant changes to the structural architecture of software systems. We first evaluate whether the metrics we propose to measure the architectural change caused by a code review are appropriate. Next, we perform a sensitivity analysis on the effect of different thresholds when identifying the code reviews with significant architectural change.

4.1 Measurement of Architectural Change

We compare measurements of cohesion and coupling between versions of a software system to detect code reviews that caused a significant change to the system's structural architecture. Even though we employ metrics that have been recently proposed and validated by developers as good proxies for their perception of architectural quality [19], [20], measurements in object-oriented systems may be subjective to a size bias [47].

To alleviate and comprehend the size bias we might have in our evaluation corpus, we performed a correlation analysis between the cohesion and coupling metrics we employ and commonly used size and churn metrics. In particular, for each merged revision, we took the before and after versions of the system and measured the difference in the following size metrics: LOC, number of packages, number of files and number of dependencies.

Regarding churn metrics, we collected the number of changed files, number of changed lines and number of hunks for each merged revision. After collecting both the size and churn metrics, we noticed the data was comprised of many ties. Hence, we employed the Kendall-τ correlation test [48], which is designed to better deal with ties in the distribution (Spearman rank correlation can be adversely affected by ties, for example).

Finally, the correlation coefficients were interpreted as proposed by Cohen [49] in his seminal book. Cohen's book is the de facto guide and reference for statistical analysis not only in social sciences but also in many software engineering papers. Thus, the correlation analysis and interpretation were performed by following the original book and related software engineering papers. Nevertheless, we report all correlation coefficients in our supporting webpage [44] to allow for full replication and validation of our study and analysis.

For all systems under study, most of the structural metrics presented either *no* or *small* correlation to both size and churn metrics, where most of the correlation coefficients lie below 0.4. An exception was observed when considering structural coupling and number of dependencies, where the

TABLE 3: Tags used in the manual classification of code reviews. We first present the tag, followed by a short description of its meaning. Next we present excerpts from the description and/or comments of a subset of code reviews as examples of relevant discussions we found during our study.

	Intent of Change	
New Feature	Developer is adding a new feature to the system	*"Add option to replace selected files with version in the git index."*
Enhancement	Developer is enhancing an existing feature or code	*"DfsReftableDatabase is a new alternative for DfsRefDatabase that handles more operations (...)"*
Feature Removal	Developer is removing an obsolete feature	*"Retire org.eclipse.ui.examples.presentation plug-in"*
Platform Update	Developer is updating the code for a new platform/API	*"Bump to BREE 1.6 to be consistent"*
Refactoring	Developer is refactoring the system	*"Refactor View mapping into distinct class (...) query handling is moved into a separate class"*
Bug Fixing	Developer is fixing a bug	*"Fix failing unit tests introduced by (...)"*
Merge Commit	Developer is merging two branches	*"Merge branch stable-0.8"*
Not Clear	There's no evidence to suggest any of the previous	-
	Architectural Awareness	
In Description	Architectural impact is discussed in the description	*"Puts the code from IgnoreActionHandler into (...) and reuses it in the Staging view."*
In Comments	Architectural impact is discussed in the comments	*"Make this public (...) actually I might just say put it in the main JAR under the io.util package."*
Never	Architectural impact is never discussed	-

correlation coefficients for these metrics varied from 0.65 to 0.75 between the systems under study. This correlation was expected as structural coupling is directly computed from dependencies. Nevertheless, structural coupling performs a qualified assessment of the system's structural coupling as it evaluates not only the number of dependencies as it is but also how dependencies affect each other in an overall fashion. In a similar case, the number of files added and/or removed by a review tend do have a *medium* to *high* correlation with the cohesion of the system. Again, this correlation was also expected because the number of files in a package directly affects the computation of the system's cohesion. Likewise, the cohesion measurement we employ performs a qualified assessment of the relationship between files and dependencies in a package.

4.2 Threshold Sensitivity Analysis

We focus our analysis on the reviews with significant changes to the system's architecture as identified by the outliers over the distribution of the reviews' cohesion and coupling measurements (see Section 3.3). There exist many different techniques for outlier identification, each of which is better suited to a different scenario. For each code review in our dataset, we compute the impact of the code change in the system's cohesion and coupling. Since we need to identify the outliers for each metric separately (there might be a code review that greatly impacts coupling but only negligibly impacts cohesion), our dataset is composed of univariate data points. Moreover, the cohesion and coupling metrics do not follow a Gaussian distribution. Hence, we chose the Tukey's method for outlier identification, which is a widely accepted outlier detection technique for univariate data points with unknown distribution (non-parametric) [45].

The outlier identification in Tukey's method relies on creating a 'fence' that functions as a threshold to identify outliers on the distribution. The fence is computed as $\alpha \times$ IQR, where IQR stands for the interquartile range. In this scenario, the value attributed to α plays an important role in our experimental design, as it is the parameter that will define whether a review had a significant impact on the architecture or not.

The default configuration for Tukey's outlier identification is $\alpha = 1.5$ [45], which is the value we have adopted in our previous work [34]. However, different choices for the value of α would alter the threshold for the identification of significant architectural changes, which might thereby change the results of our research questions. Thus, for this paper, we performed a study to evaluate how sensitive our results are to different choices of values for α.

In order to perform this validation study, we need to manually inspect all the code reviews that impacted the metrics of cohesion and coupling for a certain system. This is necessary to build a ground truth of code reviews with architectural discussion so that we can evaluate different values of α. A complete analysis of all code reviews that affected the cohesion and coupling metrics for all systems we collected would yield an analysis of 4,721 code reviews, which is an infeasible task. Hence, we restricted this analysis to include only the systems from the Couchbase community, which resulted in an analysis of 492 code reviews. Thus, for each review in the Couchbase systems, we performed a manual classification regarding the intent of the review and the developers' architectural awareness, as described in Section 3.4.

After the classification, we configured our outlier identification method to employ different values of α, ranging from $\alpha = 2.0$ to $\alpha = 0.0$, with small decrements of 0.5. Next, we computed the ratio of code reviews in which the architecture is discussed for each subset of reviews identified as outliers for the different values of α. When considering the default setting ($\alpha = 1.5$), 84 code reviews were identified as performing significant architectural changes, out of which

developers discuss the architecture in 21 of them, accounting for a 25% architectural discussion ratio. For $\alpha = 2.0$, we identified 73 outliers with a 26% discussion ratio. Similarly, for the other values of α (1.0, 0.5, 0.0), the discussion ratio is 24%, 23% and 24%, respectively.

