UCL DEPARTMENT OF
COMPUTER SCIENCE

Ph.D Thesis

Software Restructuring: Understanding Longitudinal Architectural Changes and Refactoring

July 2018

Matheus Paixao

A report submitted for the fulfillment of the requirements
for the degree of Doctor of Philosophy of
University College London

Computer Science Department
University College London

Declaration

I, Matheus Paixao, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis. The work presented in this thesis is original work undertaken between September 2014 and July 2018 at University College London.

Tables 1 and 2 below present work that has been published and/or submitted between September 2014 and July 2018. First, I list the work that compose this thesis (No. 1–4), each of which is discussed in detail in this thesis (Chapters 3–5). Next, I include my co-authored publications that, although relevant to my PhD, are not discussed in this thesis.

Table 1: List of publications that compose this thesis

No.	Publication	My contributions to the work
1	M. Paixao, M. Harman, Y. Zhang, Y. Yu **An Empirical Study of Cohesion and Coupling: Balancing Optimisation and Disruption** *IEEE Transactions on Evolutionary Computation, 2017*	I proposed the research questions based on the literature review and the experience obtained in a previous publication (No. 5). I designed the experiment with suggestions from the other authors. I manually downloaded the 233 releases of the 10 systems we studied. I adapted the static analysis tool that extracted the structural modularisation of all releases. I implemented all search algorithms and performed all results analyses. The paper has been written by all authors.
2	M. Paixao, J. Krinke, D. Han, M. Harman **CROP: Linking Code Reviews To Source Code Changes** *Proceedings of the International Working Conference on Mining Software Repositories (MSR'18)*	I proposed the creation of a new code review dataset that would provide code review data linked to complete copies of the code base for open source software systems. I designed the mining framework. I adapted D. Han's original scripts to download the code review data and the snapshots of each system. I designed and implemented the heuristics that linked the code review data with the snapshots. I implemented the scripts that processed the raw data, extracted the relevant information, and built the dataset to be easily accessed. The paper has been written by all authors.
3	M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, M. Harman **Are Developers Aware of the Architectural Impact of Their Changes?** *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*	I proposed this empirical study to investigate how developers perform architectural changes on their daily basis. I designed the experiments in cooperation with Dr. Krinke and Prof. Harman. D. Han mined the code review data used in the work. I wrote the scripts to link code reviews to git commits in all systems. I adapted a static analysis tool to extract the structural modularisations of all systems from source code. I implemented the scripts to compute the structural metrics of cohesion and coupling for all reviews. I manually inspected all reviews with significant impact on the structural architecture. D. Han and C. Ragkhitwetsagul each inspected half of the reviews to complement my manual analysis. The paper has been written by all authors.
4	M. Paixao, J. Krinke, D. Han, C. Ragkhitwetsagul, M. Harman **The Impact of Code Review on Architectural Changes** *Work in progress submission to the IEEE Transactions on Software Engineering, 2018*	This paper is a journal extension of our previous conference paper (No. 3). I proposed the extension of the dataset used in the previous paper, which generated the CROP dataset (paper No. 2). I proposed the new research question that investigates the evolution of architectural changes during code review. I designed and executed the new qualitative investigation of the refactorings that degrade the system's structure. The paper has been written by all authors.

Table 2: List of publications not discussed in this thesis

No.	Publication	My contributions to the work
5	M. Paixao, M. Harman, Y. Zhang **Multi-objective Module Clustering for Kate** *Proceedings of the Symposium on Search Based Software Engineering (SSBSE'15)*	I designed the experiment alongside Prof. Harman and Dr. Zhang. I manually extracted Kate's structural modularisation and implemented the Two-Archive search algorithm that was later used in the evaluation. The paper has been written by all authors.
6	C. Ragkhitwetsagul, M. Paixao, M. Adham, S. Busari, J. Krinke, J. H. Drake **Searching for Configurations in Clone Evaluation – A Replication Study** *Proceedings of the Symposium on Search Based Software Engineering (SSBSE'16)*	I assisted C. Ragkhitwetsagul in designing the experimental framework, running the experiments and collecting the results. I assisted with the writing of the paper.
7	G. Guizzo, M. Bazargani, M. Paixao, J. H. Drake **A Hyper-heuristic for Multi-Objective Integration and Test Ordering in Google Guava** *Accepted for publication in the Proceedings of the Symposium on Search Based Software Engineering (SSBSE'17)*	I wrote the scripts to extract the structural modularisation of Google Guava from its source code.
8	J. Wilkie, Z. Halabi, A. Karaoglu, J. Liao, G. Ndungu, C. Ragkhitwetsagul, M. Paixao, J. Krinke **Who's this? Developer identification using IDE event data** *Proceedings of the International Working Conference on Mining Software Repositories (MSR'18)*	I assisted the main authors in designing the research questions. I assisted with the writing of the paper.
9	D. Han, J. Krinke, M. Paixao, G. Rosa, C. Ragkhitwetsagul **Bad Style Can Get Your Code Change Rejected** Submitted to the *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*	I manually inspected code style violations during code review to complement the full manual inspection performed in the paper. I assisted with the writing of the paper.
10	C. Ragkhitwetsagul, M. Paixao, J. Krinke, G. Bianco, R. Oliveto **Toxic Code Snippets on Stack Overflow** Submitted to the *IEEE Transactions on Software Engineering, 2018*	I manually inspected pairs of code clones to complement the full manual inspection performed by C. Ragkhitwetsagul. I assisted with the writing of the paper.

Abstract

The complexity of software systems increases as the systems evolve. As the degradation of the system's structure accumulates, maintenance effort and defect-proneness tend to increase. In addition, developers often opt to employ sub-optimal solutions in order to achieve short-time goals, in a phenomenon that has been recently called technical debt. In this context, software restructuring serves as a way to alleviate and/or prevent structural degradation.

Restructuring of software is usually performed in either higher or lower levels of granularity, where the first indicates broader changes in the system's structural architecture and the latter indicates refactorings performed to fewer and localised code elements. Although tools to assist architectural changes and refactoring are available, there is still no evidence these approaches are widely adopted by practitioners. Hence, an understanding of how developers perform architectural changes and refactoring in their daily basis and in the context of the software development processes they adopt is necessary.

Current software development is iterative and incremental with short cycles of development and release. Thus, tools and processes that enable this development model, such as continuous integration and code review, are widespread among software engineering practitioners. Hence, this thesis investigates how developers perform longitudinal and incremental architectural changes and refactoring during code review through a wide range of empirical studies that consider different moments of the development lifecycle, different approaches, different automated tools and different analysis mechanisms. Finally, the observations and conclusions drawn from these empirical investigations extend the existing knowledge on how developers restructure software systems, in a way that future studies can leverage this knowledge to propose new tools and approaches that better fit developers' working routines and development processes.

Impact Statement

This thesis investigates how developers perform software restructuring at both high and low levels of granularity in real-world software systems. The studies we describe aim at evaluating commonly believed and accepted assumptions researchers have in regard to how developers restructure software systems, where our results confirm some of these assumptions while providing new insights for others. Our observations expand the current empirical knowledge in software restructuring, which will influence future empirical studies in this area. Moreover, our conclusions will serve as the groundwork for the research and development of new automated approaches to assist software developers when performing software restructuring.

The research presented in this thesis has been published in 3 academic publications at high-level software engineering and computer science venues. In total, the work performed during this PhD has span out 10 research papers. Moreover, 2 papers composed by the later studies reported in this thesis are currently being prepared for submission.

Other researchers and the academic community will benefit not only from the results and observations from the research presented in this thesis but also from the methodology and datasets produced. The framework we designed to identify significant architectural changes (see Chapter 5), for example, has already been adopted by other researchers in related empirical studies on software architecture.

In addition, the code review dataset we collected for the fulfillment of the empirical studies we report in this thesis has been separately published and made available for the research community. At the time of writing, 2 internal research projects within the UCL's computer science department are making use of this dataset. Moreover, I have been personally contacted by researchers from other universities and research institutes from the Netherlands, Japan, Brazil and Scotland to make me aware they are using our dataset for their own research projects.

Finally, all the results, datasets, manual classifications and source code generated by the research reported in this thesis has been made available for the research community. This allows for complete replication and extension of all the research we carried on during this PhD.

Acknowledgements

A PhD degree is not easy. Through every single part of the journey I had the help, assistance and support of many people. I will try to express my gratitude in the next few paragraphs.

First and foremost, I would like to thank my supervisory team. I thank Prof. Mark Harman for being my main supervisor for the first 2 years of my PhD, which helped me understand what it meant to do a PhD in a such a different university, country and culture. I will hold dearly on my memory all the moments we spent together, which included not only academic but also personal and professional guidance and counselling. In particular, I will most certainly remember the week we spent in Brazil, in which I can guarantee solidified and shaped my visions of what means to be a sucessful researcher.

I would like to equally thank Dr. Jens Krinke, who became my main supervisor on the second half of my PhD and drove me to the finish line in amazing fashion and support. I appreciate the belief you had in accepting a student halfway through the journey, in which I could see the same support and dedication as if I have been your student since the beginning. I thank you for the slight change in topic, which made fall in love with parts of software engineering research I would most certainly not be exposed to.

Last but not least, I thank Dr. Yuanyuan Zhang who stayed alongside the journey from beginning to end to provide not only academic support but mostly tips on how to handle my supervisors, smart advice on my smaller tasks and general guidance through the journey. Be certain your help was imperative to my success, and I fully believe and support in your new adventure in life.

During my entire PhD I've always felt grateful to be part of CREST, mostly due to the amazing people that make it exist. In particular, my PhD colleagues who made the whole thing a bit less stressful and always funnier and more enjoyable. Chaiyong Ragkhitwetsagul, Bobby Bruce, Carlos Gavidia and DongGyun Han, thank you for every moment we spent. To CREST's other members I thank for the insightful discussions, moral support and amazing work which I became familiar with.

Living abroad is certainly not easy. Hence I thank Rafael Carmo for first making all the arrangements for my moving to London and second for the three years we shared in the city. Lauren O'Neill and Brynne Inglish easily became my favorite americans. Thank you for the two years of company, conversations, cultural exchange, and of course, beer. Thank you Juliana Ramos for this last year of support. To all four of you guys, it was a blast.

To all my 'outside UCL friends', thank you for the company, parties, silly moments and sometimes insightful life advice and experiences. I would add the name of the group here if this was not a formal document, but you know who you are.

I thank the Brazilian National Council for Scientific and Technological Development (CNPq) which funded my PhD and my 4 years of living in London.

I could not forget to thank my parents who first provided me with such excellent education and later supported me in all my academic decisions.

Finally, I thank Mariana Maia for all the calls, conversations and advice during this journey. Thank you for all the times you flew more than 3,000km to spend a couple (sometimes more) of days with me. We really made it happen, to our own surprise. Your coffee unfortunately ceased to be the best, but in exchange I found much more to thank you for this time.

Contents

List of Figures

List of Tables

1 Introduction

The development of software systems requires constant change and evolution due to its naturally iterative and incremental nature (Lehman, 1979; Lehman et al., 1997). The system's architecture establishes a foundation for its development and evolution, as it guides developers' activities such as the introduction of new features, the enhancement of existing features and refactoring. However, a software system tends to grow in complexity as it evolves, which usually leads to architectural deterioration and erosion (de Silva and Balasubramaniam, 2012). In this thesis, we consider the software architecture to be the structural organisation of source code elements, i.e., the high-level decomposition of source code into submodules and/or subcomponents and the dependencies between these elements.

Technical debt is a concept first coined by Cunningham (1993) to describe 'not quite right' decisions during software development. Technical debt commonly refers to sub-optimal engineering decisions made to achieve a short-time goal, such as a faster delivery time. As technical debt can be manifested during the complete lifetime of a software project, researchers tend to report studies that focus on a particular type of technical debt, which include, but is not limited to, requirements debt, design debt, code debt and test debt (Li et al., 2015).

In the context of this thesis, we deal with architectural debt (Kruchten et al., 2012), which can be understood as architectural decisions and modifications that deteriorate the structural architecture of a software system. Such deterioration affects important architectural properties such as maintainability and comprehensibility which, in some cases, might lead to a complete re-development of software systems (van Gurp and Bosch, 2002; van Gurp et al., 2005). In recent studies, not only architectural debt has been listed as the most dangerous type of technical debt (Ernst et al., 2015), but also the accumulation of architectural debt has been observed to cause increased maintenance effort (Xiao et al., 2016) and bug-proneness (Mo et al., 2015) of files involved in such architectural decisions.

Software restructuring can be employed as a method for the reverting of structural degradation and the prevention of architectural debt. Developers restructure software systems at different levels of granularity, from the changes and improvements of high-level modules and subsystems to localised refactorings of code elements and functions. Researchers have proposed a number of approaches to automatically assist developers when performing software restructuring. For example, search-based software modularisation (Mancoridis et al., 1998, 1999; Mitchell and Mancoridis, 2006; Praditwong et al., 2011) has been evaluated as an automated technique to improve the high-level structure of software systems. In addition, tools for automated recommendation and application of refactorings (Tsantalis et al., 2008; Sales et al., 2013; Silva and Valente, 2017; Tsantalis et al., 2018) have been developed to assist developers in low-level restructuring operations.

However, to this date, there is no evidence to suggest that such automated approaches have been widely adopted by software engineering practitioners. On the contrary, recent empirical studies have revealed that search-based modularisation approaches can be highly disruptive (Hall et al., 2012), and that refactoring operations might actually lead to structural problems instead of improvements (Cedrim et al., 2017; Tufano et al., 2017). Thus, more research is necessary to better understand how software developers perceive and perform software restructuring. Moreover, these studies need to take into account the current models and practices of software development, so that the conclusions drawn from these studies can lead to actionable findings.

The process of code review is widespread in the software development industry, with evidence of adoption by companies like Microsoft (Bacchelli and Bird, 2013), Google (Kennedy, 2006) and Facebook (Tsotsis, 2011). During code review, software changes undergo a process of peer review in which developers communicate and exchange feedback regarding the change until the code is merged into the system. Empirical evidence has been reported that code review enables the detection of bugs related to maintainability that would otherwise be merged to the system's code base without code review (Mantyla and Lassenius, 2009; Beller et al., 2014). Moreover, code review has been shown to substantially improve code quality, knowledge transfer and social communication (Bacchelli and Bird, 2013; Rigby and Bird, 2013).

1.1 Problem Statements

This thesis covers several interconnected topics surrounding software restructuring, ranging from high-level architectural evolution to low-level refactorings. The research problems tackled in this thesis are described next.

As previously mentioned, the structural architecture of a system tends to become more complex as the system evolves (Lehman, 1979; Lehman et al., 1997; de Silva and Balasubramaniam, 2012). In this context, tools and approaches that employ search-based software modularisation have been proposed as a way of optimising the existing structure of a software system by considering architectural quality metrics. However, very few of these studies have actually validated the quality metrics they use to optimise the systems, and even fewer studies evaluated the disruption caused by such approaches. Hence, in the context of the architectural evolution of software systems, we answer the following questions:

To what extent do developers respect structural metrics of software modularisation?

What is the disruption caused by state of the art approaches for search-based software modularisation?

How search-based modularisation can be used to find trade-offs between architectural improvement and structural disruption?

Any sort of approach for architectural debt amelioration and/or prevention, including search-based software modularisation, as discussed above, suggests a set of architectural modifications to be performed by the system's developers. However, to the best of our knowledge, there are no empirical studies that investigate how developers perform architectural changes on their daily basis. Without knowing the developers' workflow, perceptions and directions when performing architectural modifications, approaches for the prevention and amelioration of architectural debt will likely not be uptaken by practitioners. In this context, the code review process has recently emerged as a major aspect of software development and quality control, being adopted by both industrial and open source software development. Thus, by analysing the data generated during the code review process, we answer the following questions:

How do developers perform architectural changes during code review?

Does the code review process assist developers in performing better architectural changes?

Technical and architectural debt is tackled not only by high-level restructuring, such as the architectural changes mentioned above but also through low-level restructuring at code and class level, in a practice that became known as software refactoring. When first proposed, refactoring was seen as code transformations with the sole purpose of improving the system's structure (Fowler et al., 1999). However, recent studies have shown that refactoring is sometimes used with other goals and at different times during the software development lifecycle (Murphy-Hill et al., 2012). Thus, this thesis makes again use of the data generated during the code review process to investigate the circumstances and context in which developers employ refactoring operations on their daily basis. We answer the following questions:

What are common intents when developers employ refactoring operations?

How do refactoring operations evolve during code review?

To answer all the questions presented above, this thesis presents a series of empirical studies surrounding the practice of software restructuring. We investigate current practices of restructuring at different granularity levels by analysing data from real-world software systems, which generates actionable findings for both industrial and academic software engineering practitioners, as discussed as follows.

1.2 Goal and objectives

The goal of this thesis is to *study software restructuring at different granularity levels to better understand how software developers perform restructuring on their daily basis.* To achieve this goal, we establish the following objectives:

1. To study official subsequent releases of software systems to understand the high-level architectural evolution of software systems. By performing this empirical study, we can validate whether real-world software developers design systems that respect existing metrics of architectural quality. Moreover, we can study the disruption caused by state of the art approaches for high-level restructuring to propose new approaches for automated architectural improvement that takes the disruption caused by the tool into account during the improvement process.

2. To study data from the code review process to understand how developers perform architectural modification on their daily basis. By performing this empirical study, we can identify not only the most common situations in which developers perform architectural changes but also whether such changes improve or degrade the systems' structural quality. Moreover, by studying code review data we can obtain insights on how developers perform restructuring on their daily basis so that we use this knowledge to propose tools and approaches that better fit currently used development models and processes.

3. To study data from the code review process to understand how developers perform software refactoring on their daily basis. By performing this empirical study, we can identify the context in which developers commonly employ refactoring operations. Since any approach or tool to automate software restructuring will at the end suggest a series of low-level refactorings, we need to understand how developers currently perform refactoring on their daily basis. This way we will be able to propose tools that better fit their habits, increasing the usability and acceptability of such approaches by developers.

1.3 Contributions

This thesis makes contributions to be exploited by both industrial and academic software engineering practitioners. The most important contributions are listed next:

1. This thesis shows that real-world software systems respect metrics of structural architecture quality. Moreover, we show that state of the art tools for search-based software modularisation exhibit high levels of disruption to the existing structure. These are important findings both the research community and tool developers can use when proposing or developing new tools for software architecture improvement and support.

2. Hence, this thesis proposes a new approach for automated software modularisation that employs multiobjective optimisation to improve the structural architecture of a software system while minimising the disruption to the existing architecture. Tool developers can pick up the approach we proposed to develop and/or enhance existing tools for search-based software modularisation.

3. This thesis shows that developers mostly perform architectural changes when implementing new features in the system or while enhancing existing features. Moreover, we show that developers are often not aware of the architectural changes they are suggesting and/or reviewing during code review. These are relevant insights for both academic and industrial practitioners. Academics tend to focus their studies and analysis on instances of refactoring, but we show that most of the time developers perform architectural modifications when working on features. Similarly, industrial practitioners need to be made aware of changes that significantly modify the system's architecture in order to achieve a better reviewing process.

4. This thesis shows that refactoring operations are mostly employed when working on features. Similarly to the previous contribution, we show that refactoring operations

are mostly used when developers are implementing a new feature or enhancing existing ones. This indicates a paradigm shift for the research on software refactoring, where the community needs to acknowledge refactoring not only as a way to improve the system's quality but also as a tool for the preparation of new features and extensions.

5. This thesis provides the largest open dataset to date on search-based modularisation, where other researchers can use to benchmark and compare future approaches for automated modularisation.

6. This thesis provides a comprehensive code review dataset that links code review information with complete copies of the system's code base at the time of review. Such data can be used for research in software engineering that spans beyond the topics covered in this thesis.

1.4 Thesis Organisation

Chapter 2 presents the literature review and background for this thesis. It covers the early work on software architecture and structure, followed by empirical studies and reports on software evolution and architecture degradation. Next, we discuss different metrics proposed along the years to measure architectural quality followed by a literature review on search-based software modularisation. Finally, we discuss software refactoring and the code review process.
Chapter 3 presents an empirical study that investigates state of the art search-based modularisation approaches in the context of software evolution. In this chapter, we address the first set of problem statements discussed in Section 1.1. Moreover, this is the chapter in which we tackle objective number 1 in this thesis (see Section 1.2) and present contributions 1, 2 and 5 (see Section 1.3).
Chapter 4 presents the code review dataset we use in the rest of the thesis. It describes the methodology used to select the systems and the mining process we employed to acquire the data. This chapter discusses the contribution number 6, as discussed above.
Chapter 5 presents an empirical study on how developers perform architectural modification on their daily basis. It covers the second set of problem statements described in this thesis. In this chapter, we address our objective number 2 and present contribution number 3.
Chapter 6 presents an empirical study on how developers employ refactoring operations on their daily basis. It covers the final set of problem statements described in this thesis. In this chapter, we address our objective number 3 and present contribution number 4.
Chapter 7 concludes this thesis and presents directions for future work.

2 Literature Review and Background

This chapter presents the literature review and background of this thesis. We start in Section 2.1 by discussing the early work and foundations of research and practice on software architecture. In Section 2.2, we discuss the problem of architectural degradation during software evolution followed by work which presented empirical evidence of architectural degradation in real-world software systems. Additionally, we present and discuss problems caused by architectural debt, which is a side effect of architectural degradation.

Next, in Section 2.3, we present software architecture and modularisation metrics that have been proposed, evaluated, and used in the literature. We focus our discussion on metrics that have been used to measure architectural quality, and consequently, employed as fitness functions for search-based software modularisation. We also discuss metrics that have been proposed to measure architectural change and similarity.

In section 2.4, we present the background and literature on search-based software modularisation as an approach for architectural restructuring. We present the foundations of search-based modularisation and discuss the different scenarios in which search-based modularisation has been evaluated, such as software evolution, multiobjective search-based modularisation and user interactive search-based modularisation. We present related work that attempted to evaluate the disruption caused by search-based modularisation approaches. Moreover, we discuss the work that evaluated the developer's perception of software modularisation.

In addition, in Section 2.5, we discuss the background of software refactoring, with an emphasis on presenting the most commonly employed refactoring operations. Moreover, we discuss the work that compared manual and automated refactoring, as well as work that investigated the developers' intentions behind software refactoring.

Finally, in Section 2.6, we discuss the foundations of modern code review and its adoption in current software development. In addition, we discuss empirical studies that evaluated the usefulness of code review in terms of bug identification, knowledge transfer, and social communication.

2.1 Software Architecture and Modularisation

The concepts and ideas of organising different programs and procedures into independent modules are as old as the field of software engineering itself. The work by Parnas (1972) defines a modularisation as a set of modules, where a module *"is considered to be a responsibility assignment rather than a sub-program"*. This was a paradigm change at the time since most software engineers used to modularise their systems based on the necessary subroutines to achieve the overall goal, where the modularisation was essentially a sequence of input/output steps. The idea of responsibility assignment might be considered the basis for nowaday's programming languages and software development approaches.

In the same work, Parnas (1972) describes a series of criteria to consider in order to proper modularise programs in high-level modules according to the new responsibility assignment paradigm. One of the criteria described in the paper, and perhaps the most relevant and influential, was the concept of 'information hiding', which includes the design of modules' interfaces that will reveal as little as possible from the modules' inner workings. The benefits of this software design model were advocated to include shortened development time, greater flexibility and enhanced comprehensibility.

Parnas (1979) later extended these ideas to what he coined as 'program families'. In this context, a program is part of a family when it shares commonalities among other programs in the same family, which might include subroutines and the usage of certain modules. At that time, common program families were programs that run in different hardware, programs that perform the same functionality but use different formats of input/output, and programs that perform the same functionality but differ in algorithms and data structures. The author claims that the interfaces and commonly used modules should be as generic as possible while individual implementations of modules should be as specialised as possible.

In a highly influential paper, Parnas et al. (1984) identified that commercial software development at that time was not following the software engineering principles defined and advocated by academics, especially in terms of modularisation and separation of concerns. Thus, the authors performed a study that attempted at recreating an in-flight aircraft system by rigorously following the modularisation concepts designed in academia. As a result, even though the academic practices had to be re-evaluated and adapted to cope with such a large-scale system, the authors noticed a higher quality code base in the academic project in comparison to its commercial counterpart. The software developed with academic consultancy was observed to be more comprehensible by new developers and to have an easier traceability between functionalities and bugs to specific modules' implementations.

The initial concepts of software modularisation introduced in the early work described above influenced the work by Garlan and Shaw (1993), which introduced software architecture as a new and emerging area within software development. The work described architectural styles commonly observed in software systems at the time, which include, but are not limited to, pipes and filters, data abstraction and object-orientation, and layered systems. Each architectural style was described in details, with possible applications and systems that would fit each different style.

Software systems quickly evolved to the point where a system would make use of more than one architectural style within its structure. In this context, practitioners and academics noticed that the complete architecture of a system could not be completely and precisely described in a single model. Moreover, the architecture has become a key artefact in the communication and understanding of the system among different stakeholders (Rozanski and Woods, 2011).

Thus, Kruchten (1995) proposed the '4 + 1' view model for software architecture. In this model, the system's architecture has more than one view, where each view is tailored to a different subset of stakeholders. These concepts were later standardised in the IEEE Standard 1471 and the ISO Standard 42010.

The 5 original architectural views were extended to 7, which are now called architectural viewpoints (Rozanski and Woods, 2011). Each of these viewpoints is listed and described as follows.

Development Viewpoint

This architectural viewpoint describes the architecture to support the system's development. This view is commonly described by the source code's structure, such as the classes diagrams and packages distribution. The development view is targeted at stakeholders involved in developing, building and testing the system.

Context Viewpoint

This view describes the system's relationships and interactions with its environment, such as the people and external systems it interacts with.

Functional Viewpoint

The functional viewpoint describes the system's main runtime elements. It defines their responsibilities, interfaces and primary interactions.

Information Viewpoint

This architectural view depicts how the system stores, process and distributes information. It consists of not only database layouts but also models of how the data is processed and coordinated between modules.

Concurrency Viewpoint

This viewpoint describes the concurrency aspects of the system, including the units which run in parallel, the data exchanged between modules, and mechanisms to avoid concurrency issues.

Deployment Viewpoint

The deployment view describes the environment in which the system is deployed. It includes hardware specifications, processing nodes, networking connections and storage facilities.

Operational Viewpoint

This architectural view indicates how the system will be operated, administered, and supported when it is running in its production environment.

For the rest of this thesis, for all systems we discuss and analyse, we focus on the development view of the software's architecture. This is the view that is mostly used by software developers and is the one that drives software comprehensibility, maintainability and evolution. Hence, from now on, we use software architecture and modularisation to describe the source code organisation of a software system, including its high-level decomposition in submodules and/or subcomponents and the dependencies between these elements.

2.2 Software Evolution and Architectural Degradation

Apart from specific cases, software development largely revolves around the concept of evolution and adaptation. Requirements constantly emerge during development and/or production, so that changes need to be constantly made in order to cope with this everchanging scenario that includes new features, improvements to existing features, and bug fixing. Lehman (1979) pointed out this need for adaptation as one of the intrinsic elements of software development and encouraged the software engineering community to embrace it. In fact, processes for incremental software development were created largely to cope with this need for constant adaptation and evolution.

In this seminal work, Lehman (1979) proposed five laws of software evolution that although being revisited later (Lehman, 1996; Lehman et al., 1997), still remain as well accepted concepts by the software engineering community. The laws describe the natural need for change in software systems, alongside the consequent attempts to conserve the stability and familiarity. However, for the context of this research, the 2nd Lehman's law of software evolution is the most relevant:

> *"As an evolving program is continuously changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain it or reduce it".*
> (Lehman, 1979)

In this quote, Lehman refers to the fact that the complexity of software systems will continuously grow as the systems evolve. Although 'complexity' might have different interpretations for different artefacts in a software system, Lehman mentions structural degradation as a proxy for this increased complexity. Indeed, structural architecture degradation is sometimes attributed to one of the biggest factors for the deterioration of a software system as a whole (de Silva and Balasubramaniam, 2012).

The software architecture establishes a crucial foundation for the development and evolution of a software system. It guides the development of new features and the enhancement of existing features during software evolution. As such, the degradation of such important artefact affects several quality properties of a software system under constant changes and evolution, such as maintainability, comprehensibility, and extendability. In the next section, we present empirical evidence of architectural degradation on real-world software systems, followed by problems that arise from a degraded structural architecture.

2.2.1 Empirical Evidence of Software Architectural Degradation

A wide number of case studies indicate that architectural degradation is a constant problem software practitioners have to deal in industry. Eick et al. (2001) present the study of a large C/C++ system over 15 years of development. The authors propose a set of indices to indicate the increase in the structural complexity and the problems caused by such a phenomenon. The data shows a clear relationship between increased degradation in the structural architectural and increased number of bugs and maintenance effort.

Godfrey and Lee (2000) present an attempt to extract the structural architecture of the Mozilla's internet browser from its source code. During this exercise, the authors identified a great number of undesirable dependencies between core modules of the system. Such degradation of Mozilla's structural architecture finally led to a full rewrite of some modules from scratch (van Gurp and Bosch, 2002; van Gurp et al., 2005).

Evidence of structural degradation has also been found for Ant, one of the most popular Java building tools (Barros and Farzat, 2013; Barros et al., 2015) . The authors performed an extensive study based on reverse engineering and data analysis of all releases of Ant until the moment of publication. A large suite of metrics was employed, and the results indicate a large decrease in terms of modularity quality over the years. A visual comparison of the structural architecture of Ant in its first and latest releases presents a clear increase in modular complexity.

Wermelinger et al. (2011) performed a study of the structural architecture of Eclipse's high-level plugins and modules. The authors found evidence of a large increase in modularity complexity and dependencies. Such degradation in modularity was traced back to specific releases of Eclipse. In a subsequent study, the authors contacted the Eclipse developers as a way to get feedback on their analysis. The developers confirmed that the architectural problems found by the authors are well-known issues by the development team and that such problems were indeed introduced at the releases identified by the authors.

Case studies with similar findings have been reported for several other large software systems such as Vim (Godfrey and Lee, 2000), FindBugs (Sutton, 2008) and Linux kernel (van Gurp and Bosch, 2002).

Le et al. (2015) performed a large empirical study involving 14 open source projects, involving popular systems such as Hadoop, Log4j, and Xerces. The authors employed a systematic procedure to extract the structural architecture of each system throughout its development history and measure the amount of change. The results show a large change in the structural architecture during the systems' development history, both at high-level system granularity and low-level components granularity. Although this study does not link the amount of architectural change with a decrease in quality properties, it shows how often architectural changes occur at different level of granularities in software systems.

Evidence of structural architecture degradation has been presented in more papers than we can discuss in this thesis. For the interested reader, we refer to surveys such as the ones performed by de Silva and Balasubramaniam (2012) and Williams and Carver (2010).

2.2.2 Software Architectural Debt

Technical debt is a concept first coined by Cunningham (1993) to describe 'not quite right' decisions during software development. The analogy implies that making bad development decisions have the same effect as taking a financial loan, in which it will benefit the development team in the short term, but also complicate future developments and enhancements if not paid. This analogy has been widely adopted in industry (Ernst et al., 2015) as a way to easily

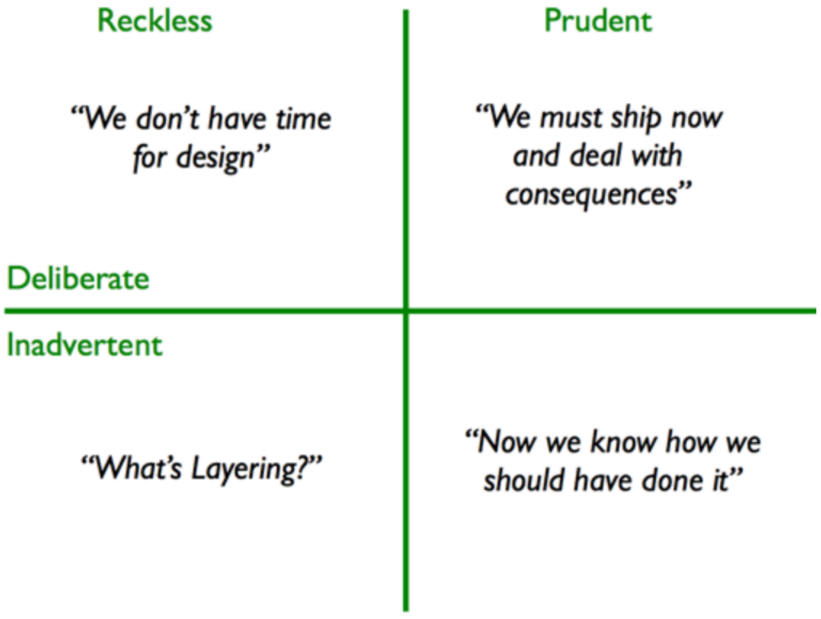

Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

Figure 2.1: Technical debt quadrant (Fowler, 2009)

communicate maintenance costs and decisions to non-technical stakeholders (Brown et al., 2010; Kruchten et al., 2012).