As one can see, the ratio of architectural discussion for the different subsets of outliers is consistent regardless of the value one may attribute to α. Since we have a different ratio of architectural discussion for the different subset of outliers, we employed a two-tailed pooled test to infer a statistical difference in the mean of architectural discussion between these different populations. Our statistical analysis did not detect any statistical difference (at the 0.01 significance level) in the discussion ratio between reviews identified by different α values. Given the observations we drew from this study, it is safe to assume that the results of our research questions are not likely to be affected by the threshold we select to identify significant architectural changes. Therefore, we choose to use the default configuration of Tukey's method ($\alpha = 1.5$) for the rest of this paper.

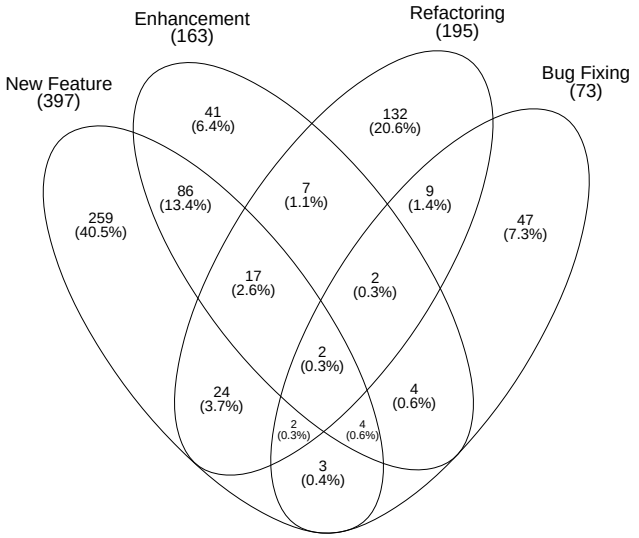


Fig. 3: Classification of reviews with significant architecture changes for each of the most common intents.

5 Experimental Results

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

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

Table 4 reports the number of reviews identified under different intents for the 731 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 *refactoring*, *enhancement*, *bug fixing*, *feature removal*, *merge commit* and *platform update*, respectively. An interesting observation is that most architecturally significant changes introduce a new feature, even though we have found a weak correlation between the metrics we used for architectural change and metrics of size and churn (see Section 4). 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 architecturally significant 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 this 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 6% and 1% of architecturally significant changes removed a feature or updated the platform, respectively. As one can see, platform.ui has a considerably higher number of reviews that perform a feature removal when compared to the other systems under study. We noticed that the developers of platform.ui tend to often move part of their modules to Github instead of having the source code in their own repository.

When considering the most common intents behind the architectural changes, i.e. new feature, enhancement, refactoring and bug fixing, we noticed that 22.5% 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 3 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, 67% of the reviews that enhance an existing feature are doing so by introducing new features, and 27% 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, refactoring, enhancement and bug fixing are the most common intents behind architectural changes, accounting for 87% of the significant architectural changes we collected and inspected. Moreover, 22.5% of these changes have more than one intent, and 67% of changes enhancing an existing feature do so by adding a new feature.

5.2 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 5 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

TABLE 4: Number of reviews that performed architecturally significant changes grouped by different intents.

Systems	New Feature	Feature Enhancement	Feature Removal	Platform Update	Refactoring	Bug Fixing	Merge Commit	Not Clear
egit	92 (62%)	43 (29%)	5 (3%)	4 (2%)	33 (22%)	19 (12%)	14 (9%)	0 (0%)
jgit	111 (57%)	31 (16%)	2 (1%)	1 (1%)	49 (25%)	5 (2%)	20 (10%)	0 (0%)
platform.ui	54 (36%)	24 (16%)	25 (17%)	1 (0%)	35 (23%)	32 (21%)	0 (0%)	0 (0%)
linuxtools	83 (51%)	44 (27%)	8 (5%)	3 (1%)	64 (40%)	11 (6%)	1 (1%)	6 (3%)
java-client	25 (78%)	10 (31%)	2 (6%)	1 (3%)	5 (15%)	2 (6%)	1 (3%)	0 (0%)
jvm-core	20 (58%)	7 (20%)	1 (2%)	1 (2%)	7 (20%)	2 (5%)	0 (0%)	3 (8%)
spymemcached	12 (66%)	4 (22%)	2 (11%)	0 (0%)	2 (11%)	2 (11%)	0 (0%)	1 (5%)
All Systems	397 (54%)	163 (22%)	45 (6%)	11 (1%)	195 (26%)	73 (9%)	36 (4%)	10 (1%)

TABLE 5: 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	297 (74%)	34 (8%)	48 (12%)	18 (4%)
Enhancement	116 (71%)	20 (12%)	17 (10%)	10 (6%)
Feature Removal	36 (80%)	8 (17%)	1 (2%)	0 (0%)
Updating Platform	4 (36%)	4 (36%)	3 (27%)	0 (0%)
Refactoring	92 (47%)	69 (35%)	11 (5%)	23 (11%)
Bug Fixing	60 (82%)	8 (10%)	4 (5%)	1 (1%)
Merge Commit	36 (100%)	0 (0%)	0 (0%)	0 (0%)
Total	641 (69%)	143 (15%)	84 (9%)	52 (5%)

indicates a substantial lack of architectural awareness from developers when performing changes with significant impact on the system's architecture.

For reviews where developers are adding a new feature, only in 8%, 12% and 4% of the time the architecture was discussed in the description, comments or both, respectively. Considering enhancements of an existing feature, the architecture was discussed 12% of the time in the description, 10% of the time in comments and 6% of the time in both. Given that these are among 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 731 architecturally significant reviews, we could find evidence of architectural awareness in the reviews' description, comments and both in only 15%, 9% and 5% 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 51% 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 on the system as a whole is minimum and only apparent dependencies are removed, by definition.

As an answer to RQ2, by inspecting and classifying 731 reviews that performed significant architectural changes, we found that developers were aware of the impact of their change in only 29% 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 higher level of awareness when refactoring the systems, where the architecture is discussed in the reviews' description, comments or both in 51% of the cases.

5.3 RQ3: How do awareness and intent influence architectural changes on a day-to-day basis?

Table 6 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 6. 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 78 reviews where the change led to a degradation of either the overall coupling of the system or the coupling of a single package. For 10 reviews, corresponding to 12%, the architecture was discussed during the review. Similarly, we identified a total of 24 reviews that improved the cohesion of jgit through refactoring. However, in only 10 (41%) of these the architecture was discussed.

As one can see from the table, when considering all reviews under study (with and without architectural awareness), most of the reviews identified as performing significant architectural changes caused a degradation in the systems' structural cohesion and coupling. This is arguably the moment which developers should be most aware of the architectural impact of their changes since poor architectural decisions might lead to bug proneness [3] and increased maintenance effort [5].

For feature-related reviews, changes that improve the architecture tend to discuss the structure of the system more often than reviews in which the architecture is degraded. In fact, the ratio of architecture discussion in feature-related reviews that improve the structure of the system is considerably higher than the overall discussion ratio for all reviews (see

TABLE 6: Number of reviews that either improved or degraded the systems' cohesion and coupling for different intents and the subset of reviews with evidence of developers' awareness.

System	Intent	Coupling						Cohesion					
		Improvement			Degradation			Improvement			Degradation		
		Total	Awareness		Total	Awareness		Total	Awareness		Total	Awareness	
egit	Feature	8	4	50%	78	10	12%	8	5	62%	32	4	12%
	Refactoring	7	5	71%	17	10	58%	8	3	37%	7	5	71%
	Bug Fixing	2	1	50%	10	3	30%	4	0	—	4	1	25%
jgit	Feature	16	5	31%	88	18	20%	11	5	45%	24	9	37%
	Refactoring	9	5	55%	16	13	81%	24	10	41%	13	11	84%
	Bug Fixing	1	0	—	1	0	—	1	0	—	3	1	33%
platform.ui	Feature	21	7	33%	35	2	5%	5	2	40%	10	3	30%
	Refactoring	21	6	28%	9	3	33%	12	4	33%	1	0	0%
	Bug Fixing	3	1	33%	24	1	4%	1	0	0%	8	0	0%
linuxtools	Feature	25	12	48%	46	24	52%	15	7	46%	46	16	34%
	Refactoring	16	8	50%	28	16	57%	17	9	52%	21	12	57%
	Bug Fixing	5	3	60%	2	0	—	4	3	75%	3	0	—
java-client	Feature	3	2	66%	16	8	50%	4	3	75%	14	4	28%
	Refactoring	1	1	100%	3	3	100%	2	2	100%	0	0	—
	Bug Fixing	0	0	—	1	1	100%	1	0	—	0	0	—
jvm-core	Feature	0	0	—	20	2	10%	0	0	—	9	2	22%
	Refactoring	2	1	50%	4	3	75%	1	1	100%	2	2	100%
	Bug Fixing	1	0	—	1	1	100%	0	0	—	0	0	—
spymemcached	Feature	0	0	—	14	1	7%	2	0	—	4	1	25%
	Refactoring	0	0	—	2	0	—	0	0	—	0	0	—
	Bug Fixing	0	0	—	1	0	—	1	0	—	0	0	—

RQ2). This indicates that architecture discussion during code review might lead towards code that improves the structure of the system even when developers are incorporating new features into the system.

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 52% and 57% 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 Section 6 we present a qualitative analysis that sheds light on these unexpected phenomena.

In order to assess the effect that architectural awareness has on the improvement and degradation of structural cohesion and coupling, we report in Figure 4 the distribution of cohesion and coupling for reviews we found evidence of architectural awareness and for reviews where we did not. Since the Couchbase systems have a small number of significant architectural changes, we include only the Eclipse systems in Figure 4. 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. For some of the box-plots, the number of observations is insufficient to perform a meaningful statistical test. For example, when considering the `jgit` system and the number of reviews that improved the single cohesion of the system, we only had 10 and 22 reviews with and without architectural awareness, respectively. Hence, we rely on a manual (and visual) investigation of the box-plots for this analysis.

Consider the box-plots that depict the distribution of cohesion and coupling for the reviews that improved either the system's overall cohesion and coupling or the cohesion and coupling of a single package. As one can see, the reviews in which the architecture was discussed presented larger improvements in structural cohesion and coupling. When looking at jgit in particular, reviews with evidence of architectural discussion presented considerably larger improvements to the coupling and cohesion of single packages in the system, as can be seen in boxplots (xiii) and (xiv), respectively.

When considering the reviews that degraded the system's cohesion and coupling, we found few cases in which the reviews with evidence of architectural discussion caused less degradation than reviews in which the architecture was not discussed. In (xii) for example, reviews with architectural discussion caused less degradation to the overall cohesion of jgit than their counterparts with no architectural discussion. However, this did not replicate to most of 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 an indication that architectural awareness has a positive effect on the cohesion and coupling of the systems we study for the reviews in which the structural architecture was improved. However, our data suggests that, when considering reviews that degrade the system's architecture, apart from specific cases, architectural awareness does not have a noticeable effect on 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 cohesion and coupling. By contrast, the architecture

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2912113, IEEE Transactions on Software Engineering