Figure 2.1 presents the technical debt quadrant proposed by Fowler (2009). As one can see, Fowler differentiates decisions in deliberate and inadvertent. Previously, the degradation of a software system was attributed to poor and erroneous decisions made by developers, where such degradation is mostly caused by developers' mistakes. The technical debt analogy brought in the idea that developers might make 'not quite right' decisions deliberately, as an attempt to reach a short-term goal such as delivery time. Thus, the technical debt literature focuses not only on identifying degradation but also the decisions that led to that degradation and the problems caused by such decisions in the long term.

Ernst et al. (2015) performed a large empirical study with software engineering practitioners to investigate their perceptions of technical debt. As a result, developers pointed out architectural decisions as the most expensive debt to pay, and the one with more damaging potential on the evolution of the system. This kind of debt has become known as Architectural Debt.

Xiao et al. (2016) proposed a method to identify and quantify architectural debt in software systems. The approach uses previously evaluated architectural smalls as indicators of architectural anomalies. The authors extract the structural architecture of 7 systems throughout their development history and automatically identify the architectural anomalies. The results show that most of the architectural anomalies tend to aggregate more and more files as the systems evolve. This is referred to as the accumulation of architectural debt. Moreover, the authors show that files involved in the architectural anomalies tend to require more maintenance effort than files that are not involved in architectural anomalies. Finally, the authors used linear regression models to describe the increase in maintenance effort based on the growth of architectural debt, as an analogy to how much interest the developers are paying because of the architectural debts.

A similar study was performed by Mo et al. (2015), where the authors identified 'hotspots patterns' in 10 software systems. These hotspots can be considered sources of architectural debt and architectural smells. Results show that files involved in the hotspots are more bug-prone than files not involved in the hotspots.

2.3 Software Architecture and Modularisation Metrics

This section discusses the most common and used metrics in software architecture and modularisation research. They are divided into quality and similarity metrics.

To better explain each metric, we make use of a module dependency graph (MDG). An MDG is a directed graph $G(C, D)$ where the set of nodes C represents the code elements in the system and D represents the dependencies between code elements. Clusters of nodes in the MDG indicate the system's high-level modules.

The MDG example presented in Figure 2.2 will be used to illustrate the following discussed metrics.

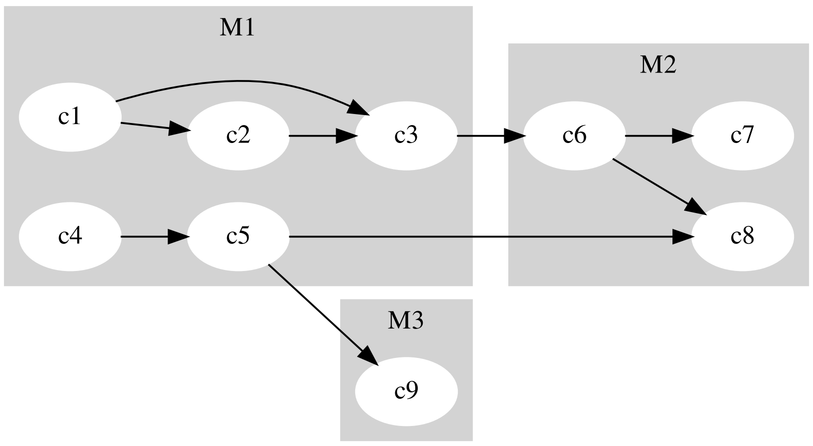


Figure 2.2: Example of a Module Dependency Graph used to represent the modular structure of a software system. Nodes represent code elements and edges represent dependencies between elements. Clusters of nodes (grey regions) indicate high-level modules.

2.3.1 Software Architecture and Modularisation Quality Metrics

The concepts of cohesion and coupling are the among the most important quality criteria in the software architecture and modularisation community. Researchers and practitioners advocate that software entities should be highly cohesive and loosely coupled. However, many different metrics for the computation of cohesion and coupling have been proposed, discussed and evaluated in the literature, where the underlying measurements range from syntactical dependencies between components to semantical relationships and historical co-changes.

Needless to say, there is no standardised and accepted definition of cohesion and coupling. Hence, in this section, we summarise some of the metrics for cohesion and coupling that have been proposed in the software engineering literature. In this section, we do not try to advocate in favour of some metric or another as we aim at simply depicting how wide the range of metrics is. In Chapters 3 and 5, we do make use of a subset of these metrics, where the motivation for the choice of metric is presented at the respective chapter.

We subdivide this section in metrics that attempt to measure cohesion and coupling separately, followed by metrics that combine both concepts in a single measurement. Next, we describe metrics that quantify quality properties in software architecture and modularisation other than cohesion and coupling. Finally, we discuss the theoretical validation of such metrics.

Cohesion Measurements

The following metrics attempt at measuring the concept of cohesion in a given software system.

Raw Cohesion

The simplest way of measuring the cohesion of a modularisation is by the sum of all of the intra-edges of the MDG, i.e., edges that are confined in the same high-level module. This metric is often referred to as raw cohesion and is commonly used as the basis for the computation of more refined cohesion measurements. The values of raw cohesion for the MDG in Figure 2.2, for example, would be $cohesion = 6$.

Lack of Structural Cohesion

In the work by Candela et al. (2016), the authors propose and evaluate new metrics for the computation of structural cohesion and coupling. The Lack of Structural Cohesion is inspired by the original Lack of Cohesion of Methods (LCOM) (Chidamber and Kemerer, 1994). Given a certain modularisation, this metric considers a file as a code element and computes the lack of structural cohesion of the modularisation as:

$$\text{LStrCoh} = \frac{\sum_{j=1}^{n} \text{LCOF}_{M_j}}{n}, \tag{2.1}$$

where LCOF_{M_j} represents the Lack of Cohesion of Files for high-level module M_j. LCOF_{M_j} is computed as the number of pairs of files in M_j without a structural dependency between them. Modules with a high amount of unrelated files will be scored a high LCOF, and, accordingly, modules with only a few unrelated files will be scored a low LCOF. Since this metric measures the *lack* of cohesion in the modularisation, a low value $LStrCoh$ indicates a cohesive modularisation.

Consider the MDG in Figure 2.2, for example. The lack of cohesion of files would be computed as $LCOF(M_1) = 6$, $LCOF(M_2) = 0$ and $LCOF(M_3) = 0$. Therefore, the Lack of Structural Cohesion of this modularisation is computed as $LStrCoh = \frac{6+0+0}{3} = 2$.

Tight Class Cohesion (TCC)
The Tight Class Cohesion metric was originally proposed by Bieman and Kang (1995), where the cohesion is measured at class level by leveraging the connections and dependencies between methods. The TCC cohesion metric is computed as:

$$TCC(C) = \frac{NDC(C)}{N \times \frac{N-1}{2}} \tag{2.2}$$

where $NDC(C)$ represents the number of direct connections, i.e., dependencies between methods, in class C. N indicates the number of methods in class C. Hence, the TCC computes cohesion as the relative number of dependencies between methods in a class given the total number of methods in a class.

Consider each high-level module in Figure 2.2 to be a class in the system, and each component to a method inside the class. The TCC of each class would be computed as $TCC(M1) = \frac{4}{5\times\frac{5-1}{2}} = 0.4$, $TCC(M2) = \frac{2}{3\times\frac{3-1}{2}} = 0.66$ and $TCC(M3) = \frac{0}{1\times\frac{1-1}{2}} = 0$.

Attributes-based Cohesion Metrics
Not all cohesion metrics employ dependencies between components to measure cohesion. Given an object-oriented system, the community has proposed a series of metrics that use the shared attributes between methods as a way to measure how cohesive the methods, and the classes, are.

The Class Cohesion (Bonja and Kidanmariam, 2006), Sensitive Class Cohesion (Fernández and Peña, 2006) and Low-level Similarity Base Class Cohesion (Al Dallal and Briand, 2010) are examples of metrics that employ shared attributes to compute the cohesion of a class. Since the MDG representation does not allow for attributes modelling, we will not depict how these metrics are computed. We point the interested reader to the original papers that propose and describe each metric.

Coupling Measurements

In the following paragraphs, we present metrics that attempt at computing the coupling of a software system.

Raw Coupling
Similarly to the raw cohesion, the raw coupling metric is the simplest way of measuring the coupling of a software system. It is computed as the sum of all of the inter-edges of the MDG, i.e., edges that cut across different high-level modules. The values of raw coupling for the MDG in Figure 2.2, for example, would be $cohesion = 3$.

Coupling Between Objects
The CBO metric has been originally proposed by Chidamber and Kemerer (1994), where the CBO of a certain high-level module is simply given by the number of other high-level modules

that it interacts with. Hence, considering the MDG in Figure 2.2, this metric would be computed as $CBO(M1) = 2$, $CBO(M2) = 1$ and $CBO(M3) = 1$.

Structural Coupling

Candela et al. (2016) used a coupling measurement that is similar to the ones previously presented in this section. In their work, the structural coupling of a modularisation, StrCop, is computed as

$$\text{StrCop} = \frac{\sum_{j=1}^{n} \text{FanOut}_{M_j}}{n}, \tag{2.3}$$

where FanOut_{M_j} indicates the number of files outside module M_j that present dependencies to files inside M_j.

Given the MDG example in Figure 2.2, the FanOut would be computed as $FanOut(M_1) = 3$, $FanOut(M_2) = 2$ and $FanOut(M_3) = 1$. Thus, the Structural Coupling would computed as $StrCop = \frac{3+2+1}{3} = 2$.

Conceptual Coupling Between Classes

Similarly to the cohesion metrics discussed above, there are coupling metrics that employ information other than dependencies between components to measure the coupling of a software system. The CCBC metrics was originally proposed by Poshyvanyk et al. (2009), and it considers the lexical information extracted from comments and identifiers to measure the coupling between two components.

Thus, the CCBC between two source code components is computed as the average textual similarity between the components, where the textual similarity is computed using Latent Semantic Indexing (LSI). The CCBC values are in the $\{0 \ldots 1\}$, where 0 represents two components having a totally different lexical similarity, while 1 represents two classes containing exactly the same text.

Logical Coupling

Finally, it is worthy to mention what has been recently called logical coupling, an approach to measure coupling between objects that rely on versioning and change history. Inspired by the seminal paper of Zimmermann et al. (2005), this approach extracts information from version control systems to identify groups of files that are constantly changed together. This approach has been originally proposed to uncover connections and relationships between files that cannot be found through static or dynamic analysis. Hence, the coupling between two objects is given by how often these files are changed together in the same commit.

Combinations of Cohesion and Coupling

The following sections depict metrics that combine cohesion and coupling measurements into a single quality metric.

Modularisation Quality (MQ)

MQ is a metric originally proposed as the fitness function for search-based software modularisation approaches (Mancoridis et al., 1998). It has evolved over the years so that only the current (and most used) version will be presented here. The formulation of MQ is presented in Equations 2.4 and 2.5.

$$MQ = \sum_{k=1}^{N} MF(M_k) \tag{2.4}$$

$$\text{where, } MF(M_k) = \begin{cases} 0, & \text{if } i_k = 0 \\ \dfrac{i_k}{i_k + \frac{j_k}{2}}, & \text{if } i_k > 0 \end{cases} \tag{2.5}$$

The Modularisation Quality (MQ) of a certain modularisation is given by the sum of the Modularisation Factor (MF) of each module M_k in the modularisation. $MF(M_k)$ represents the trade-off between cohesion and coupling for module k, where i_k is the number of intra-edges within module k (cohesion), and j_k is the number of inter-edges leaving or arriving at module k (coupling). Since inter-edges will be double counted in different modules, j_k is divided by 2.

When considering the MDG example, the values of MF would be $MF(M_1) = 0.72$, $MF(M_2) = 0.66$ and $MF(M_3) = 0$. Consequently, $MQ = 0.72 + 0.66 + 0 = 1.38$. If a module has only a single code element, its MF will be equal to 0 since $i_k = 0$. Therefore, MQ considers not only the trade-off between cohesion and coupling but also the distribution of code elements between modules.

EVM

The EVM metric stands for Evaluation Metric Function, and it was originally proposed by Tucker et al. (2001). Harman et al. (2005) later adapted EVM to be used in the context of software modularisation. EVM's formulation is presented in Equations 2.6, 2.7, and 2.8.

$$EVM = \sum_{k=1}^{N} CS(M_k) \tag{2.6}$$

$$\text{where, } CS(M_k) = \sum_{i=1}^{|M_k|} \sum_{j=i+1}^{|M_k|} USE(i,j) \tag{2.7}$$

$$\text{and, } USE(i,j) = \begin{cases} +1, & \text{if } uses(i,j) \vee uses(j,i) \\ -1, & \text{otherwise} \end{cases} \tag{2.8}$$

The EVM of a certain modularisation is given by the sum of the Cluster Score (CS) of each high-level module M_k. $CS(M_k)$ is computed by iterating over each pair of code elements in module k and incrementing 1 in the score if there is a dependency between these elements; otherwise, the score is decremented by 1.

When considering the example, the values of CS would be $CS(M_1) = -2$, $CS(M_2) = 1$ and $CS(M_3) = 0$. Consequently, $EVM = -2 + 1 + 0 = -1$. Differently from MQ, EVM does not take inter-edges into account.

Quality Metrics other than Cohesion and Coupling

When measuring the quality of an MDG, some authors also showed concerns about the distribution of code elements in high-level modules. In fact, this was one of the criteria used by Wu et al. (2005) to assess the quality of a software modularisation algorithm. In the ideal scenario, one would have a balanced distribution of elements, avoiding both modules with few elements and modules with too many elements.

In order to cope with this, the number of isolated modules can be used as a quality criteria. A module is considered to be isolated when it is composed of a single code element. In the example, module M_3 is an isolated one. Other authors have also used the difference between the number of elements of the biggest and smallest modules as a quality criteria. Considering the MDG example, the biggest module is M_1 with 5 modules, and the smallest one is M_3 with 1 module. Thus, the module size difference would be 4.

Theoretical Validation of Cohesion and Coupling Metrics

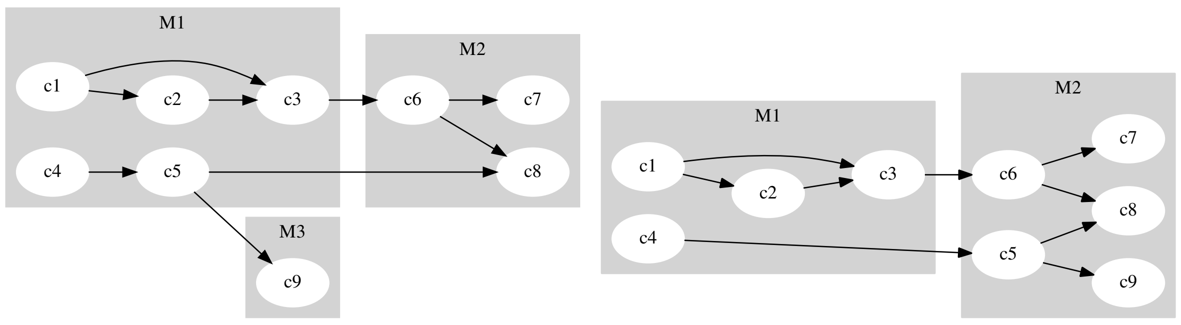
Software measurement is a long-studied topic in software engineering, and as such, many guidelines have been proposed to properly validate a certain metric. In one of the most influential papers on the topic of software metrics, Kitchenham et al. (1995) presents a framework for the proposal and validation of software metrics. The authors describe two methods for the validation of a software metric, e.g., *theoretical* and *empirical*, where the first ensures that the metric respects a series of basic properties and the latter serves as additional support for the proposed measurement.

To the best of our ability, we performed an analysis of the papers that originally proposed the cohesion and coupling metrics discussed above. We have not identified a paper in which the proposed metric was both theoretically and empirically validated according to the guidelines proposed by Kitchenham et al. (1995). In particular, the theoretical validation is the one lacking the most in the studied papers. Very few authors describe the properties which the proposed metrics hold. Even though most papers perform an empirical validation of the metrics, such experiments do not employ the strategies proposed in the guidelines.

Nevertheless, the presented cohesion and coupling metrics figure among the most accepted measurements of such aspects in state-of-the-art software engineering research. In addition, a few of those have been extensively validated through studies with developers, which despite not being sufficient for a thorough validation (Kitchenham et al., 1995), support their use in empirical studies.

2.3.2 Software Architecture and Modularisation Change and Similarity Metrics

Similarity metrics, as the name suggests, are used to measure the similarity between two different modularisations of the same system. The two modularisations presented in Figure 2.3 will be used to illustrate the similarity metrics discussed in the remainder of this section.



(a) Modularisation A (b) Modularisation B

Figure 2.3: Example of different modularisations of the same system

MoJo

$MoJo$ was the first similarity metric proposed for software modularisation (Tzerpos and Holt, 1999). It measures the distance between two modularisations by counting the number of operations necessary to transform one modularisation into the other. $MoJo$ considers only two operations: **Mo**ve and **Jo**in, where the first consists in moving one code element to another high-level module and the second consists in joining two high-level modules. For $MoJo$, both operations have the same weight when computing the similarity. Considering $mno(A, B)$ to be the minimum number of operations to transform modularisation A in modularisation B, the $MoJo(A, B)$ metric is defined in Equation 2.9.

$$MoJo(A, B) = min(mno(A, B), mno(B, A)) \tag{2.9}$$

It is important to notice that $mno(A, B)$ is not always equal to $mno(B, A)$, so the minimum value between the two is taken as the $MoJo$ value. However, when the direction of the comparison is relevant, one should use $mno(A, B)$ as the $MoJo$ value. Considering the example in Figure 2.3, $mno(A, B) = 2$ since it is possible to transform A in B by joining modules M_2 and M_3 and then moving c_5 from M_1 to M_2. Since $mno(B, A)$ is also equal to 2, the $MoJo$ metric is assigned as $MoJo(A, B) = 2$.

The $MoJo$ metric will always have a known minimum and maximum value for a particular system. When A and B are equal modularisations, $MoJo(A, B) = 0$. On the other hand, when A and B modules are placed in completely different clusters, $MoJo(A, B) = N$, where N is the number of modules in the system. Thus, the 'quality' metric Q indicates a normalised version of $MoJo$:

$$Q(A, B) = (1 - \frac{MoJo(A, B)}{N}) \times 100\% \tag{2.10}$$

Finally, the computation of the $MoJo$ metric is not straightforward; therefore, an optimal algorithm to calculate $MoJo$ is provided by Wen and Tzerpos (2003).

EdgeMoJo

As one can notice, the original $MoJo$ does not consider the dependencies between code elements in the similarity calculation. It does not matter how much cohesion and coupling is changed between two different modularisations. Such flaw in $MoJo$ was indicated by Mitchell and Mancoridis (2001). Wen and Tzerpos (2004a) then proposed $EdgeMoJo$, a new version of $MoJo$ that considers dependencies between elements to compute the similarity of two modularisations.

In the $MoJo$ metric, all **Mo**ve operations have the same weight in the similarity computation. Differently, in $EdgeMoJo$, the **Mo**ve operations that alter the cohesion and coupling of the system have a bigger weight than the ones that do not. For each **Mo**ve operation necessary to transform A in B, the respective code element of the operation is classified as an *equally-connected* or a *highly-biased* element. An *equally-connected* element is the one that its change from one high-level module to another do not alter the cohesion and coupling of the modularisation. On the other hand, the change of a *highly-biased* element from one module to another alters the cohesion and coupling of the modularisation. In this context, **Mo**ve operations related to *equally-connected* element have a weight of 1, as in the original $MoJo$. For **Mo**ve operations in *highly-biased* element, the weight w of the operation o is given by Equation 2.11 below:

$$w(o) = 1 + \frac{|E(c_o, M_{new}) - E(c_o, M_{old})|}{E(c_o, M_{new}) + E(c_o, M_{old})} \tag{2.11}$$

where c_o is the code element related to operation o. M_{old} is the high-level module c_o used to belong and M_{new} is the module c_o was moved to. $E(c_o, M_i)$ is the number of edges of c_o related to module M_i.

Consider modularisations A and B in Figure 2.3, for example. Two operations are needed to transform A in B, which are join modules M_2 and M_3 (o_1) and then move c_5 from M_1 to M_2 (o_2). Join operations have a weight of 1, so $w(o_1) = 1$. Since moving c_5 to M_2 alters cohesion and coupling, c_5 is classified as a *highly-biased* element, and this operation will have a weight $w(o_2) = 1 + \frac{|2-1|}{2+1} = 1.33$. Then, $EdgeMoJo(A, B) = w(o_1) + w(o_2) = 2.33$.

Empirical experimentations presented by Wen and Tzerpos (2004a) show that $MoJo$ and $EdgeMoJo$ have a positive linear correlation. Differently from $MoJo$, a normalisation strategy for $EdgeMoJo$ was not provided.

MoJoFM

After using the ‘quality’ metric Q presented in Equation 2.10, the $MoJo$ creators noticed the metric was not suitable for some scenarios. Then, Wen and Tzerpos (2004b) proposed an improved version of this metric called $MoJoFM$. This improved version introduced features to overcome two major shortcomings of the previous metric. $MoJoFM$ is presented in Equation 2.12 below:

$$MoJoFM(A, B) = (1 - \frac{mno(A, B)}{max(mno(\forall A, B))}) \times 100\% \tag{2.12}$$

The first issue is related to the usage of the $MoJo$ distance in the numerator. The minimum number of **Mo**ve and **Jo**in operations necessary to transform a modularisation A in another modularisation B, i.e. $mno(A, B)$, is a non-symmetric measure. Thus, the $MoJo$ value was assigned as the minimum value between $mno(A, B)$ and $mno(B, A)$ (see Equation 2.9). In all considered scenarios, the direction of the relationship is relevant, so $mno(A, B)$ should be used in the numerator instead of $MoJo(A, B)$.

The Q metric used the number of elements in the system to normalise the value, claiming that the maximal number of operations to transform a certain modularisation in another one would always be the number of modules in the system. However, this premise was found not to be true for all scenarios. Therefore, instead of simply using the number of modules for normalisation, $MoJoFM$ performs a new calculation to find the actual maximal number of move and join operations necessary to transform A in B, i.e., $max(mno(\forall A, B))$.

Considering the example in Figure 2.3, the maximal number of move and join operations to transform any modularisation in modularisation B is $max(mno(\forall A, B)) = 6$. Thus, $MoJoFM(A, B) = (1 - \frac{2}{6}) \times 100\% = 66.66\%$.

EdgeSim

After the publication of the original $MoJo$ metric (Tzerpos and Holt, 1999), Mitchell and Mancoridis (2001) argued that a similarity metric for software modularisation should consider not only the assignments of code elements to high-level modules but also the dependencies between elements. In addition, they proposed a new dependencies-based similarity metric for software modularisation called $EdgeSim$, as presented next.

$$EdgeSim(A, B) = \frac{weight(\Upsilon)}{weight(E)} \times 100 \tag{2.13}$$

where Υ denotes the set of edges that are inter- and intra-dependencies in both modularisations A and B. In other words, Υ are the edges that agree in both modularisations. E represents the set of all edges of the system. The $weight()$ function returns the sum of the weights of a set of edges.

The rationale behind $EdgeSim$ is simple: if two modularisations agree for all their edges they are equal. For the given example, two inter-edges in A are intra-edges in B, yielding an $EdgeSim$ similarity value of $EdgeSim = \frac{7}{9} \times 100 = 77.77$.

Despite being simple, $EdgeSim$ may give the same similarity metric for different modularisations. If high-level modules with no inter-edges between them are joined in a single cluster, $EdgeSim$ will not capture this modification. Considering modularisation A of the example, a modularisation with c_9 in M_2 instead of M_3, for example, would not generate any difference in $EdgeSim$ because the dependency between c_5 and c_9 would still be considered an inter-edge.

MeCl

Besides $EdgeSim$, Mitchell and Mancoridis (2001) also proposed another similarity metric based on dependencies between elements. This one is called $MeCl$, which stands for **Me**rge **Cl**usters. Its formulation is showed in Equation 2.14 below:

$$MeCl(A,B) = (1 - \frac{weight(\Upsilon_B)}{weight(E)}) \times 100 \tag{2.14}$$

where Υ_B denotes the set of edges that were turned from intra-edges in modularisation A to inter-edges in modularisation B. E and $weight()$ have the same meaning as in $EdgeSim$.

$MeCl$ shares the same rationale of $EdgeSim$, measuring the similarity between modularisations based on the difference between edges. However, $MeCl$ is slightly different from $EdgeSim$ in the sense that only edges that were turned from intra-edges to inter-edges are taken into account. The idea is that modularisations that transform inter-edges into intra-edges, hence increasing the cohesion of the modularisation, are not supposed to be different. There should only be differences between modularisations when the cohesion is decreased, and consequently, the coupling is increased.

For the considered example, the dependency between c_4 and c_5 was an intra-edge in modularisation A, and it was turned to an inter-edge in modularisation B. Thus, $MeCl(A,B) = (1 - \frac{1}{9}) \times 100 = 88.88$. One should notice that as modularisation B presents a better cohesion than A, $MeCl$ similarity is smaller than $EdgeSim$ similarity, which considers any kind of edge modifications and yielded a similarity value of $EdgeSim(A,B) = 77.77$.

Unfortunately, $MeCl$ has the same shortcoming as $EdgeSim$. It cannot differentiate between two different modularisations that have the same edges distribution but some code elements assigned to different high-level modules.

2.4 search-based Software Modularisation for Architectural Restructuring

search-based software modularisation was initially proposed as an automated technique to improve the structural architecture of a software system. The work in this area has started in 1998, and the community has already investigated these techniques in more ways than we can thoroughly cover and discuss. A comprehensive summary of 34 related work on the area is presented in Table 2.1. For the rest of this section, we discuss the details of the work that most relevantly influenced the research presented in this thesis.

Table 2.1 Related work in search-based Software Modularisation sorted by year of publication

Paper	Year	Optimisation Approach	Fitness Function	Search Algorithm	Number of Different Systems Used	Number of Releases Studied
Mancoridis et al. (1998)	1998	SO	MQ	HC	5	5
Doval et al. (1999)	1999	SO	MQ	GA	1	1
Mancoridis et al. (1999)	1999	SO	MQ	HC	1	2
Mitchell et al. (2001)	2001	SO	MQ	HC	7	7
Harman et al. (2002)	2002	SO	Coh, Cop	HC, GA	7	7
Mitchell et al. (2002)	2002	SO	MQ	HC	5	5
Mahdavi et al. (2003)	2003	SO	MQ	HC	19	19
Mitchell et al. (2003)	2003	SO	MQ	HC	13	13
Harman et al. (2005)	2005	SO	MQ, EVM	HC	6	6
Seng et al. (2005)	2005	SO	Coh, Cop, Complexity, Cycles, Bottlenecks	GGA	1	1
Shokoufandeh et al. (2005)	2005	SO	MQ	HC and Spectral Algorithm	13	13
Mitchell et al. (2006)	2006	SO	MQ	HC	2	2
Mitchell et al. (2007)	2008	SO	MQ	HC	5	5
Abdeen et al. (2009)	2009	SO	Coh, Cop, Cycles	SA	4	4
Mamaghani et al. (2009)	2009	SO	MQ	Hybrid GA	5	5
Praditwong (2011)	2011	SO	MQ	GGA	17	17
Praditwong et al. (2011)	2011	MO	MCA, ECA	Two-Archive GA	17	17
Barros (2012)	2012	MO	MCA, ECA	NSGA-II	13	13
Bavota et al. (2012)	2012	SO and MO	MQ, MCA, ECA	GA, NSGA-II	2	2
Hall et al. (2012)	2012	SO	MQ	HC	5	5
Abdeen et al. (2013)	2013	MO	Coh, Cop, Modifications	NSGA-II	4	4
Kumari et al. (2013)	2013	MO	MCA, ECA	Hyper-heuristics	6	6
Ouni et al. (2013)	2013	MO	Fixed Bugs, Effort	NSGA-II	6	6
Hall et al. (2014)	2014	MO	MQ	HC	4	4
Barros et al. (2015)	2015	SO	MQ, EVM	HC	1	24
Jeet et al. (2015)	2015	SO	MQ	BHGA	6	6
Mkaouer et al. (2015)	2015	MO	Coh, Cop, MO, NCP, NP, SP, NCH, CHC	NSGA-III,IBEA, MOEA/D	5	5
Saeidi et al. (2015)	2015	SO and MO	MQ, CQ	HC, Two-Archive GA	10	10
Candela et al. (2016)	2016	MO	Structural and Contextual Coh/Cop	NSGA-II	100	100
Huang et al. (2016)	2016	SO and MO	MQ, MCA, ECA	MAEA-SMCPs, GGA, GNE	17	17
Huang et al. (2016)	2016	SO	MQ, MS	HC, GAs and MAEA	17	17
Jeet et al. (2016)	2016	SO	MQ	HC, five GA variations	7	7
Kumari et al. (2016)	2016	MO	MCA, ECA	Hyper-heuristics	12	12
Ouni et al. (2016)	2016	MO	Bugs, Coherence, Effort, Change History	NSGA-II	6	12

First, we present the foundations of search-based software modularisation as an approach that uses search algorithms to restructure a software system. Second, we explore the work that evaluated these techniques in the context of software evolution. Third, we discuss the work that adapted multiobjective optimisation approaches for search-based modularisation. Finally, we present empirical studies that assessed the effort necessary to apply the changes proposed by search-based modularisation approaches.

2.4.1 Foundations of search-based Modularisation

search-based software modularisation was first proposed in the seminal paper by Mancoridis et al. (1998), where it was proposed as an approach to improve system comprehension. The motivation behind their approach lies in the scenario where a software engineer must perform maintenance or improvement activities on a system with outdated or even absent documentation. With no artefacts to gain insight into the system, the developer has to make modifications without a full understanding of the system structure. As the times passes, the source code organisation may degrade to a point where the system will need to be completely restructured or abandoned.

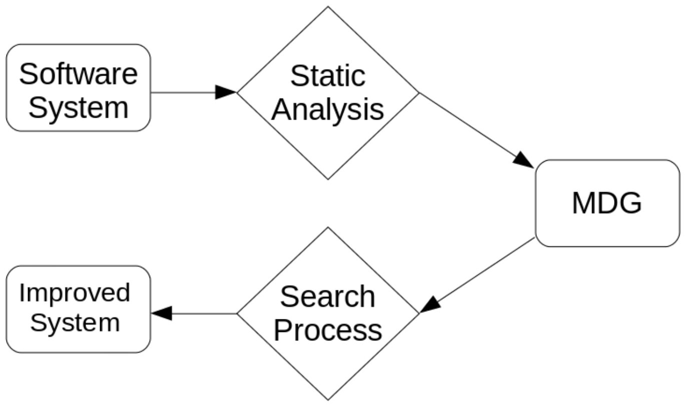


Figure 2.4: Generic framework for search-based software modularisation

In search-based modularisation, the structural architecture of a software system is commonly represented as a Module Dependency Graph (MDG) (Mancoridis et al., 1998). As previously discussed, an MDG is a directed graph $G(C, D)$ where the set of nodes C represents the code elements of the system and D represents the dependencies between code elements. Usually, software systems are organised in higher-modules, which are indicated as clusters of nodes in the MDG. An example of an MDG is presented in Figure 2.2.

An MDG can be either weighted or unweighted. In a weighted MDG, the edges have weights that represent the strength of the dependencies between the code elements. MDGs with no weights on the edges are considered unweighted.

A generic framework for search-based modularisation is depicted in Figure 2.4. The first step consists in extracting the system's MDG. This is usually accomplished through static analysis of either source or compiled code. A search algorithm is then applied in the system's MDG, generating an improved version of the system, in terms of its modular structure. As a result, an improved modularisation provides a better overall picture of the system, which might help developers to perform future maintenance and modifications tasks.

The search algorithm is guided by a fitness function that differentiates solutions in terms of how well the code elements are organised in high-level modules. The most common metrics employed as fitness functions in the search-based modularisation literature were described in more details in Section 2.3.

The search-based modularisation tool developed by Mancoridis et al. (1998) is called Bunch. It uses the Modularisation Quality (MQ) metric as fitness function for the search algorithms. Bunch implements three different search algorithms that can be selected by the user: optimal clustering, hill climbing (HC) and genetic algorithm (GA). The optimal clustering algorithm performs an exhaustive search, selecting the solution with very best MQ value. Since software modularisation is an NP-hard problem (Mitchell, 2002), the optimal clustering algorithm only scales for systems with a small number of modules.

For systems with a large number of modules, non-optimal heuristics are employed. The traditional hill climbing algorithm starts from a random modularisation and searches for neighbour solutions with better MQ until no neighbour solution is better than the current one. The Genetic Algorithm uses Darwin's concepts of natural selection to evolve a population of modularisations towards individuals with better MQ. At the end of the evolutionary process, the modularisation with better MQ is selected as the result.

Bunch was originally applied to a File System service developed in C/C++. The modularisation results were validated by the original developers of the system, where they acknowledged the generated high-level modules as a good description of the system's structural architecture. Bunch was also used to modularise other four systems of various size, where three are open source and one is proprietary. The HC algorithm outperformed the GA for all considered systems.

The Bunch tool was made available for download, and a series of papers related to Bunch have been published, where new evaluations and enhancements were presented. A subset of these papers is discussed next.