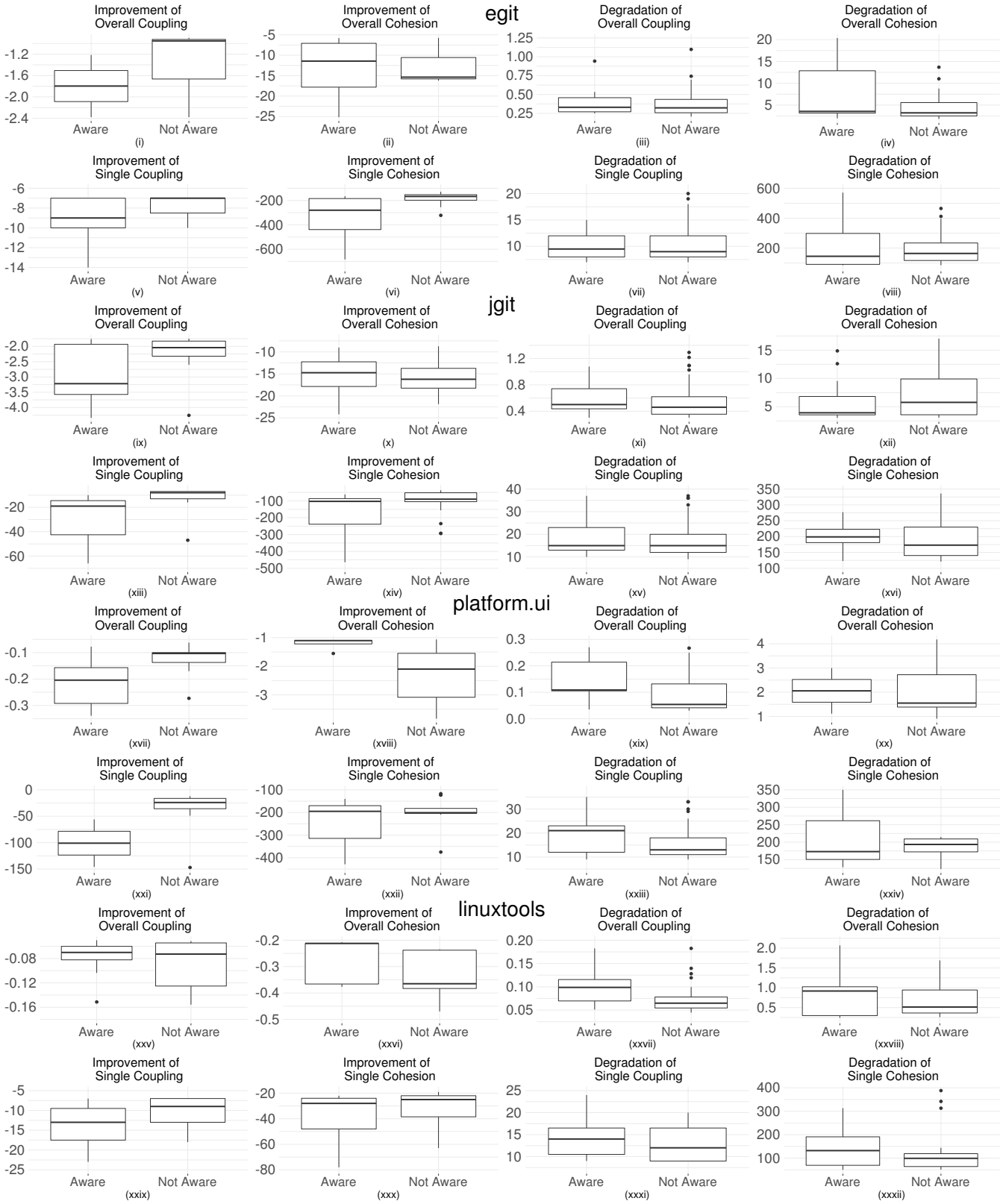


Fig. 4: Distribution of cohesion and coupling for reviews where we found evidence of architectural awareness and for reviews where we did not. We report box-plots for the reviews that improve and degrade the overall cohesion/coupling of the system and also for the reviews that improve and degrade the cohesion/coupling of a single package in the system.

is similarly discussed in reviews that perform a refactoring. 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.

5.4 RQ4: How do architectural changes evolve during code review?

In RQ1–3, we focused our analysis on the final merged revision of each code review as a representative of the architectural change being introduced to the system. However, as discussed in Section 3.1, the reviewing process is iterative, and a single code review goes through a series of revisions based on the reviewers' feedback before it is incorporated into the system. Thus, in this research question, we investigate how architectural changes evolve during the code review process.

To answer this question, we consider the architectural changes identified as outliers that have more than one revision, and compare the cohesion and coupling values between the last merged revision and all the previously submitted revisions. In this paper, we collect 8 different values of cohesion and coupling for each code review, where a review might be identified as an outlier for more than one of these 8 different metrics. As an example, a review may be identified as architecturally significant for considerably improving the coupling of a single package even though the impact on the overall coupling is small. For this particular review, we only compare the values for improving the coupling of a single package since this was the metric in which the review was identified as an outlier. This procedure avoids accounting for variation in metrics in which the reviews did not cause a significant impact on the structural architecture.

In the context of this paper, a code review may evolve in four different patterns, as detailed next.

Invariant: the cohesion and coupling values are the same for all revisions submitted during the code review. In this case, there was no cohesion and coupling evolution. All revisions submitted during the reviewing process presented the same improvement or degradation to the system's structure.

Positive: the last merged revision presents better cohesion and coupling values than all the previous revisions. Consider a code change where the first revision improved the cohesion and coupling of the system. In this case, a positive evolution indicates that the merged revision enhanced the improvement to the system's structure in comparison to the first revision. Differently, if the first revision presented a degradation to cohesion and coupling, a positive evolution indicates that the merged revision had a smaller degradation than the first revision. Although positive, the last revision still degraded the system's structure, albeit been 'less bad' than the first revision.

Negative: the last merged revision presents worse cohesion and coupling values than all the previous revisions. Consider a code change in which the first revision degraded the system's structure. In this case, a negative evolution indicates that the merged revision had a bigger degradation than the first revision. In other words, the architectural impact only got worse during the reviewing process. Differently, when the first revision presents an improvement to the system's structure, a negative evolution indicates that the impact of the merged revision in the system's structure was not as good as in the first revision submitted.

Mixed: there are revisions exhibiting both better and worse cohesion or coupling values than the last revision. In this particular case, for a code review in which the first revision improves the system's structure, we can observe subsequent revisions with both smaller and bigger improvements than the first one. Similarly, a code review that degrades the system's structure presents a mixed evolution when the subsequent revisions of a code review present both smaller and bigger degradations than the first revision.

Consider code review 83313 from egit, for example. This review was identified as an outlier due to its significant improvement in the cohesion of a single package in the system. The cohesion value itself has not changed during four revisions, which characterises this review as having an invariant evolution pattern.

By contrast, we now look at review 22194 from jgit, which has been identified as an outlier because of the significant degradation it caused to the system's overall cohesion. For this code review, the degradation caused by the last merged revision is smaller than the degradation caused by the first revision submitted for review. In fact, during its 8 revisions cycle, twice the developers changed the patch in a way that ameliorated the degradation in the system's overall cohesion. Thus, this review is considered to have had a positive evolution during code review.

Alternatively, we now consider code review 7633 from linuxtools. The improvement in the system's overall coupling caused by the last revision is smaller than the coupling improvement of its previous 3 revisions. In this case, the structural improvement that was finally merged into the system was not as good as it was on previous revisions of the code change. Hence, this review is identified as having a negative evolution.

Finally, we look at review 13688 from java-client, which caused a significant degradation to the coupling of a single package in the system. This review had a total of 13 revisions, and the coupling value of the final revision is, at the same time, better than the coupling of revision 1, and worse than the coupling of revision 6. In this scenario, this review is considered as having a mixed evolution.

Table 7 presents the evolution of architectural changes during code review according to the scenarios discussed above and the impact they caused to the system's structure. In addition, we group the reviews by different intents and different levels of architectural discussion. Reviews that exhibit an invariant evolution pattern are indicated by the columns *Same* in the table. Similarly, reviews in which we observe a positive, negative and mixed evolution are indicated by the columns *Pos*, *Neg* and *Mix*, respectively.

As one can see from the table, when considering feature-related reviews, the ratio of reviews in which the architectural impact remained the same during the reviewing process

TABLE 7: The evolution of architectural changes during code review. For the reviews that improve or degrade the system's architecture, we present the ratio of reviews in which the architectural impact remains the same. In addition, we provide the ratio of reviews in which the architectural impact only improved during the reviewing process (positive evolution) as well as the ratio of reviews in which the architectural impact only got worse during the reviewing process (negative evolution). Finally, we present the ratio of reviews that the final merged revision exhibit both better and worse architectural impact than previous revisions (mixed evolution). All the reviews are grouped by the developers' intent and level of architectural discussion during review.

Intent	Discussion	Coupling								Cohesion							
		Improvement				Degradation				Improvement				Degradation			
		Same	Pos	Neg	Mix	Same	Pos	Neg	Mix	Same	Pos	Neg	Mix	Same	Pos	Neg	Mix
Feature	**Never**	25%	35%	15%	23%	37%	13%	36%	12%	45%	40%	5%	10%	56%	11%	22%	9%
	Description	35%	23%	11%	29%	37%	11%	40%	11%	46%	30%	15%	7%	40%	3%	43%	13%
	Comments	13%	34%	26%	26%	14%	10%	41%	33%	15%	30%	46%	7%	15%	5%	50%	30%
	Overall	25%	30%	19%	23%	34%	12%	36%	15%	40%	30%	19%	9%	48%	9%	30%	12%
Refactoring	**Never**	68%	6%	20%	3%	56%	17%	21%	4%	69%	19%	11%	0%	81%	9%	9%	0%
	Description	60%	12%	20%	8%	52%	8%	28%	10%	70%	16%	8%	4%	37%	8%	37%	16%
	Comments	36%	18%	27%	18%	40%	5%	40%	15%	71%	14%	14%	0%	18%	0%	54%	27%
	Overall	61%	10%	21%	7%	52%	10%	26%	10%	70%	17%	9%	1%	46%	7%	34%	12%
Bug Fixing	**Never**	75%	25%	0%	0%	40%	16%	30%	13%	85%	14%	0%	0%	63%	18%	18%	0%
	Description	40%	20%	20%	20%	40%	0%	40%	20%	50%	0%	0%	50%	50%	0%	50%	0%
	Comments	0%	100%	0%	0%	0%	0%	100%	0%	100%	0%	0%	0%	—	—	—	—
	Overall	50%	30%	10%	10%	37%	13%	35%	13%	80%	10%	0%	10%	61%	15%	23%	0%

is often below 50%. In fact, for all feature-related reviews, cohesion and coupling values are the same in only 31% of the cases, which indicates that reviews that implement new features or enhance existing features tend to change during code review in a rate of 69%. On the other hand, for reviews where developers had the intent of refactoring or fixing a bug, the number of reviews in which the metrics of cohesion and coupling remained the same is considerably higher. For reviews in which developers refactored the system, the architectural impact remained the same in 55% of the cases, for example. In feature-related reviews, we observed that the code review process is often used to discuss the behaviour of the system for the new feature, which explains the higher amount of architectural variation during the reviewing process of these types of reviews. Differently, during our manual inspection, we noticed that reviews that refactor the system or fix a bug tend to be accepted as they are, without much feedback on how the revision can be improved.

In addition, one should note that reviews tend to present changes in their architectural impact when developers provide feedback regarding the system's architecture as comments during the reviewing process. For feature, refactoring and bug fixing reviews, the architectural impact of the latest revision was different than the initial revision in 85%, 59% and 75% of the cases, respectively.

We noticed that reviews that improve the system's structure tend to have a higher ratio of positive evolution when compared to reviews that degrade the structural architecture. Similarly, the ratio of reviews with a negative evolution is higher in reviews that degrade the architecture than in reviews that improve the architecture. For reviews that improve the system's cohesion and/or coupling, we observe a positive and negative evolution of 24% and 13%, respectively. In contrast, we observe a positive and negative ratio of 7% and 37% for reviews that degraded the system's cohesion and/or coupling.

These observations indicate that architectural changes tend to follow their initial trend, i.e., improvement or degradation, during the code review process. As an example, the architectural impact of reviews that degrade the system's structural architecture is only ameliorated (positive evolution) in 8% of the cases for feature-related reviews.

As previously mentioned, reviews in which the architecture is discussed in the comments tend to exhibit changes during the code review process. We expected that architectural feedback from other developers during code would lead to code reviews with a positive evolution, i.e., architectural changes that improve the system's structure would become even better and architectural changes that degrade the system's structure would be ameliorated through the feedback.

We observed the opposite though. We noticed that reviews in which we can observe architectural feedback most often result in worst values of cohesion and coupling, where the ratio of negative evolution for reviews with architectural discussion in the comments is 33%, compared to only 18% in its positive counterpart. This is a counterintuitive finding as we expected that architectural feedback during code review would lead to architectural improvements, i.e., architectural improvements would be enhanced and architectural degradation would be ameliorated.

Finally, the number of reviews with a mixed evolution is the smallest when compared to the reviews with positive and negative evolution. In total, only 11% of reviews exhibit a mixed evolution, while 15% and 25% of reviews present a positive and negative evolution, respectively.

As an answer to RQ4, we noticed that apart from feature-related reviews, the impact that architectural changes cause tend to remain the same during the code review process. Moreover, when the architectural impact does change, it tends to follow the trend of the initial revision, where degradations to the system's architecture tend to become worse as the review progresses and improvements tend to become better.

In addition, we noticed that architectural feedback during code review leads to adjustments in the patch. However, these adjustments tend to be negative, indicating that the

current architectural feedback provided by developers during code review is not assisting their peers in improving patches that cause significant architectural changes, i.e., improvements to the systems' structure are decreased instead of enhanced and degradations to the systems' structure are aggravated instead of ameliorated.

6 Qualitative Analysis of Refactorings that Degrade the Architecture

In RQ3 and RQ4 we observed architectural changes in which the developers had the intent of refactoring the system but the merged revision resulted in a degradation of the system's structural architecture. This is a counterintuitive finding as one expects that refactorings should have a positive effect on the system's structure. Hence, inspired by the recent study by Cedrim et al. [50], we define negative refactorings as reviews in which the developers have the intention of refactoring the system but the merged revision caused a worsening in the system's cohesion and/or coupling metrics. In order to shed light on this issue, we performed a qualitative analysis of all code reviews in which developers performed a negative refactoring.

For this analysis, we identified 81 code reviews that performed a negative refactoring. This accounts for 40% of the reviews in which developers performed a refactoring, and 11% of all significant architectural reviews. We qualitatively analysed these 81 code reviews, where we carefully read the reviews' description, comments and source code to better understand the details of the change.

In 31 (38%) of these reviews, the refactoring was performed as a side operation due to a bigger change. For all these cases, in order to implement a new feature or enhance an existing feature, developers extracted existing code to be reused by the new feature. In this scenario, since we cannot isolate the refactoring itself for source code analysis, we are unable to know for certain whether the refactoring was positive or negative. Consider review 724 from egit, for example. The developer describes the review as *"this change adds commit functionality to the staging view. The commit message part of the commit dialog was extracted to a reusable component and is now both used by commit dialog and staging view"*. As one can see, part of the existing code was extracted to a reusable component to enable the implementation of a new feature.