Doval et al. (1999) proposed a new GA approach for search-based module modularisation. Although it does not explicitly state it was developed for Bunch, the paper shares the same authors as Bunch, and the proposed GA was later officially included in Bunch. The fitness function used to guide the new GA was the MQ metric, as it also was in Mancoridis et al. (1998). The approach was applied to a single system.

The improved version of the system was compared to the original modularisation presented in the system's documentation. The search-based approach guided by MQ generated a design that was similar to the original one. An interesting observation about this work is that the parameters used for the GA (crossover and mutation rates, population size and number of generations) are still the ones used for state-of-the-art GA approaches for search-based modularisation. However, such parameter setting was not well discussed and evaluated in this particular paper.

After Bunch's original publication, Mancoridis et al. (1999) reported improvements and new features added to Bunch. Based on user feedback, they introduced two new scenarios that might occur in the search-based modularisation environment. They noticed that some systems have certain code elements that present way more dependencies than the average. Such elements are called omnipresent because they do not seem to belong to any particular high-level module, but rather to the whole system. When an omnipresent element depends on many other elements, it is called *client*. On the other hand, when many code elements depend on a single omnipresent element, the latter is called *supplier*. Omnipresent elements are identified based on thresholds, where a threshold $o_t = 3$, for example, would identify every element with 3 times more dependencies than the average as omnipresent. Bunch also allows the user to explicitly indicate which elements should be considered omnipresent. After the identification of omnipresent elements, they are isolated from the MDG, and the search algorithm will not consider them during the optimisation process.

Bunch users also reported issues when using Bunch for incremental software maintenance. After the application of the clustering tool, they would like to continue the development of the system in a way the modularisation would be preserved as much as possible. Thus, Mancoridis et al. (1999) introduced the orphan element concept to Bunch. An orphan element is either a new element being added to the system or an existing element with modified dependencies. Since the developers wanted to minimise the number of changes, the previous modularisation of the system was used as input, and the orphan elements were added to the high-level modules that would produce the highest MQ value.

The improved Bunch tool was applied to two subsequent versions of an open source system.

The first version was used to perform the modularisation considering omnipresent elements, while the subsequent version was used to validate the orphan elements adoption strategy. As a result, when the omnipresent elements were isolated, Bunch could find a much better modularisation of the system. Regarding the orphan elements, they were assigned to the correct hight level modules, according to the authors of the paper. This was the first work to apply a search-based modularisation approach to more than one version of the same system.

Software modularisation is an NP-hard problem, and depending on the size of the MDG being optimised, the search algorithms may take several minutes to yield a result. Aiming at boosting the speed of Bunch, Mitchell et al. (2001) proposed an architecture for distributing the computation of the search-based algorithms. In the case of Bunch, the HC algorithm was adapted to be executed in parallel. The approach employed several processors in parallel, each of which searching for the best neighbours of a particular node. The best neighbour, in terms of MQ, found by all processors is selected as the current solution. Such a procedure is repeated until no better solution is found. The new Bunch HC algorithm was applied to both open source and proprietary systems.

The parallel HC was compared to its previous non-parallel version, achieving the same quality in the solutions and speedups up to 6 fold. In addition, the parallel architecture was able to use, on average, 90% of the available processing power. The MQ metric used as fitness function in this paper was slightly different than the original MQ used in previous papers. The authors argued this new MQ is simpler than the previous one and yields better results.

Although the HC algorithm presented better results when compared to the GA, a well-known problem of HC algorithms is that certain initial points may lead to local optimal solutions. Thus, Mitchell and Mancoridis (2002) proposed a Simulated Annealing (SA) enhancement for the HC algorithm in Bunch. The SA improvement allows the HC algorithm to accept a worse solution in terms of MQ, so that the algorithm may escape from a local optima. The probability of accepting a worse solution depends on the 'temperature' of the process, where the higher the temperature (beginning of the algorithm) the higher the probability to accept a solution with worse MQ. The temperature is slightly decreased with each iteration of the algorithm. The new SA-improved algorithm was compared to the original Bunch's HC for 5 software systems.

Since the SA algorithm has parameters, i.e., initial temperature and cooling rate, one of the evaluations was the impact of the parameter setting on the final result. In general, none of the SA versions (different parameters) presented better results than HC, which was a surprising result. However, some parameters configurations have speeded up the convergence process of the algorithm for some systems, but not for all of them. In summary, the results did not present enough evidence to support that the SA-enhanced algorithm is better than the original HC. The MQ proposed in this work is the one being used by state-of-the-art research in search-based modularisation to date.

Mahdavi et al. (2003) proposed a multi-stage HC algorithm to partition the system into meaningful subsystems. In the first stage of the algorithm, a set of HCs is performed in order to identify 'building blocks' of code elements, which are sets of elements clustered together in most of the HCs runs. In the second and final stage, the building blocks are then considered one single element, and an HC similar to Bunch is executed to cluster the remaining elements.

When compared to the HC algorithm without the building blocks stage, the proposed approach performed better in terms of both runtime and quality. By searching for building blocks at first, the search space is considerably reduced for the final stage, which allows the algorithm to run faster and also find solutions with higher quality. The MQ used in this work was the one proposed by Mitchell and Mancoridis (2002).

Harman et al. (2005) performed an empirical study of the robustness of search-based modularisation approaches in the presence of noisy data. The study used an HC approach being guided by MQ, i.e., the fitness function used in Bunch papers, and EVM, a new quality metric and fitness function proposed in this paper. The authors considered misplaced dependencies between code elements as noise in the data. The authors argue this is a common scenario when developers have to perform maintenance and improvements tasks on systems they do not fully understand. As case studies, they considered real, random and perfect MDGs. The real systems were the ones regularly used to validate Bunch's work. Random MDGs were created completely at random, and perfect MDGs were created in a way the optimal modularisation exists and it is known. Noise in the data was modelled by mutating the MDGs according to a specific parameter that controlled the amount of noise introduced to the graph.

Results indicated the EVM-guided search was able to find the optimal modularisation for the perfect MDGs, while the MQ-guided search was not. Regarding the robustness of the two fitness functions when noisy data was considered, the EVM-guided search was considered the more robust for all MDGs evaluated.

2.4.2 Software Evolution and search-based Modularisation

As previously described in the previous section, much the of the work on search-based modularisation considers only a single version of the system. However, software systems are often developed incrementally, where each new version of the system released to the client includes new functionalities and/or improves previous ones. In the rest of the section, we discuss the work that evaluated search-based modularisation techniques in the context of the incremental evolution of software systems.

Wu et al. (2005) performed a comparison study of different modularisation algorithms when applied to subsequent versions of a system. They considered not only search-based algorithms (Bunch) but also algorithms based on distance coefficients to perform hierarchical clustering (Anquetil and Lethbridge, 1999) and comprehension-driven clustering techniques (Tzerpos and Holt, 2000). Six different algorithms were compared in terms of three quality criteria: *stability, authoritativeness* and *extremity of cluster distribution*.

Stability is related to the ability of the modularisation algorithm to produce similar modularisations for similar versions of the system. *Authoritativeness* assesses whether the modularisation algorithm can produce a solution similar to a 'gold modularisation' of the system. Finally, *extremity of modules distribution* evaluates the algorithm behaviour in terms of creating modules with too many code elements or modules with too few code elements. The study was performed in 5 open source systems. Subsequent versions of each system were directly extracted from the systems repositories, where one MDG was created for each month of development, creating several monthly versions of each system.

In terms of *stability*, Bunch was ranked as the less stable algorithm, where the other 5 were considered stable algorithms. For *authoritativeness*, they considered the gold standard as the original modularisation of the system, so the modularisation algorithms had to generate a solution that was similar to the original one. None of the algorithms performed well for this criteria. Bunch was the algorithm that produced the most balanced solutions, with modules of roughly the same size, where the other algorithms tended to create too big or too small high-level modules.

Barros et al. (2015) performed a structural architecture evolution study of the open source system *Ant*, and then applied search-based modularisation to the latest release of the system to check whether it was possible to bring the structural architecture of the system back to the simplicity of its previous releases. The data was extracted from 24 official releases of *Ant*, comprising a development time of roughly 14 years. The architecture evolution was assessed through several metrics, including the number of packages, the number of dependencies, MQ and EVM. While the first two are metrics related to the size of the system, the former two are modularisation quality metrics usually employed in search-based modularisation approaches (see Section 2.3).

The architecture evolution study showed, as expected, that the system's architecture tends to become more complex as the system evolves. For almost all new releases, more packages, classes, and dependencies are added. As a result, quality metrics such as MQ and EVM tend to get worse, causing the modularisation of the final releases to be much more complex than the modularisation of the first releases of the system.

With this scenario as background, they applied search-based modularisation to the latest release of *Ant* in order to assess if a search-based approach could optimise the system in a way it would become as simple as it was in the first release. They applied an HC based algorithm similar to Bunch, using both MQ and EVM as fitness functions, creating an MQ-optimised and an EVM-optimised version of *Ant*. Although the optimised versions showed better values for search-based modularisation metrics, they achieved quite poor results in terms of general architectural metrics, such as code and architecture smells, for example. In fact, the plotted modularisation of the optimised versions seemed much more complex than the original modularisation. The conclusion was that search-based approaches were not able to optimise the current modularisation of *Ant* so it would become as simple as it was in the first releases of the system.

Although the conclusions drawn from the study are valid based on what was assessed, we believe the final outcome was likely to happen. *Ant* evolved from a system with 4 packages and 102 classes in its first release, to a system with 60 packages and 1116 classes in its latest one. The system has been developed for almost 14 years, and in our point of view, it is virtually impossible to optimise the modularisation of the latest release to make it as simple as it was in the first release. The system is already too different, with different developers, different users, and different features. Therefore, we believe search-based modularisation should be applied to subsequent releases of a system because such a scenario would be much more similar to a real incremental software design process than the one considered by Barros et al. (2015).

2.4.3 Multi-objective search-based Modularisation

All of the work discussed in previous sections employs a single objective optimisation approach, where the user provides the system as an input, and the approach returns a single optimised modularisation of the system as an output. Most of the fitness functions used to guide the search are based on a combination of cohesion and coupling between modules, where the two metrics are combined into one equation so that a trade-off is achieved. Although these are arguably the most important drivers for software modularisation, other metrics may arise depending on the user needs, such as the number of high-level modules, the size of the modules and so on. A combination of all these metrics in a single function would generate the ‘apples and oranges’ problem, where it is not possible to normalise the metrics in a meaningful way so that they cannot be compared or combined.

Multiobjective search (Deb, 2014) is an optimisation paradigm to solve problems with more than one fitness function, where the objectives are usually conflicting and/or non-comparable. It uses the Pareto optimality concept to search not for one single optimal solution, but for a set of optimal solutions. In multiobjective search, a certain solution dominates (is better than) another one if it is better or equal for all objectives and better for at least one of the objectives. A solution that is not dominated by any other solution is called non-dominated, and the set of non-dominated solutions is called Pareto front. A multiobjective search algorithm returns a set of solutions that present the optimal or near-optimal trade-offs between objectives so that the user can select the solution that better fits one needs.

The first adaptation of search-based modularisation for the multiobjective optimisation paradigm was proposed by Praditwong et al. (2011). Two different multiobjective approaches with different modularisation purposes were evaluated. The Maximizing Cluster Approach (MCA) tries to modularise the system by creating as many high-level modules as possible while optimising the quality metrics. The fitness functions are: *cohesion (max), coupling (min), number of modules (max), number of isolated modules (min)* and *MQ (max)*. In this work, cohesion was considered to be the intra-dependencies within modules, and coupling to be the inter-dependencies between modules. Since one of the objectives to maximise is the number of modules, one of the optimal solutions would be the assignment of each code element to a different module. Hence, the approach also tries to minimise the number of isolated modules, i.e., modules with a single code element. The MQ metric was also included as an objective because it was extensively used in the single objective approaches to search-based modularisation.

The other multiobjective approach evaluated was the Equal-size Cluster Approach (ECA), where the aim is to modularize the system by creating high-level modules of roughly the same size. The fitness functions are: *cohesion (max), coupling (min), number of modules (max), difference of max and min module size (min)* and *MQ (max)*. This approach also tries to maximise the number of modules, but in a way that the number of code elements in each module is similar. In order to do this, one of the objectives is the minimisation of the difference between the number of elements of the biggest module and the number of elements of the smallest module in the modularisation. Both MCA and ECA were implemented in the Two-Archive Genetic Algorithm (Praditwong and Yao, 2006) and evaluated against well-known search-based modularisation datasets.

The multiobjective approaches were compared between themselves and also to the Bunch tool. Regarding MCA and Bunch, the first performed better for weighted MDGs while the latter had the best performance for unweighted MDGs. For ECA and Bunch, ECA was able to outperform Bunch for both weighted and unweighted MDGs. When MCA and ECA were compared, ECA also outperformed MCA for most of the MDGs. All results were validated through statistical analyses. Although the ECA approach had shown the best results in terms of quality metrics, when the Pareto fronts found by both MCA and ECA were plotted together, there was a clear division on the search space between MCA and ECA solutions. Thus, the results suggest these two different approaches tend to explore different areas of the search space, indicating a possible need to use both approaches in order to maximize the search space coverage. Even though the multiobjective approaches presented better results than Bunch in most of the MDGs, the amount of time they take to run is much longer than Bunch, which may undermine their usage.

Abdeen et al. (2013) applied multiobjective modularisation to package structure restructuring. The main idea is to improve the modularisation's quality metrics between packages while preserving the current system structure. In this work, system packages are mapped to high-level modules, classes to code elements and method calls to dependencies. The fitness functions used were *cohesion (max), coupling (min)* and *similarity to current design (max)*. The search algorithm employed was the NSGA-II (Deb et al., 2002), and the approach was applied to four open source systems.

Results show that it is possible to improve the cohesion and coupling quality metrics while performing few modifications to the current design. However, the proposed metrics of cohesion and coupling diverge from the standard metrics previously proposed and evaluated in the literature. In this work, cohesion is actually a combined function of 'packages cohesion' and 'classes cohesion', which are not comparable metrics. The measurement is similar for coupling: a combination of 'package' and 'class' coupling. Furthermore, the similarity to the original structure was not measured using standard software modularisation similarity metrics proposed in the literature (see Section 2.3). They proposed their own similarity metric that was not well discussed in the paper.

Barros (2012) performed an analysis of the effects of using composite objectives in multi-objective search-based modularisation. By composite objectives, the author considers fitness functions that are actually compositions of more than one metric. The ECA approach (Praditwong et al., 2011), for example, uses MQ as one of its fitness functions, where MQ is a combination of cohesion and coupling, which are also considered as objectives in ECA. The main objective of this paper is the assessment of the effects of such 'objectives redundancy'. Three versions of the ECA approach were evaluated: the original one (*cohesion, coupling, number of modules, difference modules size, MQ*), a version that replaces MQ by EVM (*cohesion, coupling, number of modules, difference modules size, EVM*) and a version without any composite objective (*cohesion, coupling, number of modules, difference modules size*). The different ECA versions were applied to both open source and proprietary systems.

Interestingly, results show that the EVM-ECA version performs better than the MQ-ECA version as it can find solutions with better quality in less time. When composite objectives are not considered, the algorithm can find statistically equivalent solutions as the EVM-ECA version in even less time. Such results suggest that composite objectives are indeed a redundancy in the search process, taking runtime to be computed and not leading to different areas of the search space. In fact, even when MQ is suppressed from the objectives, solutions found by the multiobjective approach are better in terms of MQ than solutions found by Bunch.

2.4.4 User Interactive search-based Modularisation

For all of the search-based modularisation work discussed so far (both single and multiobjective), the search process is fully and automatically guided by the fitness function(s). Although such metrics are proved to be relevant in representing good quality modularisations, developers have subjective knowledge or intuition about the system they are developing. In order to incorporate the software engineer knowledge during the search process, interactive optimisation can be employed. This optimisation paradigm integrates the user into the search process during the algorithm evolution, usually by incorporating user feedback into the fitness evaluation.

Bavota et al. (2012) proposed an interactive approach for software modularisation. In this approach, the user is asked for feedback during the algorithm evolution. Based on a certain solution presented to the software engineer during the optimisation process, the user may indicate constraints such as which code elements must be kept together, which elements must be placed in different high-level modules or to which module a certain element must be assigned. The algorithm repairs the solution to meet the user constraints. In addition, the constraints indicated by the user are used to penalise solutions in future generations of the algorithm. The number of times the algorithm will ask for user feedback is one of the approach's parameters. The user interaction procedure described above was incorporated in both single (Doval et al., 1999) and multiobjective (Praditwong et al., 2011) GAs for software modularisation. Finally, the approach was validated in two different systems.

Since the developers of the systems under study were not available for validation, a user simulator was proposed. The simulator used the current system modularisation as a 'gold standard' of the system, so it replicates a user that does not want the system to change too much after the optimisation process. Results show that the interactive versions of the GAs were able to present a better approximation of the original modularisation than their non-interactive counterparts. This suggests that the interactive GA is able to incorporate user knowledge in the search process. From our knowledge, this was the first paper to propose an user-in-the-loop interactive approach to search-based modularisation.

2.4.5 Disruption Caused by search-based Modularisation Approaches

The search-based modularisation approaches discussed in the previous section, both single and multiobjective, take a whole software system as input and perform a complete restructuring of the code base as an attempt to improve structural architecture quality. Although a large scale re-modularisation is sometimes necessary due to extensive architectural degradation (van Gurp and Bosch, 2002; van Gurp et al., 2005), small increments and fixes are more desirable for the retention of familiarity (Lehman, 1979; Lehman et al., 1997). Moreover, developers tend to avoid performing changes to code that is already architecturally stable. This is a re-occurring phenomenon in software development, where developers rarely touch code that is not related to their current task, regardless of the (lack of) quality of this previous code (Kapser and Godfrey, 2006). Finally, restructuring activities are difficult to convey as an important development activity because most of the stakeholders will rather prioritise the development of new features and the enhancement of existing features. In this context, the disruption caused by search-based modularisation approaches to the existing modular structure of the software system under optimisation is an important factor to be considered.

The disruption caused by a search-based modularisation technique can be measured as the number of change developers need to perform in order to adopt the solution proposed by the search algorithm, where such disruption may be assessed at both source code and modular structure levels. A previous study by Hall et al. (2014) measured how many lines of code ought to be added/changed in order to apply solutions found by search-based modularisation approaches. Apart from showing that developers would have to change up to 10% of their code base to adopt a solution proposed by automated approaches, they also showed that the LOC to be changed strongly correlates with the modular structure disruption metric MoJoFM (Wen and Tzerpos, 2004b).

Mkaouer et al. (2015) used the number of refactoring operations as a measurement of system disruption to be minimised in a search-based many-objective approach for re-modularisation. However, operations at different granularity levels, such as *move method* and *move class*, have the same weight in the disruption computation, even though coarse-grained and fine-grained refactoring operations have a different impact in the system's modularity.

In the work by Ouni et al. (2013) and Ouni et al. (2016), the authors proposed a disruption assessment of refactoring operations based on the number of operations to be performed, where each operation is weighted by a complexity factor. In this formulation, the possible refactoring operations also include different granularity levels, e.g. *pull up method* and *extract class*, where the different weights are based on the authors' expertise.

2.4.6 Practitioners' Perception of Software Modularisation

All the work presented so far has been developed inside the academic community, and although some of the papers we discussed have performed a validation with software developers, the community needs to first and foremost understand the practitioners' perception of software modularisation in order to propose approaches that will be seen as beneficial by developers. Thus, a few qualitative studies have been carried out in the literature as an attempt to capture developers needs and perceptions regarding software modularisation and architectural quality.

Bavota et al. (2013) performed an empirical study that compares developers' perception of software coupling, and different automated measurements of software coupling. The authors considered 4 different types of coupling measurements, namely structural, semantical, dynamic, and logical. These coupling measurements were used to measure the coupling between pairs of files of 3 different systems. A subset of pairs of files was selected based on how strongly coupled they were evaluated by each automated coupling metric. The pair of files with strong and weak coupling, regarding each metric, were evaluated by developers who were asked to assess the coupling strength of each pair according to their subjective judgement. The results indicate that structural and semantical coupling are the automated metrics that better capture the developers' perception of coupling.

In a related study by Candela et al. (2016), the authors applied multiobjective search-based modularisation to improve the structural and conceptual cohesion and coupling of 100 software systems. After the optimisation process, the improved modularisations were compared to the original modularisations implemented by developers. The MoJoFM metric (Wen and Tzerpos, 2004b) was employed for the comparison. As a result, the original developers' modularisation is more similar to the cohesion-optimised modularisations than to the coupling-optimised modularisations for most of the systems under study. The authors claim this is an indication that developers tend to give higher priority to cohesion over coupling when modularising their software systems.

Simons et al. (2015) performed a study focused on assessing how software developers evaluate the quality of a certain design in comparison to the standard QMOOD suite of metrics for design quality (Bansiya and Davis, 2002). The authors created a set of toy classes' designs and asked a group of experienced developers to evaluate the designs regarding one of QMOOD's quality criteria using their own subjective judgement. Later, the same quality criteria were measured using the QMOOD metrics, and the results between the automated measurements and the developers' subjective assessments were compared. Results show negative to none correlation between the developers' assessment and the metrics measurement. This indicates that the metrics do not capture the subjective perception of the developers in regard to the quality criteria being assessed.

2.5 Software Refactoring

Software refactoring is *"the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure"* (Fowler et al., 1999). Refactoring activities are widely employed by software practitioners during the software development lifecycle, and researchers have already linked software refactoring to improvements in adaptability, maintainability, understandability (Ammerlaan et al., 2015), reusability, testability (Alshayeb, 2009), and productivity (Moser et al., 2008).

The research on software refactoring is vast, and spans multiple areas that include, but are not limited to, refactoring detection (Tsantalis et al., 2018; Silva and Valente, 2017), software evolution (Bavota et al., 2015; Kim et al., 2011), library adaptation (Henkel and Diwan, 2005), software merging (Dig et al., 2008) and code completion (Foster et al., 2012). Thus, this section focuses on the work related to the refactoring investigations performed in this thesis. First, we describe the most commonly used refactoring operations by developers. Second, we discuss studies that investigate the difference between manual and automated refactorings. Finally, we present the existing work on the developers' intentions behind software refactoring.

2.5.1 Refactoring Operations

In their seminal book, Fowler et al. (1999) present a catalogue of 72 refactoring operations for Java systems that include the motivation for each refactoring and the steps in which each refactoring should be performed. Although relevant, recent empirical studies (Murphy-Hill et al., 2012) have shown that not all of these refactoring operations are employed by developers. Based on these observations, tools for automated refactoring (Fokaefs et al., 2011) and tools for refactoring detection (Tsantalis et al., 2018; Silva and Valente, 2017) only support a subset of these refactoring operations. Hence, we limit this section to discuss the refactoring operations that are more often used by developers and supported by automated tools as presented in recent empirical studies and discussed in the state-of-the-art literature (Murphy-Hill et al., 2012;

Negara et al., 2013; Silva et al., 2016; Palomba et al., 2018; Fokaefs et al., 2011; Tsantalis et al., 2018; Silva and Valente, 2017; Kim et al., 2011).

Move Class

This refactoring is often performed when a certain class does not belong to the package it is currently allocated to. In this scenario, a developer moves the entire class from one package to a more appropriate one.

Rename Class

More often than not, the name of a class does not properly denote its responsibility. Hence, this refactoring operation changes the name of the class to a name that better describes its functionality.

Extract Class/Interface

Classes tend to grow as the system evolves, which may cause a class to be performing more than one functionality or be representing more than one concern. A developer extracts a class when she takes part of the code (methods and attributes) from a certain class that represents a certain functionality and creates a new class to better encapsulate this behaviour. An extract interface refactoring occurs when the existing code is moved to a new interface instead of a new class.

Extract Method

Similarly to the extract class refactoring described above, an extract method refactoring should be employed when a method is performing more than a single function. In this case, the developer takes part of an existing method and creates a new method that performs the extracted function. Note that the method might be extracted to the same class or to a different one.

Inline Method

An inline method refactoring should be applied when a method is so simple that it doesn't need to be encapsulated in a different method. This commonly happens for methods that have a single `return` statement in its body. In this scenario, a developer simply deletes the short method and moves its code to an existing method.

Pull Up Method

The pull up method refactoring is a mechanism to remove code duplication. Consider an inheritance tree. If two or more subclasses implement the same or very similar method, this method should be moved to the superclass. Hence, all subclasses will exhibit the desired behaviour but the method will only be declared once in the superclass.

Push Down Method

This refactoring is the opposite of the pull up method operation described above. A developer should push down a method when a behaviour described in a superclass is only relevant for one of its subclasses. Thus, the method should be moved from the superclass to the subclass.

Rename Method

Similarly to rename class, sometimes methods evolve and their name does not accurately express their functionality. In this case, developers change the name of the method to a more descriptive one.

Move Method

A move method refactoring should be employed when a certain method does not belong to the class it is currently allocated to. Hence, a developer will move the method to a different class.

Pull Up Attribute

Also called pull up field. Similarly to the pull up method, if all subclasses in an inheritance tree define the same attribute, this attribute should be moved to the superclass to avoid duplication.

Push Down Attribute

This refactoring should be performed when an attribute defined in a superclass is only used by one of its subclasses. Hence, the attribute will be moved from the superclass to the subclass.

Move Attribute

The move attribute refactoring, also known as move field refactoring, should be performed when an attribute does not belong to the class it is currently defined. In this case, the attribute will be moved to a more appropriate class.

2.5.2 Manual Refactoring vs Automated Refactoring

The refactoring operations discussed above were originally described as a series of manual steps and transformations a developer would perform in the system (Fowler et al., 1999). As the practice of refactoring became popular between software developers, researchers and tool builders started providing tools and approaches to assist developers in automated refactoring. While JDeodorant (Tsantalis et al., 2008) and JMove (Sales et al., 2013) are examples of academic tools for automated refactoring, currently adopted Java IDEs, such as Eclipse and IntelliJ, provide built-in routines that automate refactoring operations. However, although some refactoring operations are conceptually simple, i.e. renaming, and can be easily automated, other operations, such as extract method, are more complex to automate, which might undermine the adoption of such tools by practitioners.

In this context, some researchers performed empirical studies to investigate the current practices of refactoring with a particular focus on the differences between manual and automated refactoring. Murphy-Hill et al. (2012) collected 7 different refactoring-related datasets to investigate how developers perform refactorings on their daily basis. The datasets include different groups of users of the Eclipse IDE, and spans from regular users who use Eclipse to develop their own projects to developers who specifically maintain Eclipse's refactoring functionalities. By monitoring these developers activities, they observe that close to 90% of refactoring operations are performed manually in spite of the existing automated support available within their IDE. Moreover, when considering the refactorings that were performed with the aid of a tool, 40% of those occurred in batches, i.e., more than one refactoring operation was performed in order to achieve the desired restructuring. In addition, the authors also observed that 90% of developers that use automated refactoring tools never alter the tools' default parameters. Finally, the developers who maintain Eclipse's refactoring functionalities employ less automated refactoring than regular Eclipse users.

In this seminal paper, Murphy-Hill et al. (2012) proposed the concept of *root canal* and *floss* refactoring, where the first indicates software changes that are performed with the sole purpose of refactoring the system and the latter indicates refactoring operations that are performed alongside other changes, such as feature improvement or bug fixing. Based on the collected data, they observed that *floss* refactorings are more often performed than their *root* counterpart. The concepts of *root* and *floss* refactoring heavily influenced the software refactoring research community (Candela et al., 2016; Cedrim et al., 2017; Tufano et al., 2017; Palomba et al., 2018) and this thesis in particular (see Chapter 6).

In a similar study, Negara et al. (2013) collected data from Eclipse users during their development sessions to investigate the differences between manual and automated refactoring. For this paper, the authors divided the participants into groups of professionals/experienced developers and students/novice developers. They observed that, on average, developers perform 11% more manual than automated refactorings, and that some refactoring operations, such as renaming, are predominantly performed manually. In addition, they noticed that experienced developers tend to perform manual refactorings more often than novice developers.

In an interesting observation, some developers claimed they perform manual refactorings because they were unaware of tool support within Eclipse. However, on average, most of the developers are aware of the tool support but still chooses to perform the refactoring operations manually. Moreover, similarly to the investigation by Murphy-Hill et al. (2012), the authors observed that automated refactorings tend to be larger and grouped in batches. Finally, the data indicate that about 30% of refactoring operations do not make to the version control system because they are shadowed by other refactorings, e.g., a method is renamed twice.

2.5.3 Developers' Intentions When Refactoring

When Fowler et al. (1999) first proposed their catalogue of refactoring operations, each refactoring had a motivation behind it, where the common goal was to fix/remove a design flaw or code smell. Extract method, for example, was proposed to deal with code duplication, and move methods were associated with the Feature Envy and Shotgun Surgery code smells. Thus, much of the early research in refactoring automation and recommendation were driven by code smells, code duplication, and design flaws (Tsantalis et al., 2008; Tsantalis and Chatzigeorgiou, 2011; Silva et al., 2014; Tairas and Gray, 2012).

In this context, the empirical studies discussed in the previous section have shown that developers tend to prefer *floss* refactorings, where the refactoring operations are mixed with other changes, such as feature implementation and bug fixing. Given this change in how researchers perceive the developers' needs and motivations towards refactoring, empirical studies were conducted to better understand the circumstances in which developers perform refactoring operations.

Silva et al. (2016) monitored a total of 124 Java projects from Github to investigate the developers' motivations behind refactoring operations. During the course of 61 days, the authors monitored the repository of each of these 124 projects by running the RefactoringMiner tool (Tsantalis et al., 2018) in every new commit to detect the presence of a refactoring operation. Given a new commit in which the tool detected a refactoring operation, one of the authors would manually validate the refactoring, and then send an email to the developer responsible for the commit asking for the motivation behind the refactoring and whether the refactoring was performed with the aid of an automated tool. After collecting all responses (41% response rate), the authors performed a thematic analysis to group the motivations expressed by each developer.

The extract method operation was the most common refactoring performed by developers, where the extraction of a method to make it reusable for a feature change or bug fixing was the most common motivation. Other popular motivations to extract a method were the introduction of an alternative signature, facilitation of future extension and decomposition to improve readability. Note that, when considering the most popular refactoring operation, only one out of the top four motivations involve pure structural improvement.

In addition, the developers claimed that 55% of the refactorings were performed manually instead of automatically. Finally, the most common reasons for performing a manual refactoring were the distrust in automated tools and the belief that the refactoring was simple enough so that automated support was not necessary.

In a related study, Palomba et al. (2017) investigated the relationship between refactoring operations and software changes performed under different intents. Based on a corpus of 12,992 software changes, the authors used an automated approach to classify each change under three possible intents: fault repairing, feature introduction, and general maintenance. Hence, after identifying the refactoring operations performed in each change, the authors performed an odds ratio analysis to understand the likelihood of a certain refactoring operation to be used in a software change under a certain intent.

As a result, they observed that 'Add Parameter', 'Move Field' and 'Replace Magic Number With Constant' refactorings are highly likely to be used (statistically significant) when developer fix a fault. When considering feature-related changes, developers are likely to perform 'Extract Method', 'Inline Method' and 'Rename Method' refactoring operations. Finally, when developers are performing maintenance activities, the most likely refactorings to be used are 'Consolidate Duplicated Code', 'Introduce Explaining Variable' and 'Rename Method'.

These studies are the first strides toward better understanding how, and most importantly, why developers perform refactoring in their software systems. These will serve as the ground rules for future researchers and tool builders when designing new approaches for automated refactoring assistance and recommendation.

2.6 Software Code Review

Software code review is the process of manual inspection and review of code performed by a developer that is not the author of the code being reviewed. Code review has been first proposed as Code Inspection by Fagan (1976). It consisted of a synchronous process, based on face to face review meetings between the author of the code and its reviewers. Based on more than 10 years of evaluation of code inspection techniques in real-world software development, Fagan (1976) concluded that code inspection is able to detect between 60% and 90% of bugs. Ackerman et al. (1989) also presented empirical evidence on the ability of the code review process to prevent the introduction of bugs. Moreover, Siy and Votta (2001) found out that code inspection improves code comprehensibility and facilitates changes.

In spite of the benefits of code inspection, Votta (1993) argued that code inspection is an unnecessary overhead for software development. Based on observations from code inspection meetings in industry, the author claims the need for meetings for every single inspection is too cumbersome, and that most of the time the goals and expectations of the meetings are not met. In fact, with the gain in popularity of agile methods for software development, the synchronous and rigid process of code inspection has been left out of practice.

Recently, code review has been brought to the software development lifecycle again due to its new lightweight and asynchronous format. In the seminal paper by Bacchelli and Bird (2013), the modern code review is described as a review that is informal, tool-based and that occurs regularly in the developers' workload. Modern code review is mostly characterised by its heavy use of automated tools to provide an asynchronous environment for review, with minimum overhead on the developers' workload. The general process of modern code review is depicted in Figure 2.5

As one can see, the process of modern code review is linked with a version control system. After a developer submits a code change, this change is not automatically merged into the repository, but it rather undertakes a reviewing process. The changed source code is first assessed by a continuous integration (CI) pipeline that might involve testing, building and other forms of quality control. After being approved by CI, the code is reviewed by one or more developers. The reviewer might point out problems in the change, request adjustments and provide other types of feedback. The author of the change will take the feedback into consideration and submit a new revision that will undergo the same process. This loop is repeated until the revision is finally merged into the repository or discarded.