The overload of the "refactoring" term is also a common reason for negative refactorings to be observed. We noticed that in 16 (19%) reviews the developers claimed a refactoring was being made when the change actually consisted of a feature improvement. When looking at review 7801 from spymemcached, the developer describes the review as *"Refactored Operations to improve correctness of vbucket aware ops"*. In this case, the developer is clearly improving a functional property of the system but is using the term "refactoring" to describe it.

Among the 81 reviews that performed a negative refactoring, we identified 13 reviews (16%) where the developers attempted an improvement to the structural architecture but failed to achieve so. Review 9818 from linuxtools is described as *"Decouple the double click listener from the editor internals"*. In this review, the developer extracted part of the logic from the CEditor class into an internal package actions.hidden. However, classes from the package that CEditor belongs were now depending on a class from another package, which considerably degraded the coupling of the ui.ide.editors package. 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.

In 13 (16%) reviews that caused a degradation to the system's cohesion and/or coupling, we identified an attempt of improvement to the code base where the developer exhibits a semantical reasoning instead of a structural one. Consider review 970 from jgit, for example. In this review, the developer *"isolates all of the local file specific implementation code into a single package"*. By moving a large portion of the code base that was related to a particular feature to a separate package, there was a steep increase in the number of dependencies between packages, which significantly deteriorated the system's overall coupling. 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 [19], [20].

The remaining set of refactoring reviews that caused a degradation to the system's structure consists of isolated scenarios that are not related to the cases discussed above. The reasoning behind such reviews include, but are not limited to, removal of dead code, removal of code duplication and refactoring claims that were not actually implemented in the source code.

7 Discussion

In this section, we discuss the main contributions of this paper and reason about their implication in future software engineering research and practice.

7.1 Architectural Awareness Expectations and Observations

The first expectation one would have regarding architectural awareness is that developers would often discuss the system's architecture for the changes with most significant impact. However, the data we collected shows that developers do not discuss the system's structure in 69% of the cases. Moreover, only 15%, 9% and 5% of the reviews discuss the architecture in the discussion, comments and both, respectively. These observations indicate a large lack of architectural awareness during the changes that most impacted the system's structure, which goes against general expectations.

Next, we would expect that reviews in which the developers were aware of the system's structure would exhibit better architectural changes. When considering the reviews that caused an improvement to the system's structure, we noticed considerably larger improvements for the reviews in which developers discussed the architecture in either the description or comments. Hence, the observations drew from the paper support the expectations that architecture discussion and awareness during code review leads to better architectural changes.

Finally, we expected that the code review process would lead to improvements in architectural changes. That is, for a change that initially degrades the system's architecture, we would expect that architectural discussion would lead to a merged revision that ameliorates the degradation presented in the first submitted revision. Similarly, for revisions that initially improve the system's structure, we expected that architectural discussion would enhance this structural improvement. However, our data suggests the opposite, where 33% of the reviews with architectural discussion in the comments exhibited a negative evolution. This indicates that more often than not, architectural feedback during code review caused both decreases to architectural improvements instead of enhancements and aggravations to architectural degradations instead of amelioration.

7.2 Tool Support for Architectural Changes During Code Review

In the course of this empirical study, we observed that the implementation of a new feature and/or the enhancement of an existing feature account for 54% of the changes that cause a significant impact on the system's structure, followed by refactoring (26%) and bug fixing (10%), respectively.

Moreover, for the code reviews we investigated, we noticed that the architecture is discussed in 31% of the cases. In addition, when considering feature-related reviews, developers discuss the system's structure in only 26% of the cases. Hence, developers are least discussing the system's architecture during the changes that most often affect it.

The lack of architectural discussion and the amount of code that is usually introduced in feature-related reviews add up to make these changes the most likely to introduce problems to the architecture of the system and the code base as a whole, such as architectural debt and code smells. This indicates that we should design approaches and build tools that assist developers not only when they refactor the system, but mostly when they are working on features.

By measuring and comparing cohesion and coupling quality metrics for before and after versions of the code base in reviews that performed significant architectural changes, we were able to assess the impact of architectural discussion in the quality of the architectural change being made. For reviews that improve the cohesion and/or coupling of the system, we observed that reviews in which the architecture is discussed tend to exhibit considerably bigger improvements in the system's structure when compared to reviews that do not discuss the architecture.

We consider this as evidence that architectural awareness and discussion during code review leads to better architectural changes. This points out to the need of tool support for architectural changes during code review, where developers would automatically be made aware of the architectural impact of their changes, possibly fostering discussion and leading to changes with bigger improvements and less degradation to the system's structural architecture.

Code review is an iterative process, in which a certain code change undertake a series of revisions until the final version of the change is merged into the system. By studying the evolution of architectural changes during code review, we noticed that reviews in which developers perform a refactoring or bug fix tend to largely remain the same during the reviewing process, i.e., the values of cohesion and coupling are not altered during all revisions.

Differently, for feature-related reviews, the architectural impact changed more often than not during the reviewing process. In addition, when developers gave architectural feedback in the form of comments during code review, we observed a 73% ratio of architectural change throughout the reviewing process. This indicates that developers are willing to consider architectural feedback during code review and adjust their changes accordingly.

However, we observed that most of the architectural changes had a negative evolution during code review, i.e., the values of cohesion and coupling of the last merged revision are worse than the first revision submitted for review. This illustrates that the current architectural feedback being provided by developers is not assisting their peers in improving architectural changes that undergo code review.

The results from RQ1–4 strongly indicate that developers need tool support for architectural changes during code review, in special for when they are introducing new features or enhancing existing features in the system. In the course of this study, we found feature-related changes where the developers were aware of the architectural impact of their changes, yet the changes were merged into the system regardless of their architectural impact (even when highly negative).

This indicates that there do exist circumstances where some tasks, e.g., feature implementation and bug fixing, can take priority over structural quality. Thus, having a tool integrated into code review that would make developers aware of the architectural impact of their changes can be beneficial in assisting developers to deal with architectural debt and code smells. Moreover, the tool could suggest modifications to the patch that would enhance architectural improvements and ameliorate architectural degradations. Hence, providing a feedback that is focused on the constant improvement of architectural quality and prevention of architectural problems.

7.3 Leveraging Code Review Data for Empirical Studies

During our empirical study, we observed a non-negligible number of reviews in which the developers performed a refactoring that degraded the structural architecture of the system. In hindsight, one could conclude that these were all cases in which developers have attempted to improve the system but failed.

However, after a careful analysis of the code review data for each of these changes, we noticed that in 38% of the cases, the refactoring was mixed with feature-related changes, which caused the review to have a negative effect in the system's structure.

Moreover, we observed that 19% of the negative refactorings were due to developers overloading the phrase 'refactoring' by performing feature-related changes instead. In parallel, out of the 81 refactorings that degraded the system's structure, 13 were improvements to the code base in which the developers were trying to improve semantical aspects of the code rather than structural. Finally, in only 16% of the reviews we could identify a failed attempt at pure structural improvement.

The empirical study performed in this paper, and the above qualitative analysis in specific, could only be achieved by leveraging code review data. During code review, developers provide reasoning and rationale for the changes they make in the system, both when they submit and review code from their peers. Thus, code review data is a valuable source of knowledge regarding motivation for and explanation of software changes, from which properties such as intent and awareness can be inferred. In this context, code review datasets, such as CROP [33], provide data that can be leveraged by empirical studies in software engineering to answer questions that previously required interactions with developers, such as interviews and surveys.

8 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 sensitivity analysis of the parameters thresholds involved in the identification of significant reviews. By inspecting all reviews of the Couchbase system that exhibited any change in structural cohesion and/or coupling, we showed that the ratio of reviews that exhibit architectural discussion is statistically the same at the 0.01 confidence level. This indicates that the results of our research questions are not likely to be affected by the threshold choice we employed.

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. Thus, 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 [47]. The low and medium correlation indicates that the cohesion and coupling metrics we employed are indeed capturing changes in the structural architecture of the system and not only size fluctuations.

Manual classifications are naturally subjected 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 seven Java projects that were selected from a recently published open dataset of code review data. Even though the metrics we use and the analysis framework we employ are language agnostic, the results may not be generalisable to software systems written 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 [20], [24], [30].

Construct validity: We focus our analysis on the structural view of the software architecture, which is a low-level representation of the systems' architecture. Hence, our analysis and observations do not generalise to all the views and perspectives [1], [29] in the systems' architecture, especially the more abstract ones. Nevertheless, our empirical study focuses on architectural changes as performed by developers in the context of code review. Thus, it is fitting that we chose to study the structural view of the software architecture, as this is the one developers interact the most. Moreover, the structural architecture is the one practitioners commonly use as the groundwork for the design of the other architectural views [22].

9 Related Work

Tufano et al. [46] 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 code review also includes feedback provided by other developers and often links to both tickets in the issue tracking system and 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.

In a more recent work, Palomba et al. [51] extended the investigation on the lifecycle of code smells by considering code smells co-occurrences. By analysing open source systems, they identified the most common pairs of smells that appear together in a code entity as well as the patterns in which the co-occurrences are commonly introduced and removed from the system. Different from their previous work, they have not manually inspected the changes that introduce and remove code smells, and mostly relied on source code analysis to investigate the lifecycle of code smells co-occurrences.

In a similar work, Cedrim et al. [50] performed an empirical study to investigate how effective refactoring operations are as a way of removing code smells from a software system. They found that refactoring operations rarely remove code smells from the system, where they even observed cases in which refactorings create new code smells in the code base. This finding is similar to our observation of reviews in which developers performed a refactoring by negatively impacted the system's structural architecture. However, their study only considered the source code and commit message to identify the presence of refactoring operations, which might include a bias for when refactoring operations are used as a complement to a larger change, as we have observed in this study.

Several studies have been performed to qualitatively evaluate the developer's perception of cohesion and coupling metrics. Simons et al. [31] prepared a set of toy examples and surveyed developers to assess whether metrics represent the developer's perception of quality. Bavota et al. [19]

and Candela et al. [20] 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.

The measurement of architectural difference between two versions of the same system has been extensively discussed in the literature [52]–[54]. However, the early metrics were mostly focused on measuring the distance between two different modularisations of the same system, lacking the capacity to consider the addition and/or removal of components between two versions.

Hence, Le et al. [26] proposed *a2a*, a new metric for architectural change that is inspired by the original suite of *MoJo* metrics but now addresses the issue of added and removed components between versions. After the publication of our previous paper, Shahbazian et al. [55] extended our analysis framework to use *a2a* as a metric to infer the architectural impact of changes. After identifying the architecturally significant changes, the authors propose a machine learning method to predict the impact of architectural changes based on textual information extracted from the change request in the issue tracking system. *a2a* was originally proposed to evaluate high-level architectural views automatically recovered from source code through the usage of an architectural recovery technique such as ACDC [56] and ARC [57]. Since our study focuses on the low-level structural view of the software systems' architecture, we did not consider a2a in our study.

Recent studies have evaluated different metrics of structural cohesion and coupling as suitable measurements for architectural quality. In the context of search based software modularisation, Paixao et al. [24] compared the modularisation developers implemented in their systems against baselines generated by different search procedures. The solutions implemented by developers outperformed most of the solutions generated by naive search procedures, indicating that structural measurements of cohesion and coupling are properties of interest for developers.

In a similar setting, Ó Cinnéide et al. [30] evaluated a set of structural cohesion metrics for automated refactoring. In this case, different cohesion metrics led to different refactorings, which indicates these metrics do not capture the same property, even though they have been all suggested as structural cohesion measurements. 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.

Other empirical studies have been performed to study the effectiveness of code review in other aspects of software quality. McIntosh et al. [58] investigated the relationship between software defects to code coverage and participation during code review. In a similar study, Morales et al. [59] extends the investigation of code coverage and participation during code review, but now with a focus on the design patterns and anti-patterns.

10 Conclusion and Future Work

Architectural decisions have implications on the development and evolution of software systems. In spite of the large body of research dedicated to aid developers in such decisions, architectural erosion remains a problem faced by software developers. In this context, a better understanding of how developers perform architectural changes on a day-to-day basis is required.

Thus, we performed an empirical study that involved the inspection and classification of 731 architectural changes mined from 7 software systems. We focused our investigation on changes that had undergone a process of code review, and we assessed the common intents behind these architectural changes. Moreover, we investigated whether developers were aware of the architectural impact of their changes when performing and/or reviewing such changes. In addition, we evaluated the effect that intent and awareness have on the structural "quality" of the software architecture as measured by structural metrics of cohesion and coupling. Finally, we looked at how changes with significant architectural impact evolve during the code review process.

After analysing 731 reviews that performed significant changes to the system's structural architecture, we noticed that the intent behind 54% of the architectural changes is to either introduce a new feature or to enhance an existing feature. In addition, we found that refactorings and bug fixing accounted for 26% and 10% of the reviews with significant architectural changes, respectively.

Surprisingly, we found that the system's architecture is discussed only in 31% of the reviews we studied, which indicates a lack of architectural awareness when performing significant architectural changes. Moreover, developers tend to be more often aware of the architecture when the change is improving the system in terms of cohesion and coupling. 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.

In regards to the evolution of architectural changes during code review, we observed that reviews in which developers performed a refactoring or bug fix tended to remain the same during the reviewing process. However, feature-related changes tended to undergo adjustments during code review, especially when fellow developers provide feedback regarding the architecture as comments during code review.