This modern process of code review is widely adopted in the software development industry, with examples from Microsoft (Bacchelli and Bird, 2013), Google (Kennedy, 2006) and Facebook (Tsotsis, 2011). Open source tools for code review are also available, where the interested reader is referred to Gerrit (Pearce, 2006) and ReviewBoard (ReviewBoard, 2017).

Modern code review has been shown effective by recent empirical studies. Mantyla and Lassenius (2009) performed a study to investigate what kind of bugs are found during code review. The authors confirmed that code review is useful to find bugs that do not affect the functionality of the system. They pointed out that 75% of the bugs found by code review

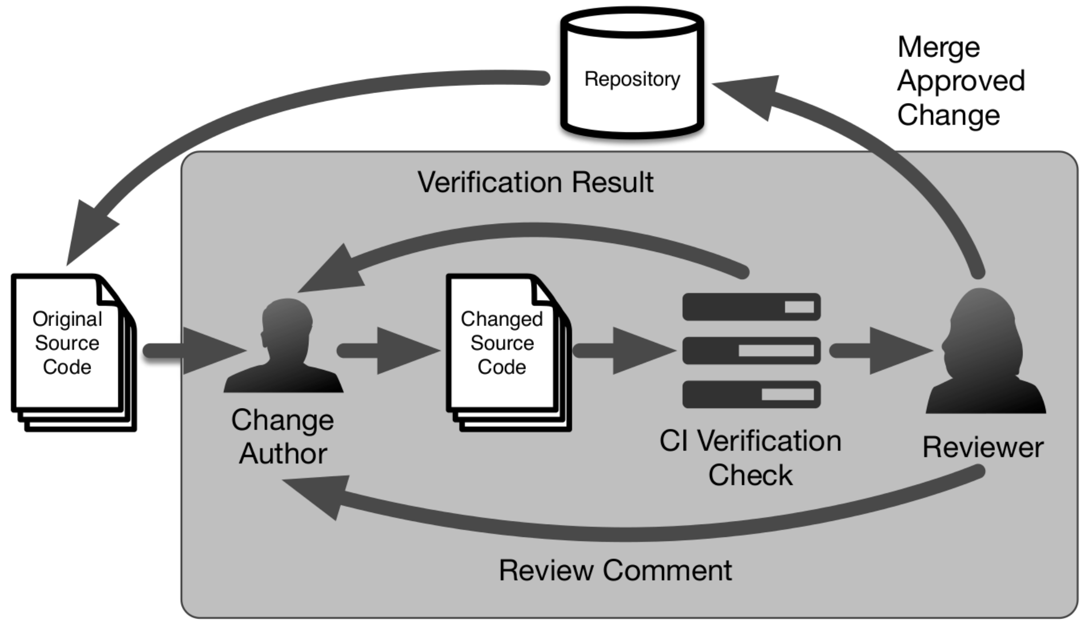


Figure 2.5: General process of modern code review

are related to maintainability, which would not be caught by testing, and therefore would be introduced in the codebase. In a similar study, Beller et al. (2014) confirmed that most bugs found during code review are related to maintenance issues and not functionality.

In the study performed by Bacchelli and Bird (2013) at Microsoft Research, the authors compared the developers' expectations and outcomes of the code review process. Although developers tend to believe the main goal of code review is to find problems, bug identification was ranked 4th in the most common outcomes of a review. The authors showed that code improvement and knowledge transfer are the most common results when a code change is reviewed. Rigby and Bird (2013) performed a similar study, where the authors also pointed out that improved knowledge transfer and social communications are the most common outcomes of modern code review.

3 An Empirical Study of Cohesion and Coupling: Balancing Optimisation and Disruption

In this thesis, we study how developers organise software systems in the context of the code level structure and modularisation, alongside the understanding of how the modularisation can be improved and how developers deal with restructurings. To do this, we first consider a popular approach for architectural restructuring, e.g, search-based software modularisation. After a careful analysis of the literature, we noticed that few papers evaluate the metrics they employ to guide the architectural improvement. Moreover, the incremental and iterative nature of software development is constantly overlooked by these approaches. Thus, most papers fail to consider the impact the proposed changes would accrue to the system's original structure.

Thus, this chapter tackles our thesis' objective number 1 (see Section 1.2), and reports a large-scale empirical study with real-world open source systems that validates commonly used metrics in search-based modularisation by investigating whether the modularisation proposed by the system's developers respect such metrics. Moreover, we take the state-of-the-art techniques in search-based software modularisation and evaluate how disruptive the suggested modifications would be to the system's structure, i.e., how much of the original system's structure would have to change to accommodate the proposed modularisation. In a side contribution, we propose a new search-based modularisation approach that finds compromises between structural improvement and disruption.

This study was originally published in the journal IEEE Transactions on Evolutionary Computation (TEVC), and this chapter presents an adaptation of the original paper. Section 3.1 presents the introduction of the original paper, while Section 3.2 contextualises the background and related work to this chapter in specific. Section 3.3 presents the software systems and dataset used in this chapter. While Section 3.4 presents our study's methodology and discusses the quantitative results, Section 3.8 depicts a qualitative analysis of some of our findings. In addition, Section 3.6 discusses the threats to the validity of the empirical study presented in this chapter. Finally, Section 3.7 presents the conclusions drawn from the empirical study as it was written in the original journal publication.

At the end of the chapter, Section 3.8 concludes this chapter by showing how the empirical study presented in this chapter contributes to the overall goal of this thesis.

3.1 Introduction

Software modularisation is almost as old as the concept of software engineering itself. The notions of cohesion and coupling were introduced in the 1970s (Yourdon and Constantine, 1979). Cohesion is the degree of relatedness enjoyed by code elements residing in the same abstract module, while coupling is the relatedness between modules. There is long-established evidence that software structure tends to degrade as the system evolves (Lehman, 1979; Lehman et al., 1997; Wermelinger et al., 2011). Therefore, one goal of software modularisation research is to find ways to improve modular structure, by increasing cohesion and reducing coupling.

search-based Software Engineering (SBSE) techniques have been widely studied and developed as one way to automate this structural modular improvement process, guided by fitness functions that capture structural cohesion, coupling, and combinations thereof. Structural cohesion/coupling is typically measured in terms of dependencies between elements. It is structural, rather than semantic, because it takes no account of the degree of semantic relations between elements, other than that which is captured through dependence measurements (Bieman and Ott, 1994).

Many different search techniques have been proposed and developed that automate the search for an improved modular structure. However, despite more than 30 publications on search-based modularisation, few studies (Hall et al., 2014; Candela et al., 2016) have performed an evaluation of the disruptive effects that automated modular improvement may cause on the original modular structure of the software systems under study. In the context of this chapter, we consider disruption to indicate the amount of change imposed in the system's original structure in order to adopt the solution proposed by an automated modularisation approach. A thorough study of the disruption caused by modular restructuring is needed because there is evidence that software engineers tend to resist structural and architectural improvement in favour of similarity and familiarity (Wermelinger et al., 2011). Therefore, high levels of disruption might undermine the industrial uptake of these techniques in the context of software restructuring.

Moreover, most of the surveyed publications on search-based modularisation consider only a single version of the systems under study, ignoring the systems' history of previous releases. A study involving a series of consecutive releases would be required in order to understand software engineers' decisions with respect to cohesion/coupling and the disruption that would have been caused by automated attempts to improve cohesion/coupling.

In this chapter, we provide the first study of search-based modularisation that considers both the opportunities for improving software structure and the consequent disruption that accrues as a result, over a series of subsequent releases of software systems. This is also the largest study in search-based modularisation: we study 233 releases of 10 different open source software systems, from which we extracted the modular structure data.

We start by investigating the validity of the quality metrics that previous work on search-based modularisation has used to improve software modularity. Our survey reveals that out of more than 30 papers that have previously studied this problem, many have used the Modularisation Quality (MQ) metric (Mancoridis et al., 1998; Mitchell and Mancoridis, 2006) to assess modularity quality. Therefore, we validate the use of this metric, investigating whether the existing modular structure implemented by developers respects MQ.

We complement our study of MQ by measuring the raw cohesion of each system. The raw cohesion is simply the number of dependencies that reside within a single module, and therefore do not cross any module boundary. The raw coupling is the obverse; the number of dependencies that cut across module boundaries. Given the proposed modular structure of a system, we can thus measure raw cohesion/coupling, simply by counting intra- and inter-dependencies between elements. Since raw coupling is the obverse of raw cohesion, we need only measure one of the two properties.

Traditional search-based modularisation does not use raw cohesion/coupling as a fitness function, because it would result in the algorithm moving all elements into one single module (with maximal cohesion and zero coupling). Such a 'god class' structure is undesirable (Brown et al., 1998), and various previous authors developed techniques to avoid this (Mitchell and Mancoridis, 2006; Harman et al., 2005; Praditwong et al., 2011). Although raw cohesion/coupling cannot be used to optimise the modular structure, it is a software engineering good practice that is easy to understand, and is accepted by developers (Bavota et al., 2013). As initially discussed by Simons and Smith (2016), humans tend to have a cognitive bias that makes it easier for developers to identify bad modularisations instead of good ones, making structural metrics of cohesion/coupling a valid proxy for the identification of not so good solutions. Hence, even though a modularisation with optimal levels of raw cohesion/coupling might not be generalised as the best possible from the developers' point of view, solutions with high levels of coupling and low levels of cohesion are widely accepted as poor modularisations. Hereinafter, when we refer to 'cohesion', we mean this simple 'raw cohesion' metric.

In order to provide an evidence-based assessment of the degree to which developers' implementations are cohesion-respecting and MQ-respecting, we introduce an approach to validation that is grounded in frequentist inferential statistics, widely used elsewhere in software engineering, and particularly recommended for SBSE (Arcuri and Briand, 2014; Harman et al., 2012). Using this statistical approach, we provide evidence that developers choose modular structures that are highly cohesion- and MQ-respecting. Furthermore, we show that although developers choose solutions in the local neighbourhood that have better cohesion and MQ values than at least 97.3% of the possible alternatives, in every release of every system, the developers' implementations are, nevertheless, suboptimal regarding both cohesion and MQ.

This motivates the study of the degree to which search-based modularisation could automatically 'improve' on the developer-implemented modular structure, according to cohesion and MQ. In order to answer this question, we empirically studied the widely-proposed hill climbing technique (Bunch) for finding improved modular structures (Mitchell and Mancoridis, 2006). The hill climbing approach is simple and fast, and has publicly available implementations, making it an obvious first choice for any developer seeking to use search-based techniques for modular improvement. After modifications to the original approach to cope with the large-scale real-world systems being studied, we found that, in most releases, automated modularisation does find modular structures with statistically significantly better cohesion and MQ values, and with large effect size.

Of course, restructuring may not be so straightforward in practice: if there was a dramatically improved modular structure available to the developers, then it seems reasonable to ask why software engineers have not adopted it. There are two potential explanations for this:

1. The developers are unaware of any better solutions; the search space is simply too large

and it defeats human-based search.

2. The developers are aware of at least one better solution, but *choose* not to implement any of the better solutions.

In all cases, we found that, even within the nearest neighbourhood to the developers' given implementation, there were always alternatives with improved cohesion/coupling. That is, improvement could be achieved simply by moving a single element from one module to another, in all of the 233 releases studied. This provides evidence that it is unlikely that developers were unaware of *any* better solution, so we turn our attention to the second possible explanation above.

If developers could easily find a better solution, even with a simple nearest neighbourhood search, why did they choose not to implement it? One possible explanation we chose to investigate, relates to the recent observation that developers are prepared to build up technical debt (Kruchten et al., 2012; Martini et al., 2015); resisting the temptation to restructure systems, and tolerating degradation in structure, in order to obtain fast delivery, retain familiarity of the existing structure and/or to preserve some other property of interest. Specifically, we investigate the degree of disruption that would be caused by moving to an improved modular structure that increases cohesion and reduces coupling. We measure disruption as the number of elements and modules that would need to be moved or merged, according to the MoJoFM metric (Wen and Tzerpos, 2004b). The results were striking: while a variation of the well-known Bunch automated modularisation approach can improve cohesion by 25% on average, these improvements result in 57% disruption.

This provides empirical evidence that developers are reluctant to disrupt the modular structure, even when this might lead to improved cohesion/coupling. Unfortunately, most of the previous work on search-based modularisation has ignored this disruptive effect, leaving open many questions that we seek to answer in the present chapter, such as how large the effect is and how often it occurs, whether it is correlated with the improvements achievable, and the degree to which it could be avoided while maintaining structural improvement.

We found that, although any modular improvement inherently inflicts some degree of disruption, in general, the disruption caused by the best improvement found by standard SBSE approaches, for every release of every system, is smaller than the average disruption. Furthermore, we found no evidence that cohesion improvement is correlated with disruption increase. This is a particularly attractive finding because it points to the possibility that a multiobjective search-based approach may be able to find balances and trade-offs between modular disruption and improvement. This more positive finding, thereby motivated our final set of experiments, in which we introduced, implemented and evaluated a novel multiobjective search-based modularisation technique.

Our new approach to automated modularisation seeks Pareto-optimal balances between disruption, as measured by MoJoFM, and improvement. On average, within the developer-determined 'acceptable' level of disruption for each system, which was calculated through longitudinal analysis between developers-implemented releases, our multiobjective approach was able to find solutions with an average of 22.52% and 55.75% improvements for cohesion and MQ, respectively.

The primary contributions of this chapter are the findings concerning the behaviour of both developers and existing SBSE techniques for automated modularisation (on 233 releases of 10 different software systems), the identification of disruption as an important problem for automated modularisation, and the novel multiobjective approach we introduce and evaluate to tackle this problem. Our empirical study and evaluation is the largest study of search-based modularisation hitherto reported in the literature, and its scientific findings have an actionable conclusion for researchers and practitioners; any and all approaches to modularisation (search-based or otherwise) need to take account of (and balance) the disruption they cause, against the improvement they offer.

3.2 Related Work and Background

We collected publications that use search-based techniques to improve the modular structure, where cohesion/coupling and combinations thereof are used to assess the quality of the modularisations. We cannot guarantee that we covered every paper, but we believe this survey presents a reasonable sample of the work performed by the search-based software modularisation community.

Table 3.1 summarises the 35 papers we collected and presents them sorted by year of publication. For each paper, we report whether it employs a Single Objective (SO) or MultiObjective (MO) optimisation approach, and what fitness functions are used to guide the search. We also report which search algorithms are used to perform the modularisation, and how many systems and releases were considered in each evaluation.

As one can see, the work on search-based software modularisation dates back to late 1990s (Mancoridis et al., 1998, 1999; Doval et al., 1999), with the proposal and first evaluations of the Bunch tool. The MQ metric was first proposed as Bunch's fitness function, and it is still the most used metric in search-based modularisation to date. In fact, suites of quality metrics more recently used for multiobjective modularisation (Praditwong et al., 2011; Bavota et al., 2012; Barros, 2012) include MQ as one of the metrics to be optimised.

3.2.1 Modular Structure Representation

In this chapter, the modular structure of each release under study is represented as a Module Dependency Graph (MDG), as defined in Section 2.3.

Since all the systems under study are implemented in Java (see Section 3.3), we are going to use the Java terminology to refer to the code elements and high-level modules; the elements are thus the classes and interfaces, while the modules are the packages. In this chapter, both classes and interfaces will be referred to simply as 'classes'. Dependencies occur by method call, field access, inheritance and interface implementation.

For each release of each system, the set of classes C is represented by $C = \{c_1, c_2, \ldots, c_N\}$, where N is the number of classes in the release. A dependency $d(c_x, c_y)$ indicates that class c_x depends on class c_y to correctly deliver its functionality. The set of all dependencies is represented by $D = \{d(c_x, c_y) \mid c_x, c_y \in C\}$. The set of packages P in a release is depicted by $P = \{p_1, p_2, \ldots, p_M\}$, where M is the number of packages in the release.

Table 3.1 Related work in search-based Software Modularisation sorted by year of publication

Paper	Year	Optimisation Approach	Fitness Function	Search Algorithm	Number of Different Systems Used	Number of Releases Studied
Mancoridis et al. (1998)	1998	SO	MQ	HC	5	5
Doval et al. (1999)	1999	SO	MQ	GA	1	1
Mancoridis et al. (1999)	1999	SO	MQ	HC	1	2
Mitchell et al. (2001)	2001	SO	MQ	HC	7	7
Harman et al. (2002)	2002	SO	Coh, Cop	HC, GA	7	7
Mitchell et al. (2002)	2002	SO	MQ	HC	5	5
Mahdavi et al. (2003)	2003	SO	MQ	HC	19	19
Mitchell et al. (2003)	2003	SO	MQ	HC	13	13
Harman et al. (2005)	2005	SO	MQ, EVM	HC	6	6
Seng et al. (2005)	2005	SO	Coh, Cop, Complexity, Cycles, Bottlenecks	GGA	1	1
Shokoufandeh et al. (2005)	2005	SO	MQ	HC and Spectral Algorithm	13	13
Mitchell et al. (2006)	2006	SO	MQ	HC	2	2
Mitchell et al. (2007)	2008	SO	MQ	HC	5	5
Abdeen et al. (2009)	2009	SO	Coh, Cop, Cycles	SA	4	4
Mamaghani et al. (2009)	2009	SO	MQ	Hybrid GA	5	5
Praditwong (2011)	2011	SO	MQ	GGA	17	17
Praditwong et al. (2011)	2011	MO	MCA, ECA	Two-Archive GA	17	17
Barros (2012)	2012	MO	MCA, ECA	NSGA-II	13	13
Bavota et al. (2012)	2012	SO and MO	MQ, MCA, ECA	GA, NSGA-II	2	2
Hall et al. (2012)	2012	SO	MQ	HC	5	5
Abdeen et al. (2013)	2013	MO	Coh, Cop, Modifications	NSGA-II	4	4
Kumari et al. (2013)	2013	MO	MCA, ECA	Hyper-heuristics	6	6
Ouni et al. (2013)	2013	MO	Fixed Bugs, Effort	NSGA-II	6	6
Hall et al. (2014)	2014	MO	MQ	HC	4	4
Barros et al. (2015)	2015	SO	MQ, EVM	HC	1	24
Jeet et al. (2015)	2015	SO	MQ	BHGA	6	6
Mkaouer et al. (2015)	2015	MO	Coh, Cop, MO, NCP, NP, SP, NCH, CHC	NSGA-III,IBEA, MOEA/D	5	5
Saeidi et al. (2015)	2015	SO and MO	MQ, CQ	HC, Two-Archive GA	10	10
Candela et al. (2016)	2016	MO	Structural and Contextual Coh/Cop	NSGA-II	100	100
Huang et al. (2016)	2016	SO and MO	MQ, MCA, ECA	MAEA-SMCPs, GGA, GNE	17	17
Huang et al. (2016)	2016	SO	MQ, MS	HC, GAs and MAEA	17	17
Jeet et al. (2016)	2016	SO	MQ	HC, five GA variations	7	7
Kumari et al. (2016)	2016	MO	MCA, ECA	Hyper-heuristics	12	12
Ouni et al. (2016)	2016	MO	Bugs, Coherence, Effort, Change History	NSGA-II	6	12
Paixao et al.	This Chapter	SO and MO	MQ, Disruption	HC, Two-Archive GA	10	233

3.2.2 Modular Structure Quality Metrics

The Modularisation Quality (MQ) metric was proposed by Mancoridis et al. (1998) to guide optimisation algorithms in the allocation of classes to highly cohesive and loosely coupled packages. In order to improve MQ's performance and quality assessment, the metric was re-formulated over the years (Mitchell et al., 2001; Mitchell and Mancoridis, 2002), and its most recent incarnation (Mitchell and Mancoridis, 2006) is adopted.

MQ consists of assigning scores to each package in the system, measuring the packages' individual trade-off between cohesion and coupling. The cohesion of a package p_i is represented by $coh(p_i)$, and it is computed as the number of dependencies between classes within package p_i. Accordingly, the coupling $cop(p_i)$ of package p_i is computed as the number of dependencies from classes within p_i to classes in other packages in the system. The MQ value of the overall system is computed as presented in Equations 3.1 and 3.2:

$$MQ = \sum_{i=1}^{P} MF(p_i) \tag{3.1}$$

$$\text{and, } MF(p_i) = \begin{cases} 0, & \text{if } coh(p_i) = 0 \\ \dfrac{coh(p_i)}{coh(p_i) + \frac{cop(p_i)}{2}}, & \text{if } coh(p_i) > 0 \end{cases} \tag{3.2}$$

The MQ is thus given by the sum of the Modularisation Factors (MF) of each package p_i in the system. MF(p_i) represents the trade-off between cohesion and coupling for package p_i. Since the dependencies involved in the measurement of the packages' coupling will be double counted during MQ computation, $cop(p_i)$ is divided by 2.

MQ is a function of the allocation of classes to packages; therefore, the MQ search space is composed of all possible allocations of classes to packages in the system. In this context, we define the k–neighbourhood as the subset of the MQ search space that can be achieved by performing k modifications to the original allocation of classes to packages that was implemented by the developers.

The raw cohesion/coupling of a system are measured by summing the cohesion/coupling of its packages. These are straightforward assessments of how many of the system's dependencies are contained in the same package and how many are cutting across the packages boundaries. Since raw cohesion/coupling are the obverses of each other, we need to measure only one of these properties, and for the rest of this chapter the raw cohesion, or simply 'cohesion' of the system is considered. Hence, the system's cohesion is computed as presented in Equation 3.3.

$$COH = \sum_{i=1}^{P} coh(p_i) \tag{3.3}$$

Apart from the selected metrics presented above, other measurements of structural cohesion and coupling have also been proposed (Ó Cinnéide et al., 2012) to account for different types of dependencies and different granularity levels. Recent studies reported quantitative and qualitative assessments of these metrics by investigating open source systems and interviewing developers (Candela et al., 2016; Simons et al., 2015). As previously mentioned, MQ is the most used quality metric in search-based modularisation (see Table 3.1), yet evidence that software systems respect this metric is scarce. Our empirical study performs an incremental assessment of the level of respect open source systems have to MQ; therefore, complementing previous literature and providing insights to the search-based modularisation community regarding its most used fitness function.

3.2.3 Modular Structure Disruption

The disruption caused by an automated modularisation technique can be measured as the number of changes developers need to perform in order to adopt the solution proposed by the search algorithm, where such disruption may be assessed at both source code and modular structure levels. A previous study by Hall et al. (2014) measured how many lines of code ought to be added/changed in order to apply solutions found by search-based modularisation approaches. Apart from showing that developers would have to change up to 10% of their code base to adopt a solution proposed by automated approaches, they also showed that the LOC to be changed strongly correlates with the modular structure disruption metric MoJoFM (Wen and Tzerpos, 2004b).

Mkaouer et al. (2015) used the number of refactoring operations as a measurement of system disruption to be minimised in a search-based many-objective approach for modularisation. However, operations at different granularity levels, such as *move method* and *move class*, have the same weight in the disruption computation, even though coarse-grained and fine-grained refactoring operations have a different impact in the system's modularity.

In the work by Ouni et al. (2013, 2016), the authors proposed a disruption assessment of refactoring operations based on the number of operations to be performed, where each operation is weighted by a complexity factor. In this formulation, the possible refactoring operations also include different granularity levels, e.g., *pull up method* and *extract class*, and the different weights are based on the authors' expertise.

As argued in a recent work by Candela et al. (2016), a modular level disruption metric, such as MoJoFM, better describes the 'mental model' developers have of their systems. Therefore, we draw inspiration from the study performed by Candela et al., and adopt a disruption measurement that is based on the widely used (Candela et al., 2016; Bavota et al., 2012; Mitchell and Mancoridis, 2001; Le et al., 2015) MoJoFM metric.

Given two different modularisations A and B of the same system, $MoJoFM(A, B)$ accounts for the proportional number of *Mo*ve and *Jo*in operations that are necessary to transform A in B, such as presented in Equation 3.4.

$$MoJoFM(A, B) = (1 - \frac{mno(A, B)}{max(mno(\forall A, B))}) \times 100\% \tag{3.4}$$

In this chapter, a *Mo*ve operation represents moving a class from its original package to another package in the system, while the *Jo*in operation represents the merge of two packages. The distance between A and B is the minimal number of operations that transform A in B, computed by $mno(A, B)$, and this value is normalised by the maximum distance between any possible modularisation partitioning of the system (denoted by $\forall A$) and B. MoJoFM is a non-trivial metric to compute, and for more technical details the reader is referred to the work by Wen and Tzerpos (2004b).

Finally, given the original developers' implementation A and a solution B suggested by an automated modularisation technique, we propose DisMoJo in Equation 3.5, a disruption metric based on MoJoFM that measures how much of the original implementation developers would need to change to adopt the modular optimised solution.

$$DisMoJo(A, B) = 100\% - MoJoFM(A, B) \tag{3.5}$$

Table 3.2 Open source systems used in the empirical evaluation of cohesion/coupling behaviour and optimisation. For each system, we report the number of releases and the median number of packages, classes and dependencies over releases. Finally, we report the median number of releases, packages, classes and dependencies for all systems.

Systems	Description	Releases	Packages	Classes	Dependencies
Ant	Tool to perform the ‘build’ of Java applications	30	25	576	2567
AssertJ	Library of assertions for Java	12	15	467	2095
Flume	Java logging API	10	17	255	849
Gson	Google’s converter of Java objects to JSON	15	6	153	724
JUnit	Java unit testing framework	20	23	196	734
Nutch	Java web crawler	13	18	272	1007
PDFBox	Java PDF manipulation library	31	48	496	3049
Pivot	Platform for building Installable Web Applications	12	13	150	568
Procyon	Java decompiler	47	36	895	6690
Proguard	Java code obfuscator	43	18	329	3513
All	-	17	18	300	1551

3.3 Software Systems Under Study

In this section we describe the systems we study in our empirical investigation of cohesion/coupling behaviour and optimisation, including the selection criteria we employed, the process for extracting the modular structure data, and a short description of each system.

The primary criteria for selecting software systems to study in our empirical investigation was the availability of at least 10 subsequent releases, so that we could evaluate more than one version of the systems and not only the latest one, like in most of the related work. We conjectured that 10 releases would be sufficient for our analysis. In particular, we did not require the system to have all of its releases available, as it is common for open source systems to avoid providing old versions of the system to users. Thus, for some systems, we have collected all the releases since the first one (e.g. Ant), and for others, we have collected all the releases available until the latest one available (e.g. JUnit).

As a result, we selected 10 open source Java systems, which are briefly described in Table 3.3. The number of releases of the systems under study varies from 10 to 47, with a median of 17 releases per system. Moreover, the median number of classes varies from 150 to 895 and the median number of dependencies between classes varies from 568 to 6690, indicating that these are non-trivial medium to large real-world software systems.

We employed a reverse engineering approach based on static analysis to obtain the modular structure of each release of each system. In order to do so, we used the pf-cda (Duchrow, 2018) tool to instrument the jar files of each release and subsequently extract the packages, classes and dependencies.

In order to facilitate replications of this study, we make available all 233 modularity datasets in our supporting web page (Paixao et al., 2017b). In addition, the web page also contains all results from this investigation, including further details elided for brevity in this chapter.

3.4 Empirical Study

This section describes and presents the results of the empirical study we carried out in this chapter to investigate cohesion/coupling behaviour and optimisation. Each of our research questions will be presented and answered, followed by a discussion of the findings.

3.4.1 RQ1: Is there any evidence that open source software systems respect structural measurements of cohesion and coupling?

By answering RQ1, we seek to investigate whether there is any evidence that the modular structure of existing software respects both raw cohesion and the MQ metric. We chose raw cohesion because it is a simple and intuitive measurement, and MQ because it is the most used metric in the automated software modularisation literature. Intuition suggests that developers do care about cohesion/coupling, and so we expect existing systems to exhibit *some* degree of 'respect' for these metrics.

We could survey developers with a questionnaire in order to discover a subjective self-assessment of the degree to which they care about these metrics, but such a study would be vulnerable to bias; developers may believe that they *ought* to care about these metrics, since cohesion/coupling have been recommended for many years (Yourdon and Constantine, 1979; Pressman, 2005; Sommerville, 2011). Such feelings may lead to an implicit or explicit bias that may influence the developers' self-assessment of the importance that they attach to measurements of cohesion/coupling. Moreover, any such assessment would be inherently subjective.

Therefore, although such results would undoubtedly be interesting, we choose to focus on a quantitative assessment of the degree to which the existing modular structure chosen by developers respects both the raw cohesion and the MQ metric.

In order to provide such a quantitative assessment of the degree of agreement with these metrics, we propose three different techniques, each of which produces a probabilistic assessment that can be used as the basis for an inferential statistical argument, concerning the likelihood of rejecting the Null Hypothesis (that the modular structure takes no account of the modularity quality metrics).

RQ1.1: How does the solution implemented by developers compare to a purely random allocation of classes to packages?

As a simple baseline, we start by considering a purely random allocation of classes to packages. Therefore, we are assuming the following Null Hypothesis H_0*: The modularity measurements of the releases of the studied open source software systems follow a purely random distribution*. That is, we assume, as a Null Hypothesis, that developers simply allocate classes to packages without any regard for the cohesion/coupling as captured by the chosen metrics. If this Null Hypothesis holds, then there is simply no evidence to suggest that developers care about cohesion or coupling. In such a situation, any attempt to optimise either raw cohesion or MQ, using search-based or other techniques, would be unlikely to be viewed as beneficial by developers.

Table 3.3 Likelihood of finding a modular structure with superior measurements of structural cohesion/coupling than that produced by the systems' developers, according to 3 search strategies. PRD simply searches for random allocations of classes to packages, while the other two techniques search the neighbourhood of the solution implemented by the systems' developers. kRNS randomly searches for solutions in the k–neighbourhood of the developers' solution, by moving k classes to randomly selected packages, while SNS systematically searches the k–neighbourhood for $k = 1$, by moving each class to one of each of the other packages. Results indicate the percentage of solutions found that improve the modular structure, as assessed using raw cohesion and MQ.

Systems	**Purely Random Distribution (PRD)**		**k–Random Neighbourhood Search (kRNS)**		**Systematic Neighbourhood Search (SNS)**	
	Cohesion	**MQ**	**Cohesion**	**MQ**	**Cohesion**	**MQ**
Ant	0.000000	0.000000	0.000590	0.000283	0.023026	0.052496
AssertJ	0.000000	0.000000	0.000344	0.000699	0.025063	0.067731
Flume	0.000000	0.000000	0.000413	0.001359	0.025661	0.072067
Gson	0.000000	0.000000	0.002233	0.013831	0.060578	0.142030
JUnit	0.000000	0.000000	0.000560	0.001616	0.029550	0.097797
Nutch	0.000000	0.000000	0.000125	0.000695	0.019316	0.047951
PDFBox	0.000000	0.000000	0.000587	0.001112	0.028475	0.086138
Pivot	0.000000	0.000000	0.000330	0.001147	0.023383	0.065127
Procyon	0.000000	0.000000	0.000042	0.000164	0.009813	0.049230
Proguard	0.000000	0.000000	0.004786	0.001588	0.083427	0.140490
All	0.000000	0.000000	0.001001	0.000866	0.032829	0.082105

In order to test H_0, a random distribution of class allocations was performed for each release of each system. A random allocation for a given release is performed by randomly allocating its set of classes to packages. The probability of a class to be allocated to a certain package is uniform. One million random class allocations were performed for each release of each system, thereby forming a sample of the space of all possible allocations of classes to packages.

Since we have 233 different releases of the 10 systems under investigation, this means that the experiment conducted to answer research question RQ1.1 involves the computation of 233 million randomly constructed modularisations. One (very obvious) advantage of our approach, from an inferential statistical point of view, is the ability to work with such a large sample. This large sample size enables us to produce precise assessments of the corresponding p-values.

The first two columns of Table 3.3 present the results of this analysis for each system under consideration, for raw cohesion and MQ, respectively. Raw coupling is simply the obverse of raw cohesion so, for brevity, we report only the values for raw cohesion. The entries in these columns indicate the percentage of random modularisations (over all releases) that achieve cohesion (or MQ) values that are equal to or greater than those achieved by the developers' implementation. As can be seen from these columns, not one of the 233 million randomly constructed modularisations produce a cohesion or MQ value equal to or greater than that achieved by the developers.

We can safely reject the Null Hypothesis H_0, and claim that raw cohesion and MQ values of open source software systems do not follow a random distribution. This result does not provide evidence that developers actually *care* about these metrics (it could simply be that they care about some other property that happens to correlate to significantly higher cohesion and MQ values). Nevertheless, these findings do strongly reject the claim that their allocation of classes to packages *fails to respect* modularity measurements; an obvious, yet important 'sanity check' result that has not hitherto been reported upon in the literature on search-based modularisation, despite the large body of previous work that use these metrics to guide modular optimisation.

Our Null Hypothesis was based on a purely random allocation of classes to packages, so the rejection of such a 'weak' Null Hypothesis can provide only a 'weak sanity check' on the intuition that the developers' modularisation structure respects modularity measurements. This last observation motivates the next two research questions, which seek to set a stronger baseline comparison, against which the developers' modularisation structure is compared.

RQ1.2: How does the developers' modularisation structure compare to randomly identified k–neighbour modularisations?

The k–Random Neighbourhood Search (kRNS) searches a randomly selected sample of solutions in the 'k–neighbourhood' of the solution implemented by the developers, as defined in Section 3.2.2. For this investigation, we use a value k equal to the number of classes in the systems. Therefore, kRNS proceeds by randomly selecting a subset of the classes in the system and randomly moving each of these classes to another package, to produce a single element of the sample. This process is repeated, using a freshly selected subset of classes on each occasion, to produce a sample of solutions from the k–neighbourhood. In our case, as with the previous experiment, we repeat this process 1 million times, for each release.

The third and fourth columns of Table 3.3 present the percentage of kRNS results that produce equal or higher cohesion and MQ values than those for the developers' modularisation. Consider Flume, for example. For all its releases, 0.000413% of k–neighbours found by kRNS had higher cohesion than the original solution, while 0.001359% of the k–neighbours had higher MQ. As can be seen from Table 3.3, over all the 10 systems, 0.004786% and 0.013831% are the highest number of cohesion-improved and MQ-improved modularisations found by the kRNS approach, respectively. Indeed, at the 0.01 α level, we would still reject the (strengthened) Null Hypothesis that *Developers simply pick an arbitrary re-allocation of classes to packages within the neighbourhood of the current solution, when producing a new version of the system.*

However, it can also be observed that, for every system, there does exist a member of the k–neighbourhood that enjoys a *higher* cohesion and/or *higher* MQ than that pertaining to the modularisation implemented by the system's developers. These results for RQ1.2, therefore, provide a deeper insight than it was possible from the purely random search used to answer RQ1.1. They show that, while modular structure tends to respect structural cohesion and coupling, developers, nevertheless, do not produce an optimal solution; a random search within the wider neighbourhood of the developers' solutions can improve the modularity in each and all the releases. Furthermore, one can observe that in the k–neighbourhood of all systems under study, the number of cohesion-improved solutions is different from the number of MQ-improved solutions. We will return to a deeper investigation of these observed differences later.

We now turn to a more systematic investigation of the neighbourhood. Clearly, for a systematic investigation of all k–neighbours, the computational cost rises exponentially (in k), and, in the limit, as k tends to the number of classes in the system, the systematic investigation tends to an exhaustive enumeration of all possible modularisations of the systems under investigation. This is clearly infeasible (Mitchell, 2002). Indeed, avoiding such an exponential explosion was our motivation for sampling from the overall k–neighbourhood for RQ1.2. However, it is computationally feasible to consider the nearest of all neighbourhoods; the k–neighbourhood for $k = 1$, and this allows us to answer an interesting research question:

RQ1.3: What portion of modularisation allocations within the nearest possible neighbourhood ($k = 1$) would yield an improvement in modularity?

The systematic enumeration of the 1–neighbourhood is interesting because this is the set of neighbouring modularisations that can be achieved by moving only a single class to another package in the system. As such, it is the single simplest (and least disruptive) possible modification to the modularisation structure chosen by the developers. In order to answer this research question, we took each class and moved it to each of the other packages which the class was not originally assigned by the developers. This yields $(M - 1) \times N$ 1–neighbours (or 'nearest neighbours'), for a system consisting of M packages and N classes, thereby systematically covering the entire 1-neighbourhood. The results of this analysis are presented in the final columns of Table 3.3.

As can be seen, for each system investigated, there are a nontrivial number of such single moves that can improve both the cohesion and the MQ score. Nevertheless, the solutions chosen by the developers are better than at least 96.7% and 91.7% of the whole 1–neighbourhood, for cohesion and MQ, respectively. This provides evidence for a strong developer preference for structures that respect modularity metrics. Moreover, as observed in RQ1.2, the number of cohesion-improved solutions is different from the number of MQ-improved solutions. In fact, all systems presented more MQ-improved solutions than cohesion-improved solutions in the 1–neighbourhood.

Overall, as an answer to RQ1, we conclude that there is strong evidence to suggest that the developers' allocation of classes to packages does respect structural cohesion/coupling, as assessed by the metrics of raw cohesion and MQ. Furthermore, there is equally strong evidence that the developers' allocation of classes could be improved, possibly with relatively little disruption to the system's modular structure, since there do always exist near neighbour modularisations that enjoy better modular structure. This is interesting and important for work on automated software modularisation since these metrics (MQ in particular) are widely used fitness functions to guide such work on automated modularisation.

As observed in RQ1.2 and RQ1.3, when searching both the k–neighbourhood and the 1–neighbourhood of the systems under study, raw cohesion and MQ sometimes do not agree in assessing the modular structure of different solutions. We define 'agreeing solutions' to be those modularisations that improve on the developers' implementation according to both cohesion and MQ, while 'disagreeing solutions' are those that have either higher cohesion or higher MQ, but not both.

Interestingly, and importantly for search-based modularisation research, we observe that, on average, 83.04% of the neighbourhood solutions that present an improvement on the original implementation in either cohesion or MQ are 'disagreeing' and only 16.96% are 'agreeing'. Moreover, for the disagreeing solutions, 67.61% present higher MQ than the original implementation but lower cohesion, and 32.39% present higher cohesion than the original implementation but lower MQ. These findings replicate previous observations (Ó Cinnéide et al., 2012). In addition, we have presented evidence that developers' solutions more closely respect raw cohesion than MQ. Since MQ is a popular metric for search-based modularisation, it is interesting and important for the community to understand how this metric is related to raw cohesion, which is a more basic and intuitive assessment of modular structure. This observation motivates our next research question.

3.4.2 RQ2: What is the relationship between raw cohesion and the MQ metric?

We performed three different analyses to investigate the relationship between MQ and raw cohesion. Each of these analyses employ a different technique to search for better modularisations.

RQ2.1: What is the relationship between raw cohesion and MQ for the solutions identified in RQ1?

RQ1 performed neighbourhood search in the developers' implemented solutions to find better allocations of classes to packages. For RQ2.1, all neighbour solutions that improved on the original developers' implementation in at least one of the metrics (raw cohesion/coupling and/or MQ) were considered for analysis, which, on average, represents 0.35% of the solutions found by the kRNS and SNS in RQ1.

The first three columns of Table 3.4 present the average differences and standard deviation in cohesion, coupling, and MQ for the neighbourhood search solutions, respectively. Consider the Gson system, for example. The neighbourhood solutions offer an average difference in cohesion, coupling, and MQ of -5.269%, 5.269% and 2.934%, respectively. These values indicate that within the set of Gson neighbourhood solutions that improve on the original implementation in at least one of the metrics, the average differences in cohesion, coupling and MQ are -5.269%, 5.269% and 2.934%, respectively.

One should notice that, as mentioned before, cohesion and coupling differences are the obverses of each other. Since we are considering cohesion to be the number of dependencies within packages and coupling to be the number of dependencies between packages, in the case of a certain dependency being moved from between packages to inside a package, cohesion will increase and coupling will decrease. Similarly, a dependency that is moved from inside a package to between packages is going to decrease cohesion and increase coupling. Therefore, for brevity purposes, only cohesion values will be reported and discussed in the rest of the chapter.

We use correlation analysis to investigate more precisely the relationship between cohesion and MQ. For each release of each system, the non-parametric Kendall-τ correlation test was applied for the cohesion and MQ values of the neighbourhood solutions. After a preliminary analysis of the results, we noticed that many runs of the neighbourhood algorithms (Bunch and package-constrained) resulted in solutions with the same values of cohesion and MQ, which generated many ties within the data points for analysis. Hence, we chose the Kendall-τ correlation as it is specifically tailored for this scenario (Cohen, 1988).

Table 3.4 Cohesion, Coupling and MQ results with standard deviation for Neighbourhood, Bunch and Package-constrained searches for improved modularisations. Cohesion, Coupling and MQ entries denote the average difference between the optimised modularisation in comparison to the original developers' implementation. In addition, we report the Kendall-τ correlation coefficients between Cohesion and MQ results for each system.

Systems	Neighbourhood Search				Bunch				Package-constrained HC			
	Cohesion	**Coupling**	**MQ**	**K-τ**	**Cohesion**	**Coupling**	**MQ**	**K-τ**	**Cohesion**	**Coupling**	**MQ**	**K-τ**
Ant	0.423% ± 1.116%	-0.423% ± 1.116%	-1.907% ± 3.644%	-0.18*	-45.919% ± 2.279%	45.919% ± 2.279%	376.611% ± 0.378%	0.56*	30.063% ± 5.348%	-30.063% ± 5.348%	40.598% ± 1.078%	0.44*
AssertJ	-0.174% ± 1.066%	0.174% ± 1.066%	0.102% ± 1.130%	-0.27	-62.888% ± 1.772%	62.888% ± 1.772%	522.197% ± 0.293%	0.41*	0.714% ± 4.217%	-0.714% ± 4.217%	61.167% ± 1.816%	0.46*
Flume	-1.092% ± 2.273%	1.092% ± 2.273%	0.857% ± 0.594%	-0.22	-35.977% ± 2.100%	35.977% ± 2.100%	336.829 ± 0.594%	0.57*	19.379% ± 4.030%	-19.379% ± 4.030%	51.026% ± 1.232%	0.55
Gson	-5.269% ± 7.090%	5.269% ± 7.090%	2.934% ± 0.569%	0.08*	-55.559% ± 3.133%	55.559% ± 3.133%	584.924% ± 0.940%	0.45*	19.832% ± 5.988%	-19.832% ± 5.988%	97.978% ± 4.089%	0.64
JUnit	-0.449% ± 1.797%	0.449% ± 1.797%	0.292% ± 1.446%	-0.14*	-28.240% ± 2.900%	28.240% ± 2.900%	226.341% ± 0.998%	-0.37*	30.157% ± 4.850%	-30.157% ± 4.850%	66.530% ± 1.608%	0.48*
Nutch	-1.252% ± 1.977%	1.252% ± 1.977%	1.018% ± 0.414%	-0.27	-42.993% ± 2.015%	42.993% ± 2.015%	321.843% ± 0.420%	0.50*	17.564% ± 5.603%	-17.564% ± 5.603%	56.962% ± 0.746%	0.44*
PDFBox	-0.377% ± 1.398%	0.377% ± 1.398%	0.119% ± 1.537%	-0.18	-33.100% ± 3.595%	33.100% ± 3.595%	180.531% ± 0.350%	0.43*	31.387% ± 5.532%	-31.387% ± 5.532%	79.773% ± 0.979%	0.43*
Pivot	-1.258% ± 2.273%	1.258% ± 2.273%	0.519% ± 0.549%	-0.20	-43.776% ± 1.680%	43.776% ± 1.680%	266.257% ± 0.508%	0.63	3.927% ± 4.102%	-3.927% ± 4.102%	44.875% ± 1.155%	0.51*
Procyon	-0.064% ± 0.289%	0.064% ± 0.289%	0.102% ± 0.054%	-0.20	-71.726% ± 1.194%	71.726% ± 1.194%	408.408% ± 0.145%	0.57	-4.799% ± 4.312%	4.799% ± 4.312%	56.538% ± 0.634%	0.44*
Proguard	1.035% ± 1.435%	-1.035% ± 1.435%	-2.494% ± 3.805%	-0.10*	-42.355% ± 5.383%	42.355% ± 5.383%	295.561% ± 0.357%	0.48*	102.282% ± 10.911%	-102.282% ± 10.911%	86.169% ± 2.606%	0.46*
All	-0.750% ± 2.071%	0.750% ± 2.071%	0.154% ± 1.374%	-0.19*	-46.253% ± 2.625	46.253% ± 2.625	351.950% ± 0.498%	0.49*	25.050% ± 5.489%	-25.050% ± 5.489%	64.161% ± 1.594%	0.46*

The fourth column of Table 3.4 presents the correlation coefficient of each system, which is computed as the median coefficient of all releases of each system. An asterisk (*) decorates the coefficient entry when not all releases exhibit a significant coefficient at the 0.01 α level. Most coefficients range from -0.2 to 0.2, which suggests little or no correlation between cohesion and MQ for the neighbourhood solutions.

However, we must be careful to not over-generalize this observation, because only simple local search procedures were employed to find the solutions that were considered in the analysis, and the search space covered by the neighbourhood solutions is small. This motivated our next research question, in which we apply more sophisticated search-based approaches for software modularisation.

RQ2.2: What is the relationship between raw cohesion and MQ for solutions found by widely used search-based cohesion/coupling optimisation approaches?

For this analysis, we use the Bunch tool (Mitchell and Mancoridis, 2006). Bunch is a tool for search-based modularisation that implements a simple hill climbing approach. There are other more sophisticated techniques for search-based modularisation, that may produce superior results (Praditwong et al., 2011) in terms of cohesion, coupling, and the MQ metric, but at far greater computational cost. We wish to investigate whether developers can use simple and fast search-based modularisation techniques to quickly produce alternative solutions that significantly improve on the developers' given modularisations, according to MQ.

We applied the Bunch optimisation tool to all releases. Since Bunch's hill climbing algorithm is a randomized search algorithm, we performed 30 executions of Bunch for each release. The 30 resulting cohesion, coupling and MQ values found by Bunch for each release were compared with the developer's implementation, and the results are presented in the fifth, sixth and seventh columns of Table 3.4, alongside the respective standard deviation.

As one can see, the Bunch tool is able to find modularisations with remarkable MQ improvement (of more than 500% for some systems). However, all these MQ-optimised solutions have lower cohesion values than the developer's original implementation. Such a surprising result can be explained by the design of the MQ metric. As one can see in the MQ definition in Section 3.2.2, the MQ score is composed of the sum of the scores of each package in the modularisation; so, solutions with more packages tend to have higher MQ values. In fact, the solutions found by Bunch have, on average, 493.11% more packages than the developers' implementation. As a result, fewer classes are allocated to each package, thereby creating several dependencies that cut across package boundaries. We will refer to this phenomena as the MQ's 'inflation effect'.

The Kendall-τ correlation test was also applied to measure the correlation between raw cohesion and MQ of the Bunch solutions. Apart from JUnit, all systems have a moderate positive correlation between cohesion and MQ, which is a surprising result given that all Bunch solutions had worse cohesion than the original implementation. First of all, one needs to keep in mind that this correlation was computed using the 30 Bunch solutions of each release of each system. A positive correlation, in this case, indicates that in spite of the fact that all cohesion values of Bunch solutions are worse than the developers' implementation, the solutions with higher MQ tend to have a higher cohesion too.

As an example, the scatter plot in Figure 3.1 presents the cohesion and MQ differences of the 30 solutions found by Bunch for Pivot 2.0.2 in comparison to the original implementation. As one can see, all the 30 solutions have higher MQ than the developers' implementation, yet lower cohesion. However, the solutions with higher MQ tend to have a higher cohesion too, which elucidates the positive correlations between cohesion and MQ reported in Table 3.4.

After an analysis of the 30 Bunch solutions, we noticed that these solutions have a similar number of packages, where 16 (out of 30) solutions have 58 packages, 10 solutions have 57 packages and 4 solutions have 59 packages. This suggests that for modularisations with similar numbers of packages, higher MQ values usually denote higher cohesion. These observations motivate our next research question, where we introduce and evaluate a package-constrained approach for search-based software modularisation as an attempt to improve the modular structure of software systems as assessed by *both* cohesion and MQ.

RQ2.3: What is the relationship between raw cohesion and MQ for solutions found by a package-constrained search for improved cohesion/coupling?

The MQ metric was originally designed to optimise the cohesion/coupling of software systems from scratch, without any previous information on the modular structure other than the dependencies between elements. However, when Bunch is applied to large-scale real-world software systems, the 'inflation effect' induced by MQ may be undesirable. Because of this effect, new packages are created and existing classes are moved to these new packages, causing a (large) decrease in the system's cohesion.

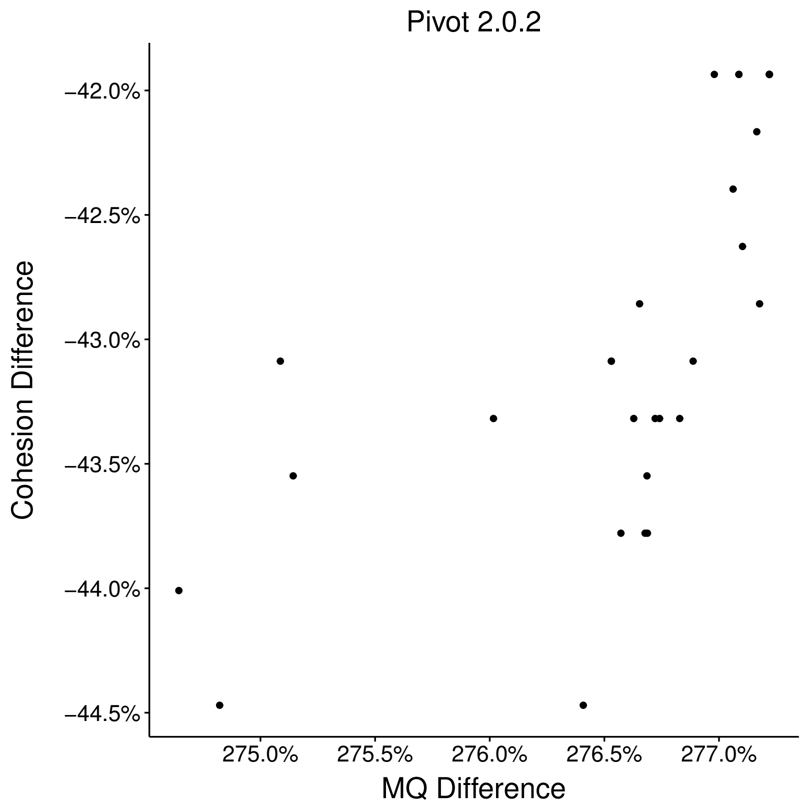


Figure 3.1: Cohesion and MQ differences for 30 modularisations found by Bunch for Pivot 2.0.2 when compared to the original developers' implementation

We performed a longitudinal analysis of the allocation of classes to packages throughout releases as implemented by the developers themselves. We found no release (out of 233) where a new package was created and only existing classes were moved to the new package. Therefore, apart from decreasing the cohesion of the system, a Bunch modularisation might also be unrealistic because developers rarely create new packages to accommodate existing classes. This observation adds evidence to recent claims (Hall et al., 2014; Candela et al., 2016) against 'big bang' modularisation approaches (i.e., a complete re-allocation of the system's classes into packages), where recent studies have used the original modular structure implemented by developers as a guide to find more suitable packages for certain classes (Abdeen et al., 2009; Bavota et al., 2014).

Thus, in this research question, we introduce a package-constrained version of search-based modularisation that maximises MQ and constrains the search algorithm to search only for modularisations with the same number of packages of the original developers' implementation. This way we avoid the creation of new packages, so that classes are only moved to packages that developers are already familiar with. Moreover, as suggested in RQ2.2, higher MQ values may lead to higher cohesion values for the same number of packages. Therefore, by maintaining the same number of packages as the original implementation, we might be able to optimise MQ and improve the overall cohesion of the system.

Hence, we re-implemented the hill climbing search approach of the Bunch tool including the number of packages as a constraint to the search. We executed the approach 30 times for each release of each system, and the average cohesion, coupling, MQ and standard deviations achieved by the package-constrained search are reported in the ninth, tenth and eleventh columns of Table 3.4, respectively.

Apart from the Procyon system, all package-constrained solutions yield improvements in *both* cohesion and MQ. On average, the cohesion of the systems under study was improved by 25.05%, and the biggest cohesion improvement was in Proguard with 102.28%. However, similar results were not achieved in some of the systems, such as AssertJ and Pivot that had small cohesion improvements, and Procyon that had a worse average cohesion than the original implementation. The results for these three systems indicate that in some cases, even in a package-constrained setting, MQ optimisation does not lead to better modularity, as assessed by raw cohesion. It might be possible that these systems already have a good cohesion, and cannot be further optimised.

The Kendall-τ correlation coefficients between cohesion and MQ for the package-constrained search are reported in the last column of Table 3.4. The moderate positive coefficients that can be seen for the package-constrained search resemble the coefficients computed for the Bunch solutions. These results reinforce the observation that MQ can indeed guide the search towards solutions with better cohesion when the search is package-constrained. This is an important finding for the search-based modularisation community.

As an answer to RQ2, we showed that raw cohesion and MQ do not commonly agree in assessing the modularity of software systems. We noticed that this is mainly due to the ‘inflation effect’ of MQ, where Bunch creates an average of 493.11% new packages in the system, which decreases the cohesion when compared to the original developers’ implementation. However, we observed that in solutions with a similar number of packages, MQ and cohesion have a moderate positive correlation, which mainly led us to introduce a new package-constrained search as an attempt to mitigate MQ’s ‘inflation effect’. In general, package-constrained automated modularisation was able to improve the cohesion of the systems under study by 25.05% without creating new packages.

Considering the results presented in RQ1 and RQ2, we showed that developers have some degree of respect for structural measurements of cohesion and coupling as the original solutions are better than the ones found by random and neighbourhood search. However, optimal values of cohesion and coupling might not be pursued since developers’ solutions are worse than the ones found by the hill climbing search. This observation endorses a recent study (Candela et al., 2016) that compared developers’ modularisations of open source systems with alternatives found by multiobjective search for cohesion/coupling improvement. Even though the empirical studies presented in this and in the related work by Candela et al. (2016) use different quality metrics, different search procedures and different software systems, they complement each other by presenting evidence that developers do respect structural measurements of cohesion and coupling, but optimisation of these properties is not sought.

3.4.3 RQ3: What is the disruption caused by search-based approaches for optimising software modularisation?

The previous section showed that search-based algorithms can be used to optimise the trade-off between cohesion and coupling in open source software systems. In fact, candidate solutions in the package-constrained search usually present improved modular structure, as measured by both cohesion and MQ metrics. This raises the obvious question: if systems can be optimised for modularity, and there is evidence that systems respect structural measurements, then why do developers implement solutions with a sub-optimal modular structure?

One answer to this question lies in the potential size and complexity of the search space. Humans have been shown, repeatedly, to be sub-optimal, in their ability to find solutions to SBSE problems such as this (Petke et al., 2014; de Souza et al., 2010). However, it is also important to explore another possibility: perhaps the improvement in modular structure achieved using SBSE comes with a price of significant disruption to the existing modularity. There is evidence in the literature (Wermelinger et al., 2011) that developers are reluctant to change the structure of systems, choosing instead, to retain the familiar structure rather than move to an improved version. Therefore, we turn our attention to assessing the degree of disruption that would result from an improvement performed by the SBSE approaches to automated software modularisation presented in RQ2. For this analysis, we use the DisMoJo metric, which is formally defined in Section 3.5.

RQ3.1: What is the disruption caused by widely used search-based tools for automated software modularisation?

In this research question, we want to assess how much disruption developers would have to endure when using a widely used tool for modularity optimisation. For this analysis, the solutions found by the Bunch tool in RQ2.2 will be considered. Each of the 30 solutions found by Bunch for each release of each system is compared to the original developers' implementation. The average disruption caused by Bunch, as assessed by DisMoJo, for each system under study is presented in the first column of Table 3.5.

Considering all systems, the average disruption that developers would need to endure in order to optimise the modular structure using Bunch is 80.39%. This observation provides evidence that even though existing SBSE techniques can improve modular structure, the high disruption caused to the original system might inhibit wider industrial uptake of search-based modularisation.

After an inspection of all 30 Bunch solutions of each release of each system, we collected the solutions with higher cohesion and MQ, and reported the average disruption caused by these best solutions over all releases. The disruption caused by the best cohesion and best MQ solutions found by Bunch are presented on the second and third columns of Table 3.5, respectively.

As one can see, the disruption caused by the best cohesion and MQ solutions is slightly smaller than the average disruption, where the biggest difference in disruption ranges around 2%. This indicates that solutions with similar values of cohesion and MQ do present different values of disruption, which encourages the investigation of the possible trade-off between

Table 3.5 Disruption to the modular structure caused by Bunch and Package-constrained search-based approaches for modularisation improvement. We report the mean disruption caused by the 30 executions of each search approach. In addition, we report the disruption caused by the best solutions (out of the 30), as assessed by Cohesion and MQ. Each entry in the table is an average over all releases of the system.

Systems	Bunch			Package-constrained		
	Mean	Best - Cohesion	Best - MQ	Mean	Best - Cohesion	Best - MQ
Ant	82.70%	80.63%	81.70%	57.86%	53.50%	55.73%
AssertJ	90.11%	89.66%	90.11%	59.02%	55.51%	56.54%
Flume	79.90%	79.09%	79.19%	62.20%	57.57%	58.16%
Gson	88.49%	85.22%	87.78%	56.04%	48.72%	51.16%
JUnit	69.87%	68.24%	69.23%	52.11%	49.53%	50.12%
Nutch	77.22%	75.92%	76.45%	61.91%	60.30%	60.15%
PDFBox	66.78%	64.18%	65.96%	53.12%	51.49%	51.74%
Pivot	79.26%	78.39%	78.32%	60.47%	56.36%	56.36%
Procyon	85.98%	84.39%	85.23%	56.94%	53.94%	54.88%
Proguard	83.66%	82.02%	82.74%	58.59%	56.44%	58.24%
All	80.39%	78.77%	79.67%	57.82%	54.33%	55.30%

quality improvement and disruption to the original structure.

RQ3.2: What is the disruption caused by the package-constrained search-based approach for automated software modularisation?

The package-constrained search approach for software modularisation was introduced in RQ2.3 as an alternative to mitigate the 'inflation effect' of the Bunch tool. The average disruption caused by the package-constrained search is presented in the fourth column of Table 3.5.

As expected, the disruption caused by the package-constrained search is smaller (57.82%), but it still denotes a high number of modifications to the original system in order to optimise the modular structure.

The average disruption caused by the best cohesion and MQ solutions for the package-constrained search are presented in the last two columns of Table 3.5, respectively. Similarly to the Bunch results, the disruption of the solutions with best modular structure is only slightly smaller than the average disruption caused by the 30 executions for each release.

In general, the disruption caused by the package-constrained optimisation approach is smaller than the disruption caused by the Bunch tool. An interesting observation from these analyses was that solutions with the best modular structure, as assessed by both cohesion and MQ, presented different disruption than the average, which encourages the study of such trade-off.

As an answer to RQ3, the disruption caused by search-based approaches to automated modularisation is high. The results found in this chapter complement a recent disruption analysis performed by Candela et al. (2016), where despite using different optimisation algorithms, different cohesion/coupling metrics and different software systems, both studies showed that search-based modularisation is highly disruptive. We conjecture that such disruption inhibits industrial uptake of these techniques.

3.4.4 RQ4: Can multiobjective search find allocations of classes to packages with a good trade-off between modularity improvement and disruption of the original modular structure?

Summarising the findings of RQ1-3: open source software systems respect structural measurements of cohesion and coupling (RQ1), but although search-based techniques can substantially improve the systems' modular structure (RQ2), these techniques tend to dramatically disrupt the original developers' implementations (RQ3).

Motivated by these findings, we introduce a multiobjective evolutionary search-based approach to find candidate modularisations with a good trade-off between modular improvement and disruption. Our intuition is that since the systems under study exhibit considerable respect for structural measurements of cohesion and coupling, developers might be willing to improve their systems' modular structure when the changes required for improvement lie within an acceptable range.

In order to carry out this analysis, we propose two different multiobjective experiments, each of which uses different search strategies; therefore, providing different insights on how multiobjective search can be used to improve software modularity while taking disruption into account.

For all multiobjective experiments, we use the Two-Archive Genetic Algorithm (Praditwong and Yao, 2006), which was demonstrated to perform well in a previous multiobjective investigation of automated software modularisation (Praditwong et al., 2011). The Two-Archive GA settings are mostly based on the work by Praditwong et al. (2011), and are the same for all experiments. The population size is set to N, where N is the number of classes in the system. Single point crossover is employed with a 0.8 probability when $N < 100$, and 1.0 probability otherwise. Swap mutation is performed with a probability of $0.004 \log_2^N$. Parents are selected by tournament, with a tournament size of 2. In addition, the probability of selecting parents from the convergence archive is 0.5, and the size of the archives is limited to 100 individuals. Finally, the number of generations is set to $50N$.

RQ4.1: What is the trade-off between modularity improvement and disruption for the package-free search?

The first multiobjective experiment is concerned with the widely used (Mitchell and Mancoridis, 2006; Praditwong et al., 2011) optimisation approach to improve software modularity where the search algorithm has no constraints on the number of packages it can create. We call this search strategy 'package-free'. In order to identify the trade-off between modularity improvement and disruption, the search algorithm attempts to maximise MQ and minimise DisMoJo. In addition, we measure the raw cohesion of the solutions found by the multiobjective search.

In RQ2.2 we used the Bunch tool to find MQ-optimised solutions for each release of each system under study; therefore, the solutions found by Bunch can be used as starting points (seeds) for the multiobjective algorithm in its search for solutions with high MQ value. Similarly, the original developers' implementation of each release is also used to seed the Two-Archive GA.

Figure 3.2 presents some of the Pareto fronts found for the package-free multiobjective execution. We selected one release as a representative of each system to be discussed in this chapter. However, we make all results available on the original paper's complementary web page (Paixao et al., 2017b). As one can see, the results for the different systems are considerably similar, where all releases present a clear and almost constant trade-off between MQ improvement and DisMoJo, which is an expected behaviour because MQ improvement is achieved by adding new packages; therefore, leading to large-scale disruption.

RQ2 showed that an improvement in MQ does not necessarily indicate an improvement in the raw cohesion of the system; therefore, we also measured the raw cohesion of all different modularisations found by the multiobjective search that targets MQ improvement. When considering all the package-free MQ-optimised modularisations in the Pareto fronts, most of them have a cohesion value that is *worse* than the original developers' implementation. These results add evidence to the observation in RQ2, that MQ-optimised solutions may decrease the cohesion of the original system. In fact, when considering the Pareto fronts computed for Nutch, PDFBox and Proguard, for example, *all* the modularisations are worse than the original system in terms of raw cohesion.

RQ4.2: What is the trade-off between modularity improvement and disruption for the package-constrained search?

This second multiobjective experiment is concerned with the automated software modularisation approach proposed in RQ2.3, where the search algorithm is package-constrained. Similarly to RQ4.1, the multiobjective search tries to maximise MQ and minimise DisMoJo. However, the search algorithm is constrained to the same number of packages as those in the original developers' implementation.

For this research question, the Two-Archive GA is seeded with the original system (as in RQ4.1) and the MQ-optimised solutions found by our package-constrained implementation of the hill climbing algorithm used by the Bunch tool (see RQ2.3). Figure 3.3 presents the Pareto fronts found by the package-constrained multiobjective search.

Similarly to RQ4.1, there is a clear trade-off between MQ and DisMoJo; however, the Pareto front structure is different: we observe a larger number of gaps and 'knee points' in the package-constrained Pareto fronts than in the package-free ones.

The cohesion improvements achieved by all modularisations in the package-constrained Pareto fronts were also measured. In most of the systems, the number of MQ-optimised modularisations with better cohesion than the original implementation is noticeably bigger than in RQ4.1. Moreover, for almost all the systems, it is possible to find modularisations with a considerable improvement in cohesion and yet a relatively small disruption. This is a positive outcome. Although modularisation approaches may be too disruptive, multiobjective search is able to find solutions with useful compromises between modular improvement and disruption.

As one can notice in the package-constrained Pareto fronts in Figure 3.3, sometimes the modularisation found by the hill climbing package-constrained search is not part of the Pareto front. Considering Ant, Flume and JUnit, for example, the hill climbing modularisation has higher disruption and lower MQ than other solutions in the Pareto front. Differently, in the package-free Pareto fronts in Figure 3.2, the Bunch solution is always the one with highest MQ on all Pareto fronts.