Such observations indicate the need for tools and approaches that aim to assist developers when performing architectural changes, especially when they introduce a new feature or enhance an existing one. Since developers tend to take architectural feedback into account during code review, we set out, as future work, the development of an approach that will automatically identify architecturally significant reviews and make the developers aware of the impact of their change. In addition, the approach should provide feedback and suggestions to developers and reviewers, so that the architectural change can be guided in a direction that will ameliorate architectural degradation and enhance architectural improvement.

Moreover, in future work, we plan to incorporate other metrics of architectural change into our analysis framework. Metrics such as Decoupling Level [25], semantic coupling [19] and co-change dependencies [18] are likely to be good candidates to enhance our knowledge about developers' behaviour when performing architectural changes during code review.

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] E. Kouroshfar, M. Mirakhorli, H. Bagheri, L. Xiao, S. Malek, and Y. Cai, "A Study on the Role of Software Architecture in the Evolution and Quality of Software," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, vol. 2015-Augus. IEEE, may 2015, pp. 246–257.

[5] 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.

[6] 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.

[7] Y. Zhou and H. Leung, "Predicting object-oriented software maintainability using multivariate adaptive regression splines," *Journal of Systems and Software*, vol. 80, pp. 1349–1361, Aug 2007.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] N. Ajienka, A. Capiluppi, and S. Counsell, "An empirical study on the interplay between semantic coupling and co-change of software classes," *Empirical Software Engineering*, pp. 1–35, nov 2017.

[19] 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.

[20] 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, pp. 1–28, 2016.

[21] P. B. Kruchten, "The 4+1 view model of architecture," *IEEE software*, vol. 12, no. 6, pp. 42–50, 1995.

[22] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Measuring the Impact of Code Dependencies on Software Architecture Recovery Techniques," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 159–181, feb 2018.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] M. Paixao, J. Krinke, D. Han, and M. Harman, "Crop: Linking code reviews to source code changes," in *International Conference on Mining Software Repositories*, ser. MSR, 2018.

[34] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman, "Are developers aware of the architectural impact of their changes?" in *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[35] S. Pearce, "Gerrit code review for git," https://www.gerritcodereview.com, 2006, accessed in: May 2018.

[36] "Eclipse projects," https://eclipse.org/projects, 2018, accessed in: May 2018.

[37] "Couchbase Projects," https://developer.couchbase.com/open-source-projects, 2018, accessed in: May 2018.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

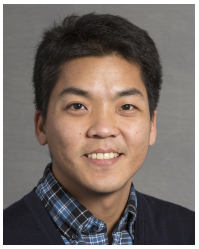
[42] M. d. O. Barros, F. d. A. Farzat, and G. H. Travassos, "Learning from optimization: A case study with apache ant," *Information and Software Technology*, vol. 57, no. 1, pp. 684–704, Jan 2015.

[43] Scitools, https://scitools.com/features, 2018, accessed in: May 2018.
[44] M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, and M. Harman. We will create a results section at crop's webpage upon publication because we do not wish to make the results publicly available for a paper under review. however, we are happy to provide data confidentially to reviewers by request if needed. [Online]. Available: https://to.be.disclosed
[45] J. W. Tukey, *Exploratory data analysis*. Addison-Wesely, 1977.
[46] 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.
[47] 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.
[48] M. G. Kendall, *Rank correlation methods*. Griffin, 1948.
[49] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Routledge, 1988.
[50] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez, "Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. New York, New York, USA: ACM Press, 2017, pp. 465–475.
[51] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "A large-scale empirical study on the lifecycle of code smell co-occurrences," *Information and Software Technology*, vol. 99, no. September 2017, pp. 1–10, jul 2018.
[52] V. Tzerpos and R. Holt, "MoJo: a distance metric for software clusterings," in *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*. IEEE Comput. Soc, 1999, pp. 187–193.
[53] B. Mitchell and S. Mancoridis, "Comparing the decompositions produced by software clustering algorithms using similarity measurements," in *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*. IEEE Comput. Soc, 2001, pp. 744–753.
[54] Zhihua Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 2004, pp. 194–203.
[55] A. Shahbazian, D. Nam, and N. Medvidovic, "Toward predicting architectural significance of implementation issues," in *International Conference on Mining Software Repositories*, ser. MSR, 2018.
[56] V. Tzerpos and R. Holt, "ACCD: an algorithm for comprehension-driven clustering," in *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE Comput. Soc, 2000, pp. 258–267.
[57] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Yuanfang Cai, "Enhancing architectural recovery using concerns," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, nov 2011, pp. 552–555.
[58] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. New York, New York, USA: ACM Press, 2014, pp. 192–201.
[59] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, mar 2015, pp. 171–180.

Matheus Paixao is currently a Research Assistant in the Computer Science Department at the State University of Ceara. He recently received the PhD degree in Computer Science at University College London, where he was part of the Centre for Research on Evolution, Search, and Testing (CREST) and Software Systems Engineering (SSE) Group. His research interests include software architecture, search-based software engineering, mining software repositories, modern code review and empirical software engineering.

Jens Krinke is Associate Professor in the Software Systems Engineering Group at the University College London, where he is Director of CREST, the Centre for Research on Evolution, Search, and Testing. His main focus is software analysis for software engineering purposes. His current research interests include software similarity, modern code review, program analysis, and software testing. He is well known for his work on program slicing and clone detection.

DongGyun Han is currently working toward the PhD degree in the Department of Computer Science, University College London, London, United Kingdom, and a member of the CREST centre. He received his MPhil degree in the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology. Before, he was a researcher at the KAIST Institute for IT Convergence. His research interests include modern code review, mining software repositories, and empirical study.



Chaiyong Ragkhitwetsagul is a lecturer at the Faculty of Information and Communication Technology, Mahidol University, Thailand. He received the PhD degree in Computer Science at University College London, where he was part of the Centre for Research on Evolution, Search, and Testing (CREST). His research interests include code search, code clone detection, software plagiarism, modern code review, and mining software repositories.

Mark Harman is an engineering manager at Facebook London, where he manages the Sapienz team. Sapienz has been deployed to continuously test Facebook's Android and iOS apps, leading to thousands of bugs being automatically found and in multimillion line communications and social media apps used by over a billion people worldwide every day. Mark is also a full professor at University College London. He is known for his scientific work on Search Based Software Engineering (SBSE), source code analysis, software testing, app store analysis and empirical software engineering. He co-founded the field SBSE, an active and impactful research area with authors spread over more than 40 countries. Automated Software Engineering research and practice is now the primary focus of his current work in both the industrial and scientific communities. In addition to Facebook itself, Mark's scientific work is also supported by an ERC advanced fellowship grant and by the UK EPSRC funding council.