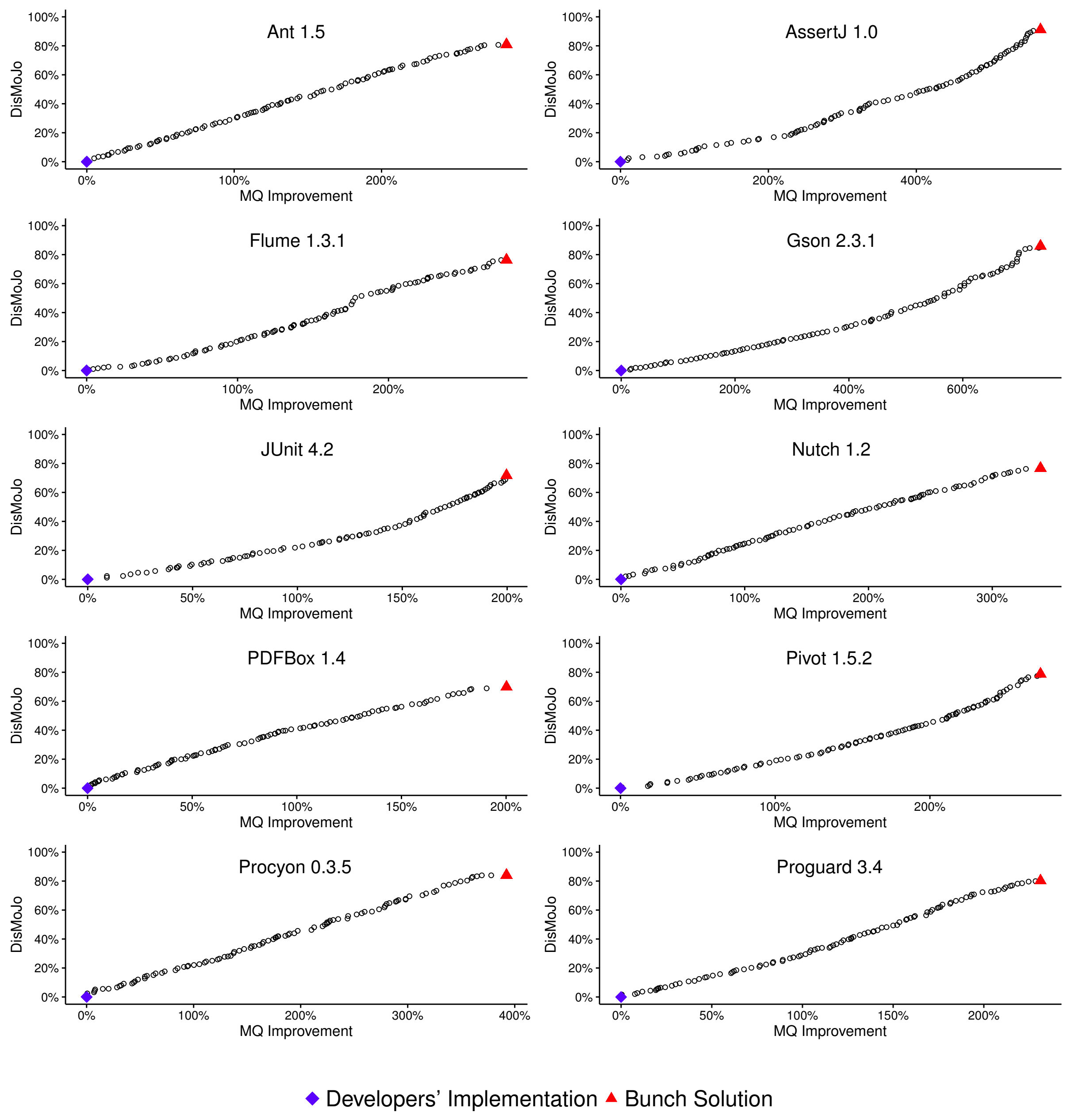


Figure 3.2: Pareto fronts reporting the trade-off between MQ and DisMoJo for the package-free multiobjective search

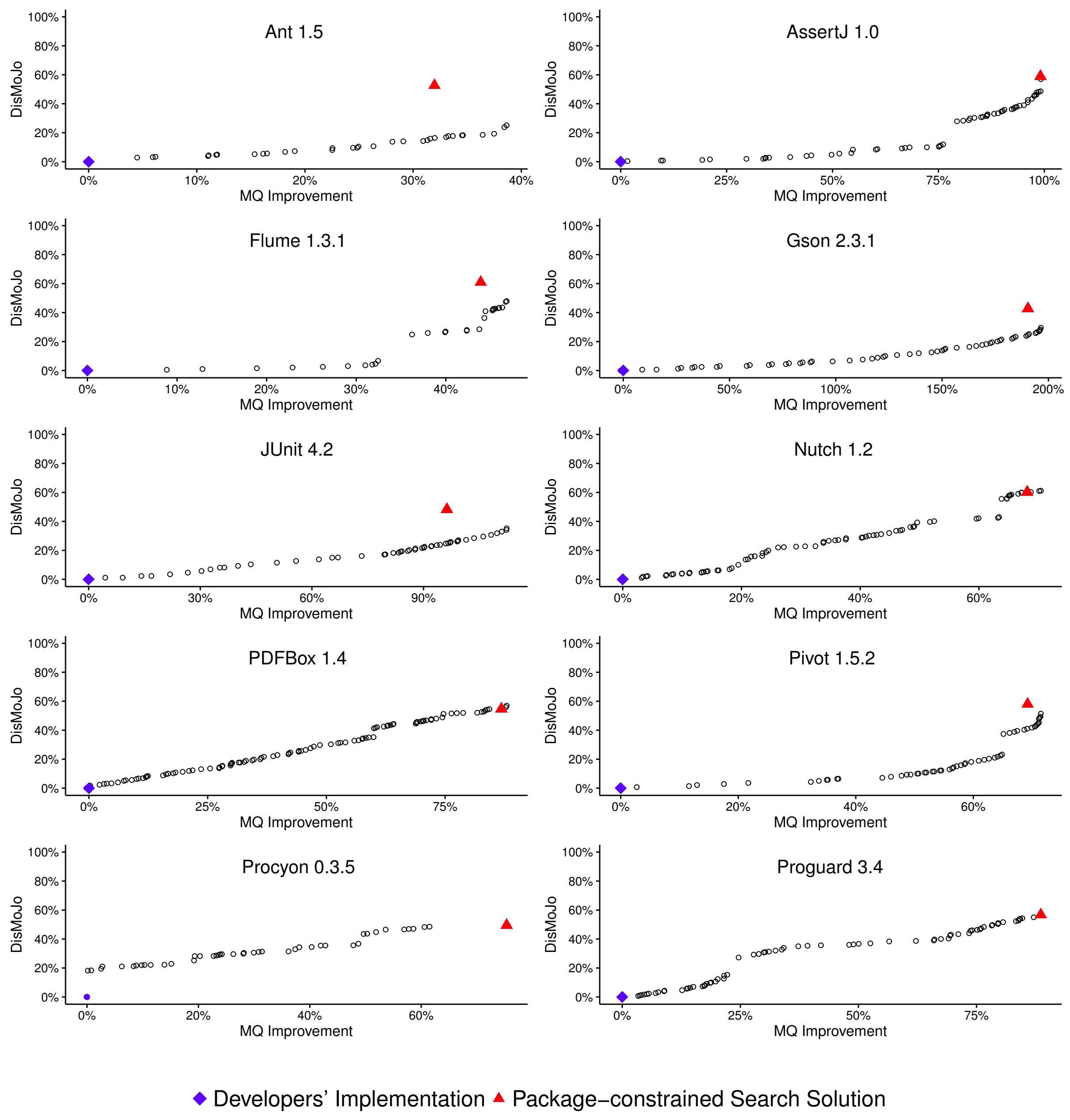


Figure 3.3: Pareto fronts reporting the trade-off between MQ and DisMoJo for the package-constrained multiobjective search

This might be possible because even though we followed what was described in the PhD thesis (Mitchell, 2002) of one of Bunch's creators, the Bunch tool has continued to be improved over the years (Mitchell and Mancoridis, 2006, 2007), so that our implementation might only be able to find local optima modularisations. According to Table 3.4, the standard deviation of our implementation of the hill climbing search is higher than Bunch's, which may be an indicator of the conjecture above. However, it might also be the case that the MQ search space of the package-constrained environment is different than the package-free one, where solutions with bigger MQ improvement can be found on the neighbourhood of solutions with small disruption.

The main goal of RQ4 (and Figures 3.2 and 3.3) is to illustrate the trade-off between improvement in modular structure and disruption to the original implementation that can be achieved with multiobjective search. The state of the art techniques for automated software modularisation, both single (Mitchell and Mancoridis, 2006) and multi (Praditwong et al., 2011) objective, are mostly concerned with modularity improvement, which we know usually causes a large disruption to the original implementation (see RQ3). Previously, developers who would like to optimise the modular structure of their systems using search-based approaches would have two choices: (i) improve the system as much as possible and thereby considerably change the original structure, or (ii) keep the original implementation and do not perform any improvement. With the multiobjective approach proposed to answer RQ4.1 and RQ4.2, developers would have a wider range of options.

The analyses performed in RQ4 took into consideration all solutions in the computed Pareto fronts, providing general insights into the shape of the fronts and on the quality of the solutions within the fronts. In RQ5 we show how developers can pick a particular modularisation from the Pareto fronts according to their needs and constraints.

3.4.5 RQ5: What is the modularity improvement provided by the multiobjective search for acceptable disruption levels?

In RQ4 we showed that the proposed multiobjective search can find solutions that improve the modularity of the original developers' implementation, as assessed by MQ and cohesion, especially for package-constrained search. However, we did not discuss how developers can use the proposed multiobjective approach. We believe that the multiobjective optimisation of modularity and disruption can be used by developers at different moments during the software lifecycle, depending on how much disruption they are willing to endure in order to achieve modularity improvement.

As an example, consider the scenario where developers are planning a major release of the software system. Since it is a major release, the system will possibly undergo large changes to accommodate the new features. In this case, developers can take advantage of the fact the system is going to undergo substantial change, and perform large restructurings to improve the modular structure. On the other hand, in minor or bug-fixing releases, developers may be less willing to change the modular structure, therefore, favouring smaller changes. However, this 'acceptable disruption' level is not obvious.

Therefore, in this research question, we introduce three different methods to estimate the 'acceptable' level of modularity disruption that can be sustained by developers in order to obtain modularity improvement. All methods are based on a longitudinal analysis of the developers' implementations of each release of each system under study. Later, we show how these different 'acceptable' levels of disruption can be used to select solutions from the Pareto fronts found by the multiobjective search approach.

RQ5.1: What is the longitudinal modular disruption introduced by developers?

As a software system evolves, new features are added, changed or removed. Hence, the modular structure of the system needs to change in order to cope with the new requirements and demands. Therefore, the modular structure of a software system is constantly disrupted by its developers during the system's lifetime, which we call the 'natural disruption' of the system. Although the 'acceptable disruption' level that developers are willing to endure to improve the modularity is difficult to measure, we argue that the 'natural disruption' level that developers introduced during the system evolution is a good proxy. Thus, we introduce three different methods to assess the 'natural disruption' of the systems under study, each of which is used as an estimation of the 'acceptable' level of disruption.

The first two methods use the DisMoJo metric in a different way than used in RQ3 and RQ4. $DisMoJo(A, B)$, as defined in Section 3.5, is used to measure the disruption between A and B when both modularisations are composed by the same set of classes. Therefore, since classes can be added or removed between two different releases of the same system, DisMoJo cannot be used to measure the disruption between releases of the same system. In order to provide a lower and an upper bound of the disruption between releases, we introduce Intersection DisMoJo and Union DisMoJo, respectively.

Consider two subsequent releases A and B of the same system. Intersection DisMoJo is computed by considering only the subset of classes that belong to both A and B. We say this is a lower bound disruption between releases because it considers the minimum number of classes that can be moved between releases. Accordingly, Union DisMoJo is computed by aggregating all classes that belong to both A and B, where classes that belong to A but do not belong to B, and vice-versa, are allocated to a separate package. This is a disruption upper bound because all possible classes that can be moved, added or deleted between A and B are taken into account.

Finally, our third method to assess the 'natural disruption' of a software system is based on the analysis of the proportional increase in the number of classes over releases. As the system evolves, the number of classes added in each release is a simple and straightforward way to asses how much of the modular structure changes during the system evolution.

Each of the three methods to assess the 'natural disruption' described above was computed for each pair of subsequent releases of the systems under study, and the results are presented in Table 3.6. For each system, we report the minimum, maximum, median and mean values for each method. This way we can assess what is the biggest and smallest disruption levels each system has undergone during its lifecycle, and also what is the average disruption developers are used to introduce during the systems' evolution.

As one can see, the minimum 'natural disruption' for all systems, according to all three estimation methods, is 0.00%. This means that for all systems, there is at least one pair of subsequent releases that have the same modular structure. The Itersection DisMoJo values are the smallest for all systems (as expected), and for Flume and Procyon, Intersection DisMoJo is

Table 3.6 'Natural disruption' levels caused by developers during the systems' evolution, as assessed by three different methods. Intersection DisMoJo computes the DisMoJo metric considering the intersection of classes between two subsequent releases, while Union DisMoJo computes the DisMoJo metric considering all classes of two subsequent releases. The proportional addition of classes accounts for the proportional increase in the number of classes between two subsequent releases of the same system. Each method was used to compute the 'natural disruption' of each release of each system, and we report the minimum, maximum, median and mean results for each system.

Systems	Intersection DisMoJo				Union DisMoJo				Proportional Addition of Classes			
	Min	Max	Median	Mean	Min	Max	Median	Mean	Min	Max	Median	Mean
Ant	0.00%	2.12 %	0.00%	0.07%	0.00%	36.92%	0.59%	6.48%	0.00%	72.54%	0.05%	11.05%
AssertJ	0.00%	0.79 %	0.00%	0.12%	0.00%	19.34%	3.53%	4.68%	0.00%	47.89%	3.78%	8.50%
Flume	0.00%	0.00 %	0.00%	0.00%	0.00%	36.55%	1.98%	8.60%	0.00%	59.59%	2.37%	14.52%
Gson	0.00%	16.45%	0.00%	1.23%	0.00%	38.71%	6.76%	10.60%	0.00%	64.70%	8.31%	14.09%
JUnit	0.00%	9.09 %	0.00%	0.79%	0.00%	41.89%	2.92%	9.34%	0.00%	196.93%	4.34%	23.30%
Nutch	0.00%	0.43 %	0.00%	0.03%	0.00%	35.15%	1.82%	5.67%	0.00%	51.77%	2.41%	7.09%
PDFBox	0.00%	2.06 %	0.00%	0.10%	0.00%	26.72%	1.57%	5.41%	0.00%	35.00%	2.54%	8.24%
Pivot	0.00%	0.79 %	0.00%	0.07%	0.00%	14.53%	1.14%	5.77%	0.00%	32.03%	1.08%	9.47%
Procyon	0.00%	0.00 %	0.00%	0.00%	0.00%	3.78 %	0.02%	0.54%	0.00%	3.03%	0.38%	0.63%
Proguard	0.00%	11.48%	0.00%	0.58%	0.00%	56.36%	1.28%	5.32%	0.00%	75.37%	1.55%	6.56%

always 0.00%. These results add evidence to the observation in RQ2.3 that existing classes rarely move between packages. Furthermore, all disruption values reported by both Intersection and Union DisMoJo lie within the range of the proportional addition of classes, which is a straightforward way for developers to understand the 'natural disruption'.

RQ5.2: How much modularity improvement can be achieved within lower and upper bounds of 'acceptable' disruption?

In this analysis, the 'natural disruption' levels computed in RQ5.1 are used as proxies for the 'acceptable' level of disruption that developers would be willing to endure in order improve the modular structure of their systems. As previously mentioned, developers have different 'acceptance' levels at different moments of the software lifetime. Thus, we report the modularity improvement that can be achieved at the lower and upper bounds of the 'acceptable disruption'.

The lower bound denotes the smallest greater than zero disruption level we could ascribe from the average disruption caused by developers over the period of evolution of the systems studied. This is a reasonable lower bound because it is chosen to be the lowest possible value (median or mean, using either intersected or unioned DisMoJo) over all releases, for each system. If the developers are prepared to tolerate this amount of disruption during the system's development, on average, then it is not unreasonable that they might allow this amount of disruption when it can occasionally improve the modular structure.

The upper bound denotes the largest possible disruption value we can ascribe from the disruption caused by developers in any release of the system studied (using either intersected or unioned DisMoJo). This is a reasonable upper bound because we know that there does exist a release of the software that causes this level of disruption, and therefore we know that it was, at least on one occasion, tolerated by the developers.

Therefore, for each system, we identified the lower and upper bounds of 'acceptable disruption' as described above. Next, we selected modularisations from the Pareto fronts computed in RQ4

Table 3.7 Modularity improvement, as assessed by cohesion and MQ, that can be achieved within the lower and upper bounds of the 'acceptable disruption' level.

Systems	**Package-Free**				**Package-Constrained**			
	Lower Bound		**Upper Bound**		**Lower Bound**		**Upper Bound**	
	Coh	**MQ**	**Coh**	**MQ**	**Coh**	**MQ**	**Coh**	**MQ**
Ant	0.00%	0.00%	1.24%	148.00%	0.00%	0.00%	7.66%	35.21%
AssertJ	0.00%	0.00%	0.52%	157.01%	0.00%	0.00%	4.73%	41.02%
Flume	0.00%	7.82%	0.54%	168.65%	1.25%	15.21%	21.35%	59.73%
Gson	1.28%	8.57%	8.97%	365.62%	4.70%	17.09%	40.98%	121.53%
JUnit	0.10%	0.00%	3.16%	148.37%	0.76%	2.22%	22.21%	49.09%
Nutch	0.00%	0.00%	0.00%	159.76%	0.00%	0.00%	1.89%	70.03%
PDFBox	0.00%	0.00%	2.73%	66.65%	0.00%	0.00%	6.61%	45.47%
Pivot	0.00%	0.00%	0.46%	83.23%	0.00%	0.00%	9.80%	45.44%
Procyon	0.00%	0.00%	0.00%	8.32%	0.00%	0.00%	0.00%	0.00%
Proguard	0.00%	0.00%	6.19%	200.29%	0.21%	0.00%	110.00%	94.05%
All	0.13%	1.63%	2.38%	150.28%	0.69%	3.45%	22.52%	55.75%

according to these lower and upper bounds. Consider the Ant system, for example. The lower and upper bounds for 'acceptable disruption' were identified as 0.07% and 36.92%, respectively. For each release of Ant we selected the solutions with the best cohesion and MQ improvements found by both package-free and package-constrained search approaches that have a DisMoJo value equal or smaller the lower and upper bounds of 'acceptable disruption'. Results for all systems under study are reported in Table 3.7.

As can be seen from the table, the modularity improvements achieved within the lower bound disruption, for both package-free and package-constrained are small. In fact, for most of the systems, neither package-free nor package-constrained has found any improvement in neither cohesion nor MQ within the lower bound disruption. However, for some software systems, it is possible to have modularity improvements even considering a lower bound disruption, such as Gson, where package-constrained search found a 4.70% cohesion improvement within the minimum 'acceptable disruption' level.

The small modularity improvement achieved within the lower bound disruption is due to the fact that the changes performed to the classes that remain between releases (Intersection DisMoJo) are usually small, as presented in Table 3.6. Nevertheless, small modularity improvements can still be relevant. Cohesion improvements that are focused on a core package or sub-module of the system might be small, but still represent a great impact on the overall maintainability and understandability, for example. Correspondingly, the qualitative analysis in Section 3.8 reports relevant restructurings suggested within the lower bound disruption for one of the systems under study.

As expected, modular improvements within the upper bound disruption levels are the biggest for all systems. When considering the biggest disruption level the systems have already undergone, package-constrained search was able to find modularisations with considerable cohesion improvements, such as 40.98% and 110.00% for Gson and Proguard, respectively.

As an answer to RQ5, multiobjective search can find modularisations with improved modular structure, as assessed by both cohesion and MQ, even within lower and upper bounds of disruption introduced by developers between releases.

3.5 Qualitative Analysis

In this section, we select one of the systems we studied in our empirical study and describe with more details some of the results we achieved throughout our research questions. Table 3.8 reports detailed results for each release of JUnit, including the cohesion and MQ values of the original developers' implementations and the results achieved by Bunch, package-constrained and multiobjective search. Finally, we also report the natural disruption between all releases of the system. We have chosen JUnit because it presented a wide range of modularity variation during its releases, enabling us to illustrate different aspects of the studies we performed.

Table 3.8 Detailed results for all releases of JUnit. For each release, we report the raw cohesion and MQ values for the original developers' implementation and the results achieved by both Bunch and Package-constrained search. In addition, we report the lower and upper bounds of the natural disruption between releases, computed by Intersection and Union DismoJo, respectively. Finally we report the cohesion and MQ results achieved by the proposed multiobjective approach to maximise modularity improvement and minimise disruption, where we use the natural disruption of each release to pick a solution from the Pareto front.

Release	Original Implementation		Bunch		Package-constrained HC		Multiobjective Package-constrained				Natural Disruption	
							Lower Bound		Upper Bound			
	Cohesion	MQ	Cohesion	MQ	Cohesion	MQ	Cohesion	MQ	Cohesion	MQ	Lower Bound	Upper Bound
3.7	175	2.92	118 ± 3	10.69 ± 0.15	186 ± 9	4.78 ± 0.17	175	2.92	175	2.92	0.00%	0.00%
3.8	175	3.02	116 ± 2	11.00 ± 0.07	198 ± 7	4.28 ± 0.15	175	3.02	183	3.55	0.00%	7.50%
3.8.1	176	3.03	115 ± 2	11.06 ± 0.04	194 ± 7	4.21 ± 0.13	176	3.03	177	3.13	0.00%	1.27%
3.8.2	183	3.10	126 ± 2	11.60 ± 0.06	201 ± 8	4.33 ± 0.17	183	3.10	183	3.24	0.00%	2.50%
4.0	147	3.83	97 ± 6	11.02 ± 0.07	193 ± 9	6.31 ± 0.29	167	5.27	185	6.07	9.09%	39.86%
4.1	164	4.01	105 ± 6	11.72 ± 0.06	215 ± 11	6.62 ± 0.30	164	4.01	174	4.72	0.00%	3.66%
4.2	164	3.98	107 ± 7	11.95 ± 0.06	214 ± 9	7.14 ± 0.31	164	3.98	169	4.34	0.00%	1.20%
4.3	541	3.82	365 ± 15	28.90 ± 0.06	711 ± 23	9.15 ± 0.27	541	3.82	668	5.14	0.00%	2.92%
4.3.1	168	4.03	114 ± 6	12.01 ± 0.06	213 ± 9	6.93 ± 0.33	168	4.03	168	4.03	0.00%	1.10%
4.4	232	6.85	173 ± 5	17.40 ± 0.05	323 ± 13	10.67 ± 0.30	256	7.58	298	9.83	2.60%	41.89%
4.5	265	7.39	220 ± 14	20.89 ± 0.05	373 ± 16	13.20 ± 0.37	282	8.48	345	11.15	2.40%	32.42%
4.6	297	8.53	223 ± 10	22.99 ± 0.06	412 ± 13	14.31 ± 0.35	297	8.53	339	10.54	0.00%	5.43%
4.7	320	9.06	236 ± 8	25.78 ± 0.10	447 ± 17	14.90 ± 0.39	320	9.06	365	11.18	0.00%	5.34%
4.8	327	9.70	243 ± 8	26.62 ± 0.07	452 ± 12	15.48 ± 0.37	327	9.70	327	9.70	0.00%	0.00%
4.8.1	327	9.70	246 ± 16	26.58 ± 0.08	455 ± 17	15.51 ± 0.33	327	9.70	327	9.70	0.00%	0.00%
4.8.2	327	9.70	236 ± 9	26.59 ± 0.09	455 ± 18	15.49 ± 0.37	327	9.70	327	9.70	0.00%	0.00%
4.9	334	9.49	248 ± 9	27.70 ± 0.09	469 ± 12	16.08 ± 0.29	334	9.49	352	10.75	0.00%	2.33%
4.10	336	9.48	278 ± 14	27.72 ± 0.09	469 ± 15	15.99 ± 0.40	336	9.48	340	10.39	0.00%	1.83%
4.11	311	8.71	249 ± 9	27.34 ± 0.06	404 ± 16	15.45 ± 0.49	312	9.34	339	11.80	0.49%	6.19%
4.12	471	11.00	334 ± 11	33.94 ± 0.08	603 ± 14	18.04 ± 0.42	472	11.61	509	14.72	0.50%	22.05%

The second and third columns of Table 3.8 report the cohesion and MQ values of the original modularisation implemented by JUnit developers for the 20 subsequent releases we collected. Both cohesion and MQ metrics are affected by the size of the system, where a higher number of classes and dependencies usually leads to a higher cohesion and MQ. Therefore, different releases of JUnit cannot be compared by either MQ or cohesion. However, the techniques described in this chapter aim at optimising the modular structure of each release in particular, so that comparisons within a certain release are valid.

The average and standard deviation values of cohesion and MQ for Bunch search are reported in the fourth and fifth columns of the table, while the results for package-constrained search are reported in the sixth and seventh columns. As discussed in RQ2, the cohesion of the Bunch optimised modularisations is always lower than their original counterparts, even though the MQ is considerably higher. Differently, all package-constrained solutions are able to improve upon the original implementation in both cohesion and MQ.

In release 4.2, for example, the average Bunch solution has a cohesion value of 107 while the developers' implementation has a cohesion value of 164, which corresponds to a difference of -34.94%. On the other hand, the average package-constrained modularisation for release 4.2 has a cohesion value of 214, which represents an improvement of 30.16% over the original modularisation. This particular case elucidates the MQ 'inflation effect' discussed in RQ2, showing how package-constrained search can be used to avoid this undesirable behaviour and improve the structural cohesion of software systems.

In the last columns of Table 3.8, we report the natural disruption caused by each release of JUnit, in comparison to the previous immediate release. For each release, we computed lower and upper bound levels of disruption according to Intersection and Union DisMoJo (described in RQ5), where the first is a disruption measurement that considers only the classes that remained between releases and the latter considers not only classes that remained but also classes that were added or removed between releases.

As one can see, the most disruptive release of JUnit was release 4.4 with an upper bound disruption of 41.89%, yet still smaller than the average disruption caused by Bunch and package-constrained search (see Table 3.5). This observation adds evidence to the claim that even though cohesion and coupling optimisation is achievable, complete restructurings are unrealistic in real-world software development, so that approaches that seek for a compromise between modularity improvement and familiarity to previously established structure are more likely to be adopted by software developers.

Therefore, we report on columns 8-11 of Table 3.8 the results achieved by the proposed multiobjective approach for modularity improvement and disruption minimisation. For each release of JUnit, we used the lower and upper bounds of natural disruption to pick solutions from the Pareto front. Consider release 4.5, for example. For the lower bound cohesion value, we picked the modularisation from the Pareto front with highest cohesion and disruption smaller than 2.40%. Similarly, for the upper bound cohesion improvement, we picked the solution with highest cohesion and disruption smaller than 32.42%. This way we are able to suggest modularity improvements that are bounded by the same range of disruption that is already familiar to the system's developers.

As an example, we report part of the Pareto front found by the proposed multiobjective approach for release 4.0 of JUnit in Table 3.9, where duplicate and similar solutions were omitted. For each solution in the table, we present the cohesion value, disruption given by DisMoJo and the number of classes developers would need to move to a different package in comparison to the original implementation. As one can see, the multiobjective approach proposed in this chapter is able to suggest modularisations with different levels of improvement and disruption, in a way that developers can choose the one the better suits the project needs in a specific scenario.

Release 4.0 of JUnit has a lower bound disruption of 9.09%; therefore, modularisation number 4 is the one that presents the most similar level of disruption, as depicted in Table 3.9. This solution moves only 4 classes from the original implementation, affecting only 3 out of the 11 packages in the system. More specifically, class Description is moved from package org.junit.runner to package org.junit.internal.runners, which is a reasonable refactoring since this class is used to describe different test runners in package org.junit.internal.runners. Moreover, class Request is moved from package org.junit.runner to package org.junit.internal.requests,

Table 3.9 Modularisations suggested by the package-constrained multiobjective search for JUnit 4.0. For each solution, we report the raw cohesion, the disruption given by DisMoJo and the number of moved classes in comparison to the original developers' implementation. We also highlight the solutions that best match the lower and upper bounds disruption levels of release 4.0.

Modularisation	Cohesion	Disruption	Number Of Moved Classes
(Original) 1	147	0.00%	0
2	150	3.70%	2
3	155	4.94%	3
(Lower Bound) 4	**162**	**6.10%**	**4**
5	167	9.76%	7
6	168	10.98%	8
7	169	14.81%	11
8	172	17.28%	13
9	175	18.52%	14
10	177	22.22%	17
11	183	23.46%	18
(Upper Bound) 12	**185**	**28.40%**	**22**

which contains all classes related to requests in the system.

On the other hand, developers can select the solution that is more similar to the upper bound disruption caused by release 4.0, which is modularisation 12 in Table 3.9. Interestingly, this solution performs the same modifications discussed above plus some 'follow-ups' to improve the cohesion even more, such as moving other classes related to Request to the org.junit.internal.requests package. In total, this solution moved 22 classes and affected 8 out of 11 packages of the system, achieving a cohesion improvement of 25.85%.

This case study illustrates how multiobjective search can be used in conjunction with longitudinal analysis of disruption to propose a set of modularisation solutions that present a compromise between modularity improvement and familiarity to existing structure, yet still bounded by the level of disruption inflicted by the developers of the system.

3.6 Threats to the Validity

This section describes the threats that might affect the validity of the empirical study reported in this chapter and discusses our attempts to mitigate these threats.

Conclusion Threats are related to the analyses we performed and the conclusions we drew from these analyses. Random and k-neighbourhood searches were executed one million times for each release, while the systematic search covered the whole nearest neighbourhood of the releases under study. Furthermore, both Bunch and Package-constrained search were executed 30 times for each release. In total, our analyses of RQ1-3 were based in more than 466 million different modularisations of the 10 systems and 233 releases under study, which we believe thoroughly accounts for the random nature of the algorithms we applied. In RQ4, the multiobjective approaches were only executed once for each release due to the large computation effort required to run the multiobjective GA for 233 releases of medium to large real-world software systems. However, we applied the Two-Archive GA, which was demonstrated to be stable and perform well in previous work (Praditwong et al., 2011).

Internal Threats consider the design of the experiments we carried out and the effects our design choices might have in our analyses. All the algorithms, fitness functions and parameters were based on previous and widely used literature on automated software modularisation (Mancoridis et al., 1998; Mitchell and Mancoridis, 2006; Praditwong et al., 2011; Wen and Tzerpos, 2004b). Moreover, our data collection was performed based on a clear selection criteria and involved manual validation of all systems, releases and modularity data that were extracted.

External Threats are related to the generalisation of the findings reported by the empirical study. We performed the largest empirical study on automated software modularisation to date, involving subsequent releases of medium to large real-world software systems. Furthermore, we make available in our supporting web page all the modularity data we used in our empirical study to facilitate replications and extensions (Paixao et al., 2017b).

3.7 Conclusion

The notions of software modularisation and cohesion/coupling have been proposed as good practices for software development since the 1970s, and many SBSE techniques have been proposed and evaluated since late 1990s to automate the decomposition of software systems in highly cohesive and loosely coupled modules. However, after surveying more than 30 related papers, we could not identify any study that has investigated the trade-off between the modularity improvement these automated techniques offer and the inherent disruption they cause to the original modular structure of software systems. Moreover, most of the surveyed papers only consider a single version of the systems under study, ignoring the previous releases. Therefore, we performed the largest empirical study on search-based software modularisation so far, involving 233 subsequent releases of 10 medium to large real-world software systems.

This study revealed that the modular structure of existing systems respect the raw cohesion and the MQ quality metrics, where the developers' implementation have better cohesion and/or MQ of more than 96% of the alternative modularisations created by random and neighbourhood search. However, we noticed that raw cohesion and MQ do not commonly agree when assessing the modularity of software systems due to the 'inflation effect' of the MQ metric that we exposed by applying the Bunch tool to the systems under study. Modularisations with more packages favour the MQ metric; therefore, Bunch creates an average of 493.11% new packages and decreases the cohesion of the systems in -46.25%, on average. As an attempt to mitigate the MQ's 'inflation effect', we introduced the package-constrained approach for automated modularisation, in which the search algorithm is constrained by the number of packages implemented by the developers. The package-constrained search was able to find modularisations with an average cohesion improvement of 25%.

Even though search-based approaches can be used to improve the modular structure of software systems as assessed by both cohesion and MQ, we showed that the disruption caused by these approaches is high. On average, developers would have to change 80.39% and 57.82% of the structure to adopt modularisations suggested by Bunch and package-constrained search, respectively. Surprisingly, the disruption caused by Bunch and package-constrained solutions with very best modularity, as assessed by both cohesion and MQ, caused less disruption than the average. Motivated by this opportunity, we employed a multiobjective optimisation approach for automated software modularisation that attempts to maximise the modularity improvement and minimise disruption.

We showed that modularity improvement and disruption have a clear and constant trade-off over the Pareto fronts of all systems under study. Moreover, based on a longitudinal analysis of developers implementations over releases, we estimated lower and upper bounds of ‘acceptable’ levels of disruption that developers have introduced. We found that our new multiobjective approach was able to improve, on average, 3.45% and 22.59% of the cohesion of the systems within this range of ‘acceptable’ disruption.

Finally, we performed a more detailed and qualitative analysis of some of the results we achieved for the JUnit system, where we presented in a series of case studies how the experiments and analyses carried out in this chapter can be used together to provide a full picture of cohesion and coupling optimisation for a certain system. Among other analyses, we showed the evolution of cohesion throughout JUnit’s releases, providing insights on how package-constrained search is able to avoid MQ’s ‘inflation effect’, and how multiobjective search can be used in conjunction with longitudinal analysis of disruption to suggest modularisation solutions.

3.8 Conclusions From the Chapter

As previously mentioned, this thesis aims at understanding how developers organise code elements and modules, and how improvements in the structure can be performed. Hence, this chapter considered a popular approach for software restructuring, e.g., search-based modularisation, and presented the largest empirical study to date on important aspects of such approaches for their adoption by developers.

We showed that the structural architecture implemented by developers exhibit a degree of respect to common structural metrics of cohesion and coupling used in the literature. This validates these metrics as acceptable proxies for the developers’ perception of structural architecture quality, so that manual and/or automated techniques for software restructuring may rely on such measurements to guide their suggestions for improvements and changes.

Moreover, we presented evidence that developers often perform small restructurings of the system instead of complete ‘big bang’ restructurings. This is particularly alarming for state-of-the-art approaches for search-based modularisation since we found such techniques to induce a high-level of disruption to the existing structure of software systems.

4 CROP: Linking Code Reviews to Source Code Changes

In the last chapter, we studied longitudinal architectural modifications between official releases of software systems. We noticed that most of the architectural changes between versions were due to newly introduced code elements and high-level modules. In addition, the newly introduced modules and existing ones tend to remain architecturally stable in the subsequent releases. This indicates that (i) developers do not often restructure code that is architecturally stable, and (ii) new code is commonly released when it already presents a desired level of architectural stability. Hence, a study of fine-grained software changes might reveal different insights on how developers perform software restructuring.

Software changes that restructure the code are performed under a certain context, with a certain intent to achieve a desired goal. Thus, in order to obtain a deeper understanding on how developers perform and deal with software restructuring, one needs to consider not only fine-grained source code modifications but also the context and motivation in which these modifications were implemented. We believe the data generated during the code review process enables these analyses since it provides natural language descriptions and feedback regarding each particular fine-grained software change. Hence, this chapter describes the mining, processing and, collection of the code review data that is the basis for the investigations later discussed in this thesis.

This dataset report was originally published in the proceedings of the International Working Conference on Mining Software Repositories in 2018, and this chapter presents an adaptation of the original paper. Section 4.1 presents the introduction of the dataset as it was written in the original conference paper. Section 4.2 depicts the mining framework we employed to collect the code review data while Section 4.3 shows how we stored the data to facilitate analysis. Section 4.4 presents details and ideas for research that can be carried out using the dataset. Section 4.5 shows the conclusion of this dataset report as it was published in the original paper.

Finally, Section 4.6 concludes this chapter and discusses how the dataset collected and presented in the chapter can be used to achieve the goals set out by this thesis.

4.1 Introduction

In software development, code review is an asynchronous process in which changes proposed by developers are peer-reviewed by other developers before being incorporated into the system (Bacchelli and Bird, 2013). The modern code review process has been empirically observed to successfully assist developers in finding bugs (Mantyla and Lassenius, 2009; Beller et al., 2014), transferring knowledge (Bacchelli and Bird, 2013; Rigby and Bird, 2013) and improving the general quality of a software system. Given its benefits, code review has been widely adopted by both industrial and open source software development communities. For example, large organisations such as Google and Facebook use code review systems on a daily basis (Kennedy, 2006; Tsotsis, 2011).

In addition to its increasing popularity among practitioners, code review has also drawn the attention of software engineering researchers. There have been empirical studies on the effect of code review on many aspects of software engineering, including software quality (McIntosh et al., 2014; Morales et al., 2015), review automation (Balachandran, 2013), and automated reviewer recommendation (Zanjani et al., 2016). Recently, other research areas in software engineering have leveraged the data generated during code review to expand previously limited datasets and to perform empirical studies. As an example, Chapter 5 reports the usage of code review data to analyse whether developers are aware of the architectural impact of their changes.

Code review research relies heavily on data mining. In this context, some researchers have attempted to mine code review data and have made their datasets available for the community (Mukadam et al., 2013; Hamasaki et al., 2013; Gousios and Zaidman, 2014; Yang et al., 2016). However, code review data is not straightforward to mine (see Section 4.2.2), mostly due to difficulties in linking the reviews to their respective source code changes in the repository. This limits the potential research that can be carried out using existing code review datasets. In fact, to the best of our knowledge, there is no curated code review dataset that identifies and provides the complete state of the system's source code associated with a set of code reviews.

Based on this observation, we introduce CROP, the **C**ode **R**eview **O**pen **P**latform: a curated open source repository of code review data[1] that provides, not only the review's metadata like existing datasets, but also links, to each code review, a complete state of the system's source code at the time of review. For each code review in CROP, one will be able to access the source code that represents the complete state of the system when the review was carried out. Thus, researchers will now have the opportunity to analyse code review data in combination with, for example, source code analysis performed by static and dynamic techniques such as profiling, testing, and building. The combination of code review data and source code analysis will facilitate research in areas that previously required a significant amount of human participation, as outlined in Section 4.4.

Gerrit (Pearce, 2006) is a popular open source code review tool that has been widely used in research (Bosu et al., 2014; Yang et al., 2016; McIntosh et al., 2014). In addition, notable open source organisations adopted Gerrit as their code review tool, including Eclipse, OpenStack, and Couchbase. Thus, since CROP focuses on curating code review data from open source software systems, we chose to mine data from projects that adopted Gerrit as their code review tool.

At the time of writing, we have mined the Gerrit code review data for 8 software systems, accounting for a total of 48,975 reviews and 112,617 revisions (see Gerrit's structure in Section 4.2.1). In addition, we have mined and linked 225,234 complete source code versions to each of the 112,617 revisions.

The rest of this chapter is organised as follows: Section 4.2 describes the Gerrit code review

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

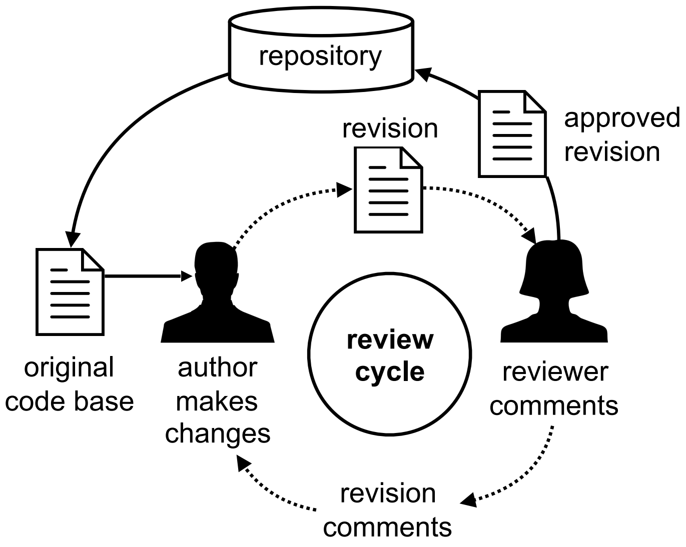


Figure 4.1: Code review process in Gerrit.

structure, the challenges in mining such a dataset and the approach we employed to mine the data. Section 4.3 describes how the data is organised in CROP and Section 4.4 indicates potential research that can be carried out using CROP's data. Finally, Section **??** concludes the CROP discussion.

4.2 Harvesting the CROP

In this section, we first describe the code review process employed by Gerrit followed by a discussion of the challenges of mining this data. Finally, we detail the approach we used to first mine the code review data and later link it to the system's source code.

4.2.1 Code Review in Gerrit

The Gerrit system is built on top of git, and its code review process is outlined in Figure 4.1. A developer starts a review by modifying the original code base in the repository and submitting a new *revision* in the form of a commit, where the commit message is used as the revision's description. For each new review, Gerrit creates a *Change-Id* to be used as an identifier for that review during its reviewing cycle. Other developers of the system will serve as reviewers by inspecting the submitted source code and providing feedback in the form of comments. Revisions that improve upon the current revision according to the received feedback are submitted as new revisions of the same commit. Finally, the latest revision is either *merged* or *abandoned*, where the first indicates the review was incorporated into the system and the latter indicates the commit was rejected.

4.2.2 Challenges in Mining Gerrit Review Data

Gerrit provides RESTful APIs that one can use to access the review's metadata for a project, such as author, description, comments etc. However, linking the reviews to changes in the system's source code is far from straightforward.

As previously mentioned, Gerrit is built on top of git. Thus, the git repository of the system would be the obvious first choice to access the versions of the source code that correspond to the code reviews. However, the system's git repository is an unreliable record because Gerrit constantly *rewrites* and *deletes* history information.

When a new review is submitted by a developer, Gerrit creates a temporary branch in the git repository to be used for review. Every improved revision submitted by a developer is committed to this branch and *replaces* the previous revision through a *commit amend* operation. Therefore, given a merged review, the review's revision history is lost and only the source code of the latest accepted revision can be accessed. Moreover, when developers opt to abandon a review, the current revision is simply deleted from the repository.

In addition to the issues of lost history described above, the system's git repository might also contain inconsistencies if we fail to fully account for the overall review process: code review is a laborious task, and it is common for some reviews to take a few days to complete one iteration of the core cycle (Weißgerber et al., 2008; Jiang et al., 2013; Xia et al., 2015). Between the time a comment is initially submitted and the time the revision is finally merged to the system's repository, other developers might have merged and/or committed other changes to the repository. In this case, each new revision submitted during the review needs to be *rebased* to be up-to-date. Thus, when one reverts the system back to the merged review, the source code will reflect not only the changes due to the revision but also all the other changes that were merged to the repository while the revision was open. These difficulties in isolating the source code changes associated with a specific review pose serious threats to the validity of empirical studies that use code review data.

4.2.3 Mining Code Review Data From Gerrit

We performed a preliminary analysis of the different open source communities that adopted Gerrit and selected the data source we would use for CROP's development. As a result, we identified the Eclipse and Couchbase communities as those that provided all the data we needed to build CROP. The data mining process we employed is outlined in Figure 4.2.

As one can see from the figure, our mining framework consists of 4 sequential phases. The framework is written in Python, and we made it available online[2]. Given a certain Eclipse or Couchbase project, the review harvester explores Gerrit's RESTful API to download the code reviews' metadata for the project. The API returns the data in JSON files that are kept to be used later.

In Phase 2, the snapshot harvester downloads the complete source code of the project for each code review. Both the Eclipse and Couchbase communities have a server that is separated from their git repositories and the Gerrit system where they keep complete snapshots of their

[2]https://github.com/crop-repo/crop

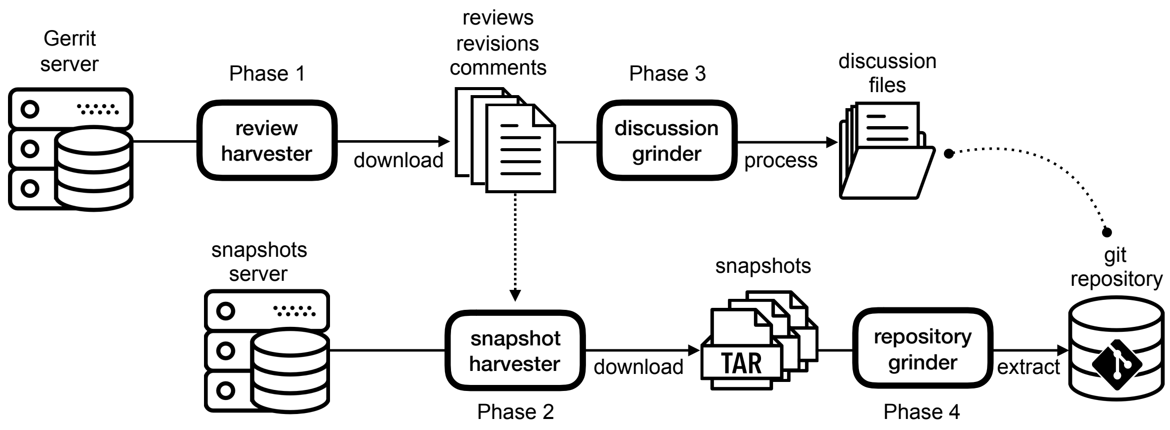


Figure 4.2: The framework employed to mine code review data from Gerrit and link it to complete versions of the code base.

projects for every commit ever performed in the project. These snapshots include the complete code base, i.e., source code, testing code, build files and so on. Thus, for each review, we iterate over all revisions and download the project's snapshots that correspond to the code base both *before* and *after* each revision.

As a result of this process, we were able to access versions of the project's code base that would otherwise have been lost in the official git repository, such as reviews that were abandoned and intermediate revisions that were submitted during the review process. Moreover, by downloading the *before* and *after* versions of the code base for each revision, we guarantee that the observed changes in the code were specifically attributable to the revision.

After downloading all the code reviews' metadata and the project's snapshots (Phases 1 and 2), Phases 3 and 4 handle the data. The discussion grinder processes the code review data stored in the JSON files and creates discussion files for each revision. Discussion files are text files that present, among other information, the review's author, description, and comments in a format that is easy to read and analyse (see Section 4.3 for more details).

In Phase 4, all downloaded snapshots are extracted to a new git repository in order to reduce the disk space occupied by CROP. The repository grinder creates a new git repository and then iterates over each snapshot, automatically extracting and committing the snapshot to the new repository. In the end, every snapshot will be accessible through the new git repository. If CROP would have included the snapshots' raw data from the 8 projects we have mined, the repository would have a size of 4.2TB. Instead, this approach reduced the size of CROP to 7.8GB, which accounts for a 99.8% reduction rate.

4.3 Storing the CROP

The CROP repository is organised as three major directories. A CROP user starts at the metadata directory, where he or she will find a CSV file for each project in CROP. The CSV files contain metadata information about the project's code review and the necessary information to access the project's code base. Since we mined the projects' snapshots revision by revision, each line in the CSV corresponds to a project's revision.

For each revision, we create an **id** to serve as a unique identifier to the revision. The **review_number** column indicates the review to which the revision belongs. The **revision_number** denotes the number of the revision within the review. **Author** and **Status** indicate the revision's author and status, respectively. The **change-id** is the unique identifier that Gerrit creates, as explained previously. We provide a **URL** that can be used to access the revision's data in Gerrit's web view. In the previous section, we showed how we created a new git repository to store the project's code base for each revision. Accordingly, the **before_commit_id** indicates the commit id in the new git repository that should be used to access the project's code base as it was before the revision was submitted. Similarly, one should refer to **after_commit_id** in order to access the code base as it was after the revision was submitted.

The git repositories we created to re-build the projects' reviewing history are contained in the git_repos directory. Each repository has a single master branch, where the before and after versions of the source code for each revision were committed sequentially, based on the review and revision numbers. Such versions are accessible through the commit ids provided in the projects' CSV file, as discussed above.

We store the discussion files for each revision in the discussion directory. This directory follows a tree structure, organised by review number, in which the discussion files for each revision are contained in the directory of its respective review. A discussion file presents reviewing data in the following order: first, the description of the revision is presented, which denotes the commit message of the revision. Such a message includes the revision's **change-id** and **Author**. The comments that were made during the review by other developers are presented next. In the discussion file, we include the author of the comment and the respective message.

4.4 Grinding the CROP

For the first iteration of CROP, we mined data from the four projects with most reviews at the time of mining [3] in the Eclipse and Couchbase communities. Table 4.1 reports statistics concerning the data collected for each of these 8 systems, where the Eclipse projects are presented in the upper section of the table and the Couchbase projects in the lower section. As one can see from the table, all projects have more than 3.5 years of reviewing history, where the data for egit spans more than 8 years. In addition, each project has more than 3,000 reviews and more than 8,000 revisions. In total, CROP provides comprehensive code review data linked to versions of the code base for 48,975 reviews and 112,617 revisions. Finally, these 8 projects are developed in a wide range of programming languages that include Java, C++, JavaScript, Python, Go and others.

CROP is an ongoing research project, where we will periodically update the code review data to reflect the evolution of the systems in the dataset. In addition, we will be constantly mining and including reviewing data for other open source systems. Finally, CROP's code base

[3] At the time of writing, papyrus is the project with most code reviews in Eclipse. However, this was not the case when we started mining.

Table 4.1 Statistics about each project currently in CROP

Systems	Time Span	#Reviews	#Revisions	kLOC	Language
jgit	Oct-09 to Nov-17	5382	14027	200	Java
egit	Sep-09 to Nov-17	5336	13211	157	Java
linuxtools	Jun-12 to Nov-17	5105	15336	270	Java
platform-ui	Feb-13 to Nov-17	4756	14115	637	Java
ns_server	Apr-10 to Nov-17	11346	34317	253	JavaScript
testrunner	Oct-10 to Apr-16	7335	17330	117	Python
ep-engine	Feb-11 to Nov-17	6475	22885	68	C++
indexing	Mar-14 to Nov-17	3240	8316	107	Go

is open[4] for contributions.

4.5 Conclusion

Code review has been widely adopted in the industrial and open source communities due to a number of benefits, such as knowledge transfer, bug detection, and code improvement. Although research in code review is highly dependent on datasets, there is currently no curated dataset that provides code review data that is linked to complete versions of the code at the time of reviewing. To address this limitation, we introduced CROP in this chapter: the **C**ode **R**eview **O**pen **P**latform, an open source repository of code review data that provides links between code reviews and changes in the system's code base. We mined data of 8 software systems, accounting for 48,975 reviews and 112,617 revisions, where we provide not only code review information but also links to versions of the source code that we archived and which would otherwise no longer be available in the systems' original repositories.

4.6 Conclusions From the Chapter

As previously discussed, to obtain a deeper understanding of how and why developers restructure software systems, one needs to study not only the changes in the source code but also the context and motivation in which these changes are performed. Hence, we believe that code review data enables these analyses.

The source code changes within code review are often small and localised, which allows for fine-grained investigations. Moreover, in the code review process, each change is accompanied by a natural language description given by the author of the change and feedback from other developers in the system.

Thus, one can use source code analysis to identify software changes that restructured a software system, in which the context and motivations behind the changes can be inferred

[4]https://github.com/crop-repo/crop

through a follow-up analysis of the changes' description and feedback. The code review data described in this chapter will be used in the investigations that follow in this thesis.

5 Are Developers Aware of the Architectural Impact of Their Changes?

In Chapter 3, we performed an empirical study of architectural changes between official releases of software systems. This chapter complements the latter as it investigates architectural changes at a much finer-grained level by considering individual commits and modifications that had a significant impact on the system's structure. In particular, we are interested in understanding the context and conditions in which architectural changes normally occur. Moreover, since architectural changes usually have a large number of ramifications to the rest of the system, we study whether the developers were aware of the architectural impact at the time they were working in the change. To do this we make use of the CROP code review data described in Chapter 4.

This study has span out two papers. The first paper was published in the proceedings of the IEEE/ACM International Conference on Automated Software Engineering in 2017. We have extended the original conference paper to a work-in-progress journal submission for the IEEE Transactions on Software Engineering. This chapter presents an adaptation of the journal version of the study.

Section 5.1 presents the introduction of the journal submission paper as it was originally written. Section 5.2 discusses the background necessary for this particular chapter while Section 5.3 depicts the experimental design of the empirical study reported in this chapter. Section presents the preliminary studies we carried out to validate our methodological approach. While Section 5.5 shows the quantitative results obtained in the empirical study, Section 5.6 discusses qualitative aspects of some of our findings. Moreover, Section 5.7 discusses how the findings reported in this chapter impact both academic and industrial software engineering practitioners. Section 5.8 discusses the threats to the validity of the study reported in the chapter, and Section 5.9 presents some related work to this chapter in specific. Section 5.10 presents the conclusion of this empirical study as submitted in the original paper.

Finally, Section 5.11 concludes this chapter by contextualising the reported empirical study with the overall goals of this thesis.

5.1 Introduction

Architectural decisions are among the most important decisions to be taken by practitioners (Rozanski and Woods, 2011), due to the high risks and costs accrued by poor architectural design (Williams and Carver, 2010). Recent studies have empirically shown that bug-prone files are more architecturally connected than clean files (Schwanke et al., 2013), and that architectural flaws can lead to increased maintenance effort (Xiao et al., 2016).

The notions of cohesion and coupling as guides for software architecture design have been extensively associated with different aspects of software quality, including, but not limited to, maintainability (Li and Henry, 1993; Zhou and Leung, 2007), comprehensibility (Zhifeng Yu and Rajlich, 2001; Counsell et al., 2005) and code smells (Fowler et al., 1999; Lanza and Marinescu, 2007). Structural dependencies between code components were the most used assets for cohesion and coupling measurement for many years (Hitz and Montazeri, 1995; Briand et al., 1998, 1999; Mancoridis et al., 1998), where other sources of information have been taken into account more recently, such as semantics (Poshyvanyk et al., 2009) and revision history (Beck and Diehl, 2013; Ajienka et al., 2017). Nevertheless, recent studies (Bavota et al., 2013; Candela et al., 2016) have revealed structural dependencies to be one of the best proxies for developers' perception of cohesion and coupling.

Despite the large body of work aimed at aiding developers in the structural organisation of systems (Mitchell and Mancoridis, 2006; Paixao et al., 2017a; Mo et al., 2016), we still see evidence of architectural degradation as systems evolve (Le et al., 2015; de Silva and Balasubramaniam, 2012). Developers sometimes *choose* to accept suboptimal solutions in order to achieve a desired goal, such as short-term delivery (Ernst et al., 2015); thereby accruing technical debt (Kruchten et al., 2012).

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 (Bavota et al., 2013; Candela et al., 2016), different systems and different developers work under different conditions and have different perspectives regarding architectural quality. This diversity indicates the need for studies aimed at better *understanding* of how developers deal with architectural changes.

In this chapter, we extend the body of empirical knowledge regarding architectural changes in software systems by studying these changes on a day-to-day basis. We investigate the *intent* of developers when performing changes that will impact the system's architecture. Moreover, we also assess whether developers are *aware* of the architectural impact of their changes at the time these changes are being made. 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 (Mitchell and Mancoridis, 2006; Paixao et al., 2017a; Ó Cinnéide et al., 2012) 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 (Simons et al., 2015) or selected past changes (Bavota et al., 2013; Candela et al., 2016). Since surveys are subjective to bias (Runeson and Höst, 2009) 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 code change being submitted. The author of the change submits the code 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 code change. In these cases, the author submits new revisions until the change is incorporated into the system or 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 chapter, we adopt CROP (as presented in Chapter 4), 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 are finally merged. It is important to note that 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).

The main contributions of the chapter are listed as follows:

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. We noticed that the architectural feedback provided by developers during code review is often not adequate for improving architectural changes.

3. A framework for the identification of significant architectural changes.

4. 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.

5. 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 (Paixao et al., 2017c), 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 chapter 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 4. In total, 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 performed a sensitivity analysis in order 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 chapter, we describe a qualitative analysis in which we investigated in details the causes for such phenomena.

5.2 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 dependencies capture compile time dependencies, such as method calls, data access and inheritance. In this chapter, the architectural structure of a system is represented as a Module Dependency Graph (MDG), as defined in Section 2.3..

Once the MDG of a system is computed, structural cohesion and coupling measurements can be used to assess the system's structure. In this chapter, we employ structural metrics for cohesion and coupling measurement that have been quantitatively and qualitatively evaluated in a recent study (Candela et al., 2016).

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

$$\text{LStrCoh}(M) = \frac{\sum_{j=1}^{m} \text{LCOF}_{P_j}}{m}, \tag{5.1}$$

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 $\text{LStrCoh}(M_{i+1}) - \text{LStrCoh}(M_i)$ 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

$$\text{StrCop}(M) = \frac{\sum_{j=1}^{m} \text{FanOut}_{P_j}}{m}, \tag{5.2}$$

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

5.3 Experimental Design

The goal of this chapter 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 are 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.

5.3.1 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 chapter, we focus on Gerrit (Pearce, 2006), one of the most popular code review systems currently in use by large open source communities, such as Eclipse (Eclipse, 2018) and

Couchbase (Couchbase, 2018). Although we focus on Gerrit in this chapter, the methodology presented here is adaptable and extensible for other code review systems.

In Gerrit, a developer submits a new 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 code, and provide feedback in the form of comments. Improved code changes 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 chapter, 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 code changes submitted during the reviewing process of a single review according to the feedback from other developers.

In this chapter, we make use of CROP, 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 (Paixao et al., 2017c). Given a certain 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] for additional information on the dataset.

For this particular chapter, 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 5.1 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.

[1]`https://crop-repo.github.io`

Table 5.1 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	91.55	98.66	84.33	70.59	107.661	16.07	59	81	19	641	839	137	2,720	4,017	356
jgit	4,463	11,891	10/09 to 11/17	99.49	99.86	98.56	84.25	114.36	34.00	47	71	19	776	990	338	5,650	7,304	1,965
platform.ui	3,802	12,005	2/13 to 11/17	98.93	99.31	98.88	460.76	472.396	453.295	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	90.11	93.57	85.38	170.28	205.89	89.99	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	100.00	100.00	97.21	9.3	29.16	0.55	16	45	3	184	467	10	704	1,898	14
jvm-core	785	2,184	4/14 to 11/17	100.00	100.00	100.00	13.68	24.59	1.78	43	55	17	328	457	70	1,317	2,093	204
spymemcached	383	1,098	5/10 to 7/17	98.76	99.05	98.45	10.78	13.68	7.19	14	17	11	192	235	133	917	1,113	606

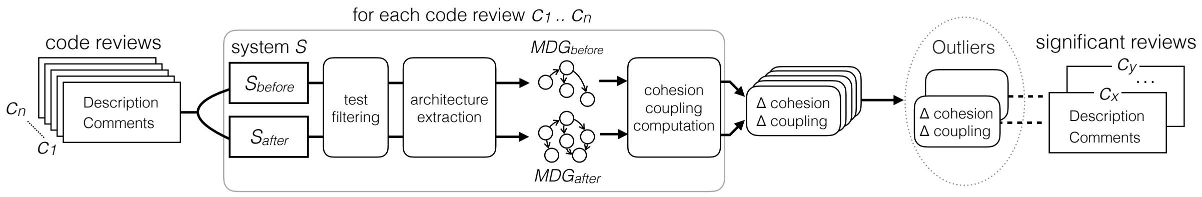


Figure 5.1: Framework for the identification of code reviews with significant changes to the system's architecture. Given a set of code reviews, our automated framework identifies significant reviews in terms of the impact to system's architectural structure.

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.

5.3.2 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 5.3.1, where the formal definitions of the metrics being computed are presented in Section 5.2.

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 5.1. 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 chapter, 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 (Candela et al., 2016; Paixao et al., 2017a; Hall et al., 2014; Beck and Diehl, 2011; Hall et al., 2012; Garcia et al., 2013; Barros et al., 2015). 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 (Scitools, 2018), 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 5.1 and 5.2 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 for 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, each of which indicating the difference in overall cohesion/coupling and the biggest difference in cohesion/coupling for a single package, respectively. In this chapter, 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 dataset of all extracted structural architectures and the respective cohesion and coupling values computed for each revision will be made available at our supporting webpage.

5.3.3 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.

Next, for each of the 8 subsets, we identified the outliers using Tukey's method (Tukey, 1977), and defined the outlier 'fence' as $1.5 \times \text{IQR}$ (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 5.2 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.

Table 5.2 Number of reviews identified as outliers according to different architectural aspects. We report the number of outliers for the reviews with an overall improvement ($\oplus$) or degradation ($\ominus$) in cohesion and/or coupling. We also report the number of outliers for a single package. The number of unique outliers considers all aspects discussed above.

	Coupling				Cohesion				Unique
System	**Overall**		**Single P.**		**Overall**		**Single P.**		**Out-**
	$\oplus$	$\ominus$	$\oplus$	$\ominus$	$\oplus$	$\ominus$	$\oplus$	$\ominus$	**liers**
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

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 stands for 17.5% of all reviews that caused an architectural change.

We discuss and validate this methodology for identification of reviews with significant architectural change in a study described in Section 5.4.

5.3.4 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. (2017). The manual classification process consisted of two researchers 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. (2017). During the open coding classification, we augmented the set of tags with different intents that emerged from the reviews' data in a bottom-up fashion. The final set of tags used in the reviews' classification is presented in Table 5.3, alongside a short description of each tag.

Table 5.3 Tags being used in the manual classification of code reviews.

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

This manual process of inspection and classification of code reviews is paramount to our work. Thus, to make the classification process as transparent as possible, and to allow for better reproducibility, we present in Table 5.4 examples of code reviews that were found during our coding process to have different intents. We present the system and review number of each code review. Moreover, we include the excerpt of the review's description and/or comments that made us classify the review under the specific intent.

Table 5.4 Examples of code reviews classified under different intents.

System	**Review Number**	**Excerpt**	**Intent**
egit	310	"Add option to replace selected files with version in the git index."	New Feature
jgit	17122	"DfsReftableDatabase is a new alternative for DfsRefDatabase that handles more operations (...)"	Enhancement
platform.ui	18057	"Retire org.eclipse.ui.examples.presentation plug-in"	Feature Removal
linuxtools	16258	"Bump to BREE 1.6 to be consistent"	Platform Update
java-client	42324	"Refactor View mapping into distinct class (...). The view query handling is moved into a separate class"	Refactoring
java-core	41436	"Fix failing unit tests introduced by (...)"	Bug Fixing
egit	795	"Merge branch stable-0.8"	Merge Commit

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.

In this chapter, 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.

Similarly to the classification of code reviews into different intents, we present in Table 5.5 examples of code reviews identified as the developers having different types of architectural awareness. We indicate the system the review belongs to, the review number and an excerpt of the description and/or comment that made us identify the architectural awareness.

Table 5.5 Examples of code reviews identified with different types of architectural awareness.

System	Review Number	Excerpt	Awareness
egit	7992	"Puts the code from IgnoreActionHandler into a new IgnoreOperationUI and reuses it in the Staging view."	In Description
jgit	7631	"Make this public and make it available to any command type class. Actually I might just say put it in the main JAR under the io.util package."	In Comments

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

5.4 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.

5.4.1 Measurement of Architectural Change

In this chapter, 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 (Candela et al., 2016; Bavota et al., 2013), measurements in object-oriented systems may be subjective to a size bias (El Emam et al., 2001).

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. We employed the Kendall-τ correlation test (Kendall, 1948), and the correlation coefficients were interpreted as proposed by Cohen (1988).

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 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.

5.4.2 Threshold Sensitivity Analysis

In this chapter, 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 5.3.3). The outliers identification is based on Tukey's method (Tukey, 1977), and 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 outliers identification is $\alpha = 1.5$ (Tukey, 1977), which is the value we have adopted in our previous work (Paixao et al., 2017c). However, different values of α will alter the threshold for the identification of significant architectural changes, which might change the results of our research questions. Thus, for this chapter, we performed a study to evaluate how sensitive our results might be when considering values of α that are different from the default.

The first step for this validation study is a manual inspection of architectural discussion in all code reviews that resulted in changes to the cohesion and coupling of a certain system. Given the total number of reviews that would be necessary to evaluate in a full manual inspection (see Table 5.1), we restricted this analysis to include only the systems from the Couchbase community. 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 5.3.4. For this particular study, we inspected and classified 492 code reviews.

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, our data suggests that the ratio of architectural discussion for the different subsets of outliers is consistent regardless of the value one may attribute to α. In fact, a two-tailed pooled test 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 chapter.

5.5 Experimental Results

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

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

Table 5.6 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 5.4). 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.

Table 5.6 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%)

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 5.2 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.

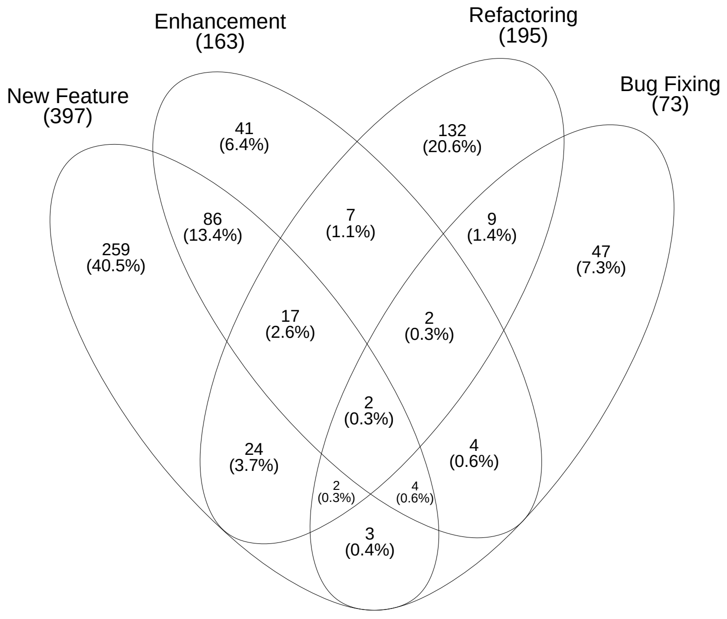


Figure 5.2: Classification of reviews with significant architecture changes for each of the most common intents.

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.

5.5.2 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 5.7 reports the number of reviews with different levels of architectural awareness according to our inspection and classification. Reviews for which the intent is not clear were left out of the analysis. In total, the number of reviews where the architecture is never discussed is higher than the number of reviews where the architecture is discussed in the description, comments or both. This indicates a substantial lack of architectural awareness from developers when performing changes with significant impact 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 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

Table 5.7 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)**			
	Never	**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%)

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.

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

Table 5.8 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 5.8. 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).

Table 5.8 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	—

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 reviews we found evidence of architectural discussion.

As one can see from the table, 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 (Schwanke et al., 2013) and increased maintenance effort (Xiao et al., 2016).

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 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 5.6 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 5.3 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 5.3. For each system, we computed 8 box-plots. First, we report the distribution of cohesion and coupling for the reviews that improved or degraded the overall cohesion and coupling of the system. Next, we report cohesion and coupling for the reviews that improved or degraded the cohesion and coupling of a single package in the system. In all box-plots, smaller values of cohesion and coupling are more desirable for the system's structural architecture.

Consider the box-plots that depict the distribution of cohesion and coupling for the reviews that improved either the system's overall cohesion 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 evidence that architectural awareness has a positive effect on the cohesion and coupling of the systems for the reviews in which the structural architecture was improved. However, for reviews that degrade the system's architecture, apart from specific cases, architectural awareness does not have a noticeable effect 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. Differently, the architecture 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

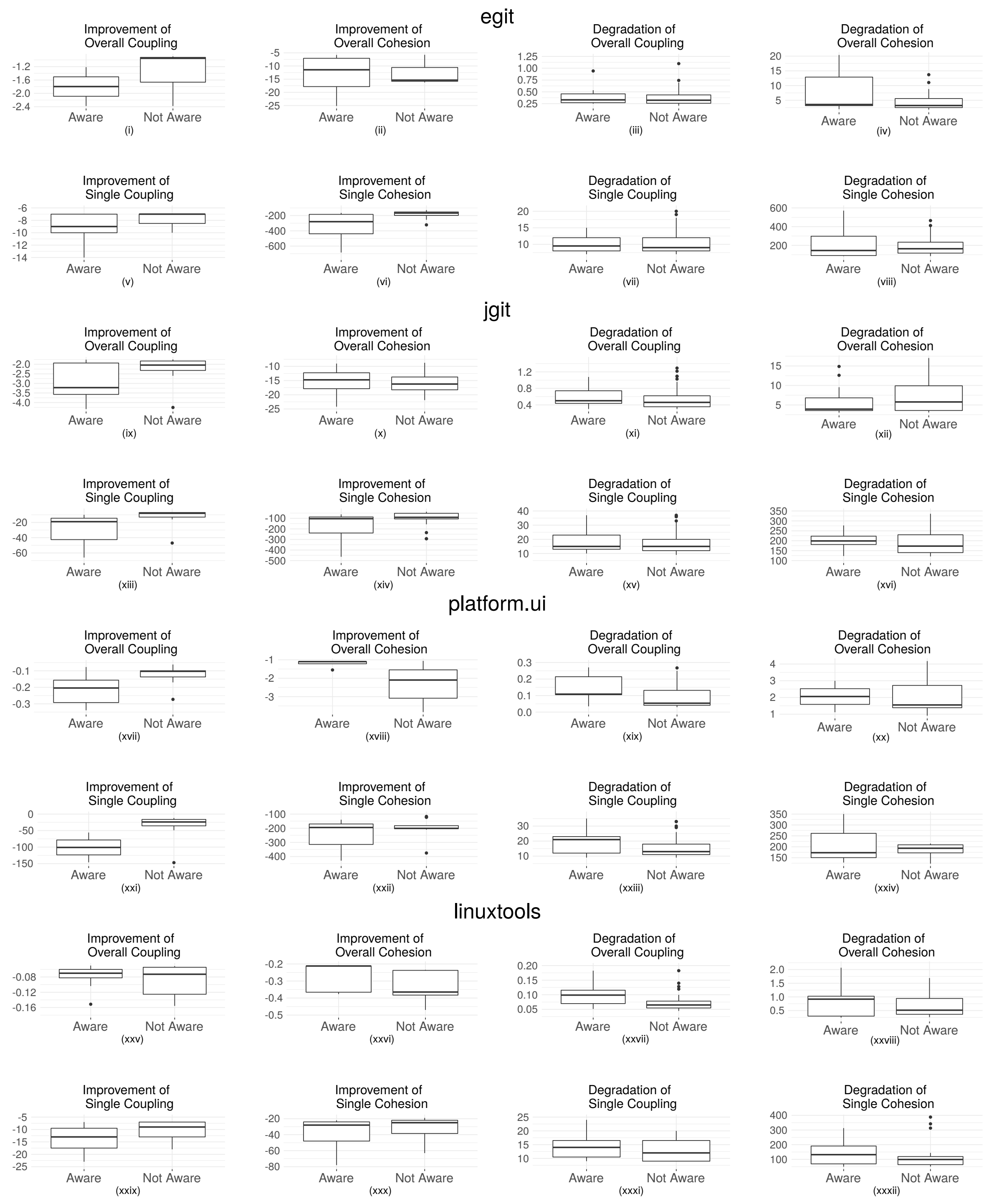


Figure 5.3: Distribution of cohesion and coupling for reviews where we found evidence of architectural awareness and for reviews where we did not. We report box-plots for the reviews that improve and degrade the overall cohesion/coupling of the system and also for the reviews that improve and degrade the cohesion/coupling of a single package in the system.

system's architecture, where the architecture discussion often leads to larger improvements in cohesion and coupling in these reviews.

5.5.4 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 5.3.1, 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 chapter, 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 chapter, a code review may evolve in four different ways. First, the architectural impact remains the *same* throughout the reviewing process, i.e., the cohesion and coupling values are the same for all revisions submitted during the code review. Next, in the case where the last merged revision presents better cohesion and coupling values than all the previous revisions, we consider this code review to have had a *positive* evolution. Similarly, we consider a review to have a *negative* evolution when the cohesion and coupling values of the last revision are worse than the previous revisions. Finally, the reviews in which there are revisions exhibiting both better and worse cohesion or coupling values than the last revision are considered as having a *mixed* evolution.

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 the *same* architectural impact throughout the reviewing process. Differently, 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 code change 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

Table 5.9 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%

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 5.9 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.

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 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. However, we noticed that such reviews 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 expect that architectural feedback during code review will lead to architectural improvements, i.e., a positive evolution. Hence, for the systems we studied, the architectural feedback provided by developers is not resulting in better architectural changes.

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 code change. 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 reviews that cause significant architectural changes.

5.6 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. In order to shed light in this issue, we performed a qualitative analysis of all code reviews in which developers performed a negative refactoring, i.e., reviews in which the merged revision caused a worsening in the system's cohesion and/or coupling metrics.

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 (Bavota et al., 2013; Candela et al., 2016).

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.

5.7 Discussion

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

5.7.1 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 chapter 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 final revision that presents a smaller degradation than the originally submitted revision. Similarly, for revisions that initially improve the system's structure, we expected that architectural discussion would boost the improvement in a way that the final merged revision would be better than the first submitted one. However, our data suggests the opposite, where 33% of reviews with architectural discussion in the comments exhibited a negative evolution. This indicates that more often than not, architectural feedback during code review made architectural degradations worse and architectural improvements smaller.

5.7.2 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. We have observed that architectural awareness and discussion not only leads to better architectural changes, but also that developers take architectural feedback in consideration during the evolution of their changes. 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, given an initial architectural change submitted for review, such a tool could provide suggestions that would lead the change in a better direction in terms of ameliorating architectural degradation and boosting architectural improvement.

5.7.3 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 chapter, 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, 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.

5.8 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 (El Emam et al., 2001). 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 subjective to bias. To mitigate this threat, we employed a two-stage manual classification procedure. In the first stage, all reviews were separately classified by two researchers following a strict guideline previously discussed and agreed by all researchers involved in the classification process. In the second stage, for all reviews in which a disagreement was found, both researchers 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 (Candela et al., 2016; Paixao et al., 2017a; Ó Cinnéide et al., 2012).

5.9 Related Work

Tufano et al. (2017) 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 code, 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 code change, a review also includes feedback provided by other developers and often links to tickets in the issue tracking system and links to related reviews performed in the past. As such, during our manual inspection, we extended Tufano et al. (2017)'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. (2018) extended the investigation on the lifecycle of code smells by considering code smells co-occurrences. By analysing open source systems, the authors 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. (2017) 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. (2015) prepared a set of toy examples and surveyed developers to assess whether metrics represent the developer's perception of quality. Bavota et al. (2013) and Candela et al. (2016) 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 (Tzerpos and Holt, 1999; Mitchell et al., 2001; Zhihua Wen and Tzerpos, 2004). 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. (2015) 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 original conference paper, Shahbazian et al. (2018) 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. Although being a promising metric for architectural change, *a2a* cannot be used to directly measure structural difference at source code level, where an approach for architectural recovery such as ACDC (Tzerpos and Holt, 2000) or ARC (Garcia et al., 2011) needs to be first applied.

Recent studies have evaluated different metrics of structural cohesion and coupling as suitable measurements for architectural quality. In a context of search-based software modularisation, the study reported in Chapter 3 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 developers have some degree of respect by structural measurements of cohesion and coupling. In a similar setting, Ó Cinnéide et al. (2012) 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. (2014) investigated the relationship between software bugs to code coverage and participation during code review. In a similar study, Morales et al. (2015) extends the investigation of code coverage and participation during code review, but now with a focus on the design patterns and anti-patterns.

5.10 Conclusion

Architectural decisions have large 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 degradation is still 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 undergone a process of code review, and we assessed what are 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 in 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 reviewing 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 enhance an existing feature. In addition, we found that refactorings and bug fixing account for 26% and 10% of the reviews with significant architectural changes, respectively.

Surprisingly, we found that the architecture is only discussed 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 that 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 fixed a bug tend to remain the same during the reviewing process. However, feature-related changes tend to undergo adjustments during code review, especially when fellow developers provide feedback regarding the architecture as comments during code review.

5.11 Conclusions From the Chapter

In this chapter, we investigated significant architectural changes performed in the context of code review. We observed that most of the architecturally significant restructurings occurred when developers were introducing a new feature or enhancing an existing feature. Moreover, although we noticed a lack or architectural discussion between developers, we also observed that changes in which developers discuss the system's structure tend to exhibit higher architectural improvements. Finally, our data indicates that currently provided architectural feedback during code review is not assisting developers in performing better architectural restructurings.

All of these findings have considerably broadened our understanding of how developers perform architectural restructuring. From now on, researchers and tool builders will have to consider these observations when proposing methods and tools to assist developers in architectural restructuring and/or improvement.

6 Are Developers Refactoring When Refactoring?

In the previous chapter, we studied architectural changes in the context of code review. Architectural changes might be considered large-scale software restructuring since these changes commonly deal with sets of classes, entire modules, and subsystems. However, not all restructuring operations occur at such large-scale. Developers often restructure minor parts of a software system, such as single classes and methods, in a practice that has become known as software refactoring.

Thus, this chapter presents an empirical study that attempts to better understand the context and conditions in which developers perform low-level refactorings. By automatically identifying refactoring operations performed during code review, we sought to investigate the most common intents developers have when employing refactoring operations. Moreover, we study the types of refactoring operations most commonly employed by developers when performing changes with different intents.

The report presented in this chapter represents a work-in-progress paper to be submitted to the International Conference on Software Engineering (ICSE'19). Thus, Section 6.1 presents the introduction to the paper as it will be submitted. Section 6.2 depicts the design of the empirical study presented in this chapter while Section 6.3 presents our findings. Section 6.4 discusses the threats to the validity of this study. In addition, Section 6.5 presents the work related to the empirical study reported in this chapter.

Finally, Section 6.7 discusses how this chapter fits within this thesis and concludes the chapter.

6.1 Introduction

Software refactoring is the process of changing a software system without altering its behaviour, with the goal of improving the software's internal structure. Refactoring is widely adopted by software engineering practitioners, and it has been linked to improvements in adaptability, maintainability, understandability (Ammerlaan et al., 2015), reusability, testability (Alshayeb, 2009), and productivity (Moser et al., 2008). Refactoring operations were originally proposed as a series of manual steps with the common goal of fixing a design problem and/or code smell (Fowler et al., 1999). Hence, most of the research dedicated at assisting developers through refactoring recommendations and automated refactoring focused on the improvement of quality properties, removal of code smells and code duplication (Tsantalis et al., 2008; Tsantalis and Chatzigeorgiou, 2011; Silva et al., 2014; Tairas and Gray, 2012).

In spite of the plethora of tools and approaches to assist software refactoring, recent studies have shown that developers largely prefer to perform refactorings manually rather than automatically (Murphy-Hill et al., 2012; Negara et al., 2013). Motivations for this include distrust in automated tools and lack of support for the refactoring operations they often need. In addition, these studies observed that refactoring operations are most commonly performed in batches, usually mixed with other activities, such as feature implementation and bug fixing. This phenomenon has become known as *floss* refactoring. In this context, the need to better understand the context, conditions, and motivations behind refactoring operations becomes increasingly important.

Recent work has been done as an attempt to shed light in this direction. Palomba et al. (2018) investigated the likelihood of different refactoring operations to be used in software changes that involve feature development, bug fixing, and pure maintenance activities. The authors observed that some refactoring operations, such as extract method, are statistically more likely to be employed in feature development changes than maintenance tasks. However, only 3 possible intents were considered for each software change, while recent empirical studies that manually categorised developers' intentions have considered from 4 to 7 unique intents, as presented both in Chapter 5 and in the work by Tufano et al. (2017). Furthermore, the classification of each software change into a certain intent was performed automatically through a keyword based technique that originally reported an approximate precision of 61% (Mockus and Votta, 2000), which introduce a considerable threat to the study's validity.

Silva et al. (2016) investigated developers' intentions behind refactorings by monitoring Github projects. During the course of 61 days, the authors detected refactoring operations performed in commits of 124 systems. For each commit in which refactoring operations were detected, an email was sent to the developer responsible for the commit in which she was asked for the motivation behind the performed refactoring. This '*firehouse interview*' approach encourages fast and accurate responses since the developers are asked about activities they performed a few days prior. However, interviews and surveys often suffer from a confirmation bias (Runeson and Höst, 2009) since the questions being asked only concern the phenomena of interest.

Hence, we propose an investigation that moves forward the empirical knowledge on the developers' intentions when performing refactoring operations. Our methodology performs a manual analysis on the developer's intent at the time the change was performed, in a way that avoids any confirmation and/or memory biases that are intrinsic to studies that involve developers. To achieve this, we make use of code review data.

In the code review process, a code change is only incorporated into the system after inspection. The change's author submits the code and a natural language description of the change, where other developers will review the code and provide feedback. Depending on the feedback from the reviewers, the author of the change might need to improve the code. Hence, the author submits new revisions to the code until the change is incorporated into the system.

Apart from playing a crucial role in currently adopted software development models, the code review process generates a rich body of data that can be leveraged for empirical studies in software engineering, as discussed in Chapters 4 and 5. The combination of natural language descriptions and feedback provides a reliable source of information for each particular change, in which the developers' intents and motivations can be inferred. In this context, all the data we study was provided by the change's original developers and reviewers at the time the change was developed and reviewed, which alleviates the memory and confirmations biases from previous studies, as discussed above.

Hence, in this chapter, we analyse the code review data of software changes that perform refactoring operations to complement existing work on understanding the most common intents and motivations developers have when performing refactoring. Moreover, we perform a new study that investigates software changes in which the developers have the intention of refactoring but no refactoring operation is employed. Finally, we provide new insights on how refactoring operations evolve during code review.

6.2 Experimental Design

The goal of this investigation is to study the intent developers have when they employ refactoring operations in the context of code review. To this end, we ask the following research questions:

RQ1: *What are common intents when developers perform refactoring operations?* This research question investigates changes that employ refactoring operations and identifies common intents behind these changes. Thus, we classify changes with refactoring operations regarding their intent at the time the change was reviewed, such as *Feature*, *Bug Fixing*, *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 refactoring operations are employed.
RQ2: *What are common refactoring operations employed under different intents?* The number of possible refactoring operations is large, where each refactoring serve a specific purpose. This research question investigates the most common refactoring operations used when developers have different intents behind a software change.
RQ3: *How often do developers have the intent of refactoring but do not employ any refactoring operation?* The concepts and definitions of refactoring are broad and can be achieved through different means. Thus, this research question investigates how often we can observe software changes in which the developers have a refactoring intent but do not employ any of the refactoring operations we consider.
RQ4: *How do changes that employ refactoring operations evolve during code review?* By comparing the last merged revision to all the other previous revisions of a change that employ refactoring operations, we study how the code review process influences the evolution of these changes.

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

6.2.1 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. For this investigation, we focus on Gerrit (Pearce, 2006), the most popular open source code review system currently in use by large software communities, such as Eclipse (Eclipse, 2018) and Couchbase (Couchbase, 2018).

In Gerrit, a developer submits a new code change for review in the form of a git commit, where the commit message is used as the review's description. Other developers of the system will then inspect the change and provide feedback in the form of comments. Improved code changes are submitted in the form of revisions according to the feedback until the review is *merged* or *abandoned*. For the rest of this chapter, 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 code changes submitted during the reviewing process of a single review according to the feedback from other developers.

In this chapter, we make use of CROP, an open source dataset that links code review data with their respective software changes. Given a certain 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.

For this particular investigation, we adopt all the Java systems included in the CROP dataset for which we could successfully run the refactoring detection tool we chose for the study[1] (see Section 6.2.2). For the Eclipse community, we study egit, jgit, and linuxtools. 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. Table 6.1 reports the number of *merged* reviews for each system, the time span of the system's history we are investigating, and the system's kLOC.

Table 6.1 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 kLOC.

Systems	No. of Reviews	No. of Revisions	Time Span	kLOC		
				Med	Max	Min
egit	4,502	11,430	9/09 to 11/17	70.59	107.661	16.07
jgit	4,463	11,891	10/09 to 11/17	84.25	114.36	34.00
linuxtools	3,695	10,892	6/12 to 11/17	170.28	205.89	89.99
java-client	798	2,394	11/11 to 11/17	9.3	29.16	0.55
jvm-core	785	2,184	4/14 to 11/17	13.68	24.59	1.78
spymemcached	383	1,098	5/10 to 7/17	10.78	13.68	7.19

6.2.2 Identification of Refactoring Operations

We employ RefMiner (Tsantalis et al., 2018) to identify the refactoring operations performed in a certain code review. RefMiner is a tool that automatically detects refactoring operations performed in the code base of two different versions of the same system. RefMiner implements

[1]RefMiner constantly entered infinite loops when executed on the data from eclipse.platform.ui

a rule-based statement matching at AST level to identify refactoring operations performed in a set of elements.

Table 6.2 indicates the refactoring operations detected by RefMiner and the code elements affected by each operation. In its original study, RefMiner achieved 97.96% of precision and 87.20% of recall in a curated refactorings benchmark, which makes RefMiner the current state-of-the-art tool for automated refactoring detection.

Table 6.2 Refactoring operations detected by RefMiner and the code elements affected by each operation

Code Element	Refactoring Operations
package	Move Package, Split Package, Rename Package
type	Move Class, Rename Class, Extract Superclass, Extract Interface
method	Extract Method, Inline Method, Pull Up Method, Push Down Method, Rename Method, Move Method
attribute	Pull Up Attribute, Push Down Attribute

Hence, we used RefMiner to detect the refactoring operations performed in each merged revision of each of the selected systems contained in CROP. When considering all 6 systems, we identified 1,780 code reviews that performed refactoring operations for a total of 7,259 unique refactorings performed. Table 6.3 provides details on the number of reviews and unique refactorings performed for each system. In egit, for example, we identified 531 code reviews in which refactoring operations were employed, which accounts for 11.79% of all code reviews we studied in egit. In addition, we identified 1,734 unique refactorings distributed over the 531 reviews, where the median number of refactorings per review is 1, and the maximum and minimum number of refactorings identified in a single review is 49 and 1, respectively.

Table 6.3 Number of reviews that performed refactoring operations for each system. In addition, we provide statistics on the number of unique refactoring operations performed for each system.

Systems	No. of Reviews With Refactoring Operations	No. of Refactorings			
		All	**Med**	**Max**	**Min**
egit	531 (11.79%)	1,734	1	49	1
jgit	568 (12.72%)	2,375	1	148	1
linuxtools	427 (11.55%)	2,069	2	197	1
java-client	116 (14.53%)	430	2	26	1
jvm-core	91 (11.59%)	468	2	96	1
spymemcached	47 (12.27%)	183	2	21	1

As one can see from the table, the percentage of reviews that perform refactoring operations is consistent throughout all systems we analyse, where we observe from 11% to 14% of reviews

Table 6.4 Intents considered when classifying reviews that performed refactoring operations.

Feature	Developer is adding a new feature or enhancing an existing feature
Refactoring	Developer is refactoring the system
Bug Fixing	Developer is fixing a bug
Feature Removal	Developer is removing an obsolete feature
Platform Update	Developer is updating the code for a new platform/API
Merge Commit	Developer is merging two branches
Not Clear	There's no evidence to suggest any of the previous

that do so. Moreover, the number of refactorings performed per code review tend to be small, as the median number of unique refactorings is either 1 or 2 for all systems we consider.

6.2.3 Manual Inspection and Classification of Reviews

Following the refactoring identification procedure described in the previous section, we considered all 1,780 reviews that employed refactoring operations and performed a manual inspection and classification inspired by the work described in Chapter 5. The manual classification process consisted of two researchers analysing each review and identifying the developers' intent behind the review.

Table 6.5 presents the intents considered during the reviews' classification. In the course of our previous investigation reported in Chapter 5, we noticed a considerable overlap between the 'New Feature' and 'Enhancement' intents. This is due to the incremental and iterative nature of software development, where new features are constantly developed through the enhancement of existing features, and vice versa. Hence, for this investigation, we merged these both intents under a single 'Feature' intent.

In order to mitigate threats to internal validity during the classification process, we employed a two stages classification. In the first stage, two researchers solely inspected and classified the reviews. In the second stage, the researchers discussed all the reviews for which there was a disagreement in the classification. For this investigation, there was no disagreement on any review after the second stage of classification. The set of manually classified code reviews that performed refactoring operations will be made available at our supporting webpage.

6.3 Experimental Results

This section presents and discusses the results we found for each research question.

6.3.1 RQ1: What are common intents when developers perform refactoring operations?

Table 6.5 presents the number of reviews that performed refactoring operations grouped by intent. Note that one review might have more than one intent. Consider egit, for example. A total of 413 reviews that employed refactoring operations were found to have a Feature intent, which accounts for 77.78% of all reviews that employed refactoring operations in egit. In addition, in only 17.14% and 15.25% of the egit's reviews the developers had the intention of Refactoring or Bug Fixing, respectively.

These findings hold when one considers all systems under study. As one can see, most of the software changes that employ refactoring operations are either introducing a new feature or enhancing an existing feature in the system, accounting for 63.09% of all reviews. The second and third most common intents behind code reviews that employ refactoring operations are Refactoring (31.12%) and Bug Fixing (13.76%), respectively. For this investigation, the number of reviews identified as having an intent of Feature Removal, Platform Update and Merge Commit are negligible in comparison to the most popular intents previously discussed.

Table 6.5 Number of reviews that performed refactoring operations grouped by different intents.

Systems	Feature	Refactoring	Bug Fixing	Feature Removal	Platform Update	Merge Commit	Not Clear
egit	413 (77.78%)	91 (17.14%)	81 (15.25%)	7 (1.32%)	4 (0.75%)	19 (3.58%)	7 (1.32%)
jgit	326 (57.39%)	161 (28.35%)	78 (13.73%)	1 (0.18%)	1 (0.18%)	42 (7.39%)	12 (2.11%)
linuxtools	230 (53.86%)	200 (46.84%)	59 (13.82%)	4 (0.94%)	3 (0.70%)	1 (0.23%)	18 (4.22%)
java-client	74 (63.79%)	48 (41.38%)	13 (11.21%)	1 (0.86%)	1 (0.86%)	0 (0.00%)	2 (1.72%)
jvm-core	59 (64.84%)	37 (40.66%)	12 (13.19%)	1 (1.10%)	1 (1.10%)	0 (0.00%)	3 (3.30%)
spymemcached	21 (44.68%)	25 (53.19%)	3 (6.38%)	0 (0.00%)	0 (0.00%)	1 (2.13%)	0 (0.00%)
All Systems	1123 (63.09%)	562 (31.57%)	246 (13.82%)	14 (0.79%)	10 (0.56%)	63 (3.54%)	42 (2.36%)

These results indicate that developers most commonly employ refactoring operations when performing *floss refactoring*, i.e., the refactoring is mixed with other changes, normally as preparation for the implementation of the new feature or for the fixing of a bug. These observations are aligned with previous empirical investigations regarding the phenomenon of *floss refactoring* and motivations behind refactoring operations (Murphy-Hill et al., 2012; Palomba et al., 2018; Silva et al., 2016).

As an answer to RQ1, we observed that refactoring operations are mostly employed in feature-related changes, accounting for 63.09% of all reviews. Only in 31.57% of the time developers have the intention of refactoring when they perform refactoring operations. Finally, bug fixing appears as the third most common intent, where developers refactored the system to fix a bug 13.82% of the time.

6.3.2 RQ2: What are common refactoring operations employed under different intents?

In this investigation, we classified each code review under 7 different intents. Moreover, each review may employ from 1 to 15 different refactoring operations. Hence, Figure 6.1 presents the number of code reviews that employ each different refactoring operation grouped by the developers' intent behind the review. We present the results for the most popular intents as discussed in the previous research questions, i.e., Feature, Refactoring, and Bug Fixing. For this analysis, we do not take into account the amount of refactoring operations of the same type used in a review. Instead, we simply consider whether a review performs a certain refactoring operation or not.

Consider the feature-related reviews for example. As one can see, around 38% of these reviews performed an extract method refactoring. This is more than double the number of reviews that rename a method, which accounts for the second most common refactoring when implementing a feature with 16% of the reviews. This is an expected result since feature-related changes are the most common when developers perform refactoring (see RQ1), and method extraction is a common preparation for feature implementations and enhancements. Move method and move attribute refactorings follow as the third and fourth most common operations in feature-related changes, appearing in 10% and 8% of the reviews.

When considering reviews in which the developers had the intent of refactoring, these observations vary. Extract method is still the most common operation, but it accounts for only 21% of reviews, where rename method and move method account for 16% and 15% of the reviews, respectively. Hence, developers tend to use a more even distribution of refactoring operations when they have the intent of refactoring the system. Although the ranking of most common operations between changes with Feature and Refactoring intents are similar, an outlier can be observed for the move class refactoring operation. For feature-related reviews, developers only move classes in 4% of the time, where classes are moved in 10% of the reviews with a refactoring intent.

The refactoring operations employed in reviews that fix bugs are similar to the ones where developers implement features. Extract method is employed 42% of the time, followed by rename method and move method, both employed at 21% and 10% of the times.

As an answer to RQ2, extract method is the most common refactoring operation for all intents. However, when considering reviews in which we observe refactoring intentions, the number of refactoring operations is more evenly distributed. Furthermore, some refactoring operations, such as move class, are considerably more employed in reviews with a refactoring intention than otherwise.

6.3.3 RQ3: How often do developers have the intent of refactoring but do not employ any refactoring operation?

In both RQ1 and RQ2, we studied code reviews that performed refactoring operations as identified by RefMiner. However, RefMiner only supports the identification of 15 refactoring operations, while some refactoring catalogues provide details for more than 70 refactoring operations (Fowler et al., 1999). Moreover, Chapter 5 has shown that developers discuss and use the ‘refactoring’ term in a broader and more relaxed manner than defined in textbooks and research papers (Fowler et al., 1999; Tsantalis et al., 2018; Silva and Valente, 2017). Thus, this research question investigates the code reviews in which the developers have the intention of refactoring but do not employ refactoring operations.

To do this, we would need a complete ground truth of all code reviews with a refactoring intent for all systems under study. This is infeasible given the number of code reviews provided by CROP for each system (see Table 6.1). However, in Chapter 5, we classified the developers’ intent behind code reviews following a similar procedure to the one employed in this investigation.

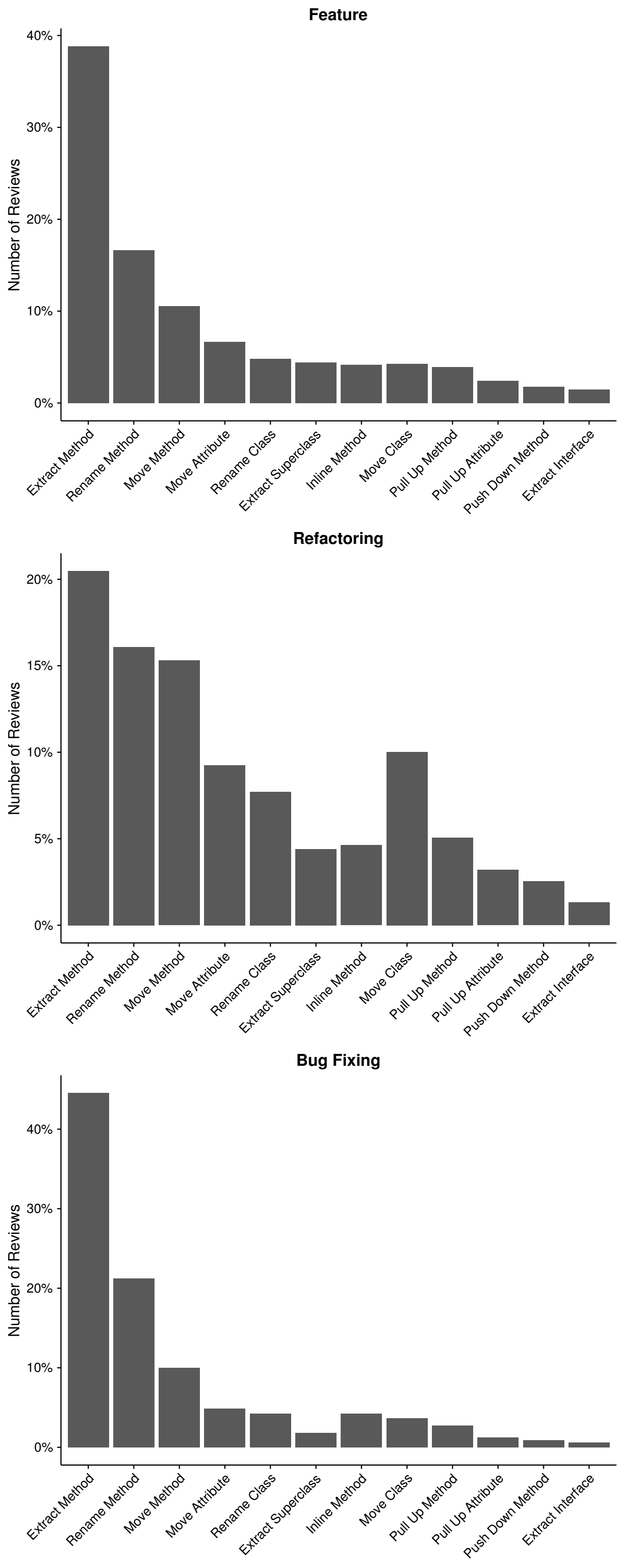


Figure 6.1: Number of reviews that employ different refactoring operations grouped by intent.

Table 6.6 Number of reviews with a refactoring intent that did not employ any refactoring operation grouped by system.

Systems	No. Reviews Refactoring Intent	No. Reviews Refactoring Operations	Recall
egit	83	48	57%
jgit	49	29	59%
linuxtools	122	69	56%
java-client	29	19	65%
jvm-core	35	22	62%
spymemcached	14	9	64%
All Systems	332	196	59%

Hence, in order to answer this question, we make use of the reviews' classification performed in Chapter 5 as a partial ground truth for code reviews with a refactoring intent.

For each review identified by in Chapter 5 as having a refactoring intent, we used RefMiner to identify the refactoring operations that might have been employed. Table 6.6 presents the number of code reviews with a refactoring intent alongside the number of those reviews which employ refactoring operations. Consider egit, for example, the study performed in Chapter 5 identified 83 code reviews as having a refactoring intent, where in 48 of them we observed a refactoring operation being employed, which accounts for 57% of the reviews. Thus, for egit, 43% of the reviews with a refactoring intent did not employ any refactoring operation as identified by RefMiner.

As one can see from the table, when considering all systems under study, 41% of the reviews with a refactoring intent did not perform any refactoring operation. This is a surprising finding as one would expect developers to employ refactoring operations when they have the intent of refactoring the system.

Hence, to shed light on this observation, we separately inspected each refactoring-related review that did not employ a refactoring operation. After an open coding analysis of all these reviews, Table 6.7 presents the most common reasons for reviews with a refactoring intent not to make use of refactoring operations.

As can be seen from the table, 64% of reviews that have a refactoring intent but do not employ refactoring operations are removing dead code. Dead code removal is a refactoring activity as it improves the code base while not affecting the system's behaviour. However, this is not a refactoring operation supported by RefMiner.

Next, 16% of the reviews are performing a floss refactoring in which the refactoring is too tangled with the other changes so that RefMiner cannot identify any refactoring operation. Moreover, we observed a non-negligible (9%) number of false negatives, i.e., reviews in which refactoring operations were employed but RefMiner failed to detect. After a closer analysis, all of the false negative reviews performed an extract method refactoring. Hence, both the *floss* refactorings and the extract method false negatives serve as examples of room for improvements in automated approaches for refactoring identification.

Table 6.7 Reasonings behind reviews with a refactoring intent but no refactoring operations.

Dead code removal	87 (64%)
Floss refactoring	21 (16%)
RefMiner's false negative	12 (9%)
Constant Extraction	4 (3%)
Code replacement	4 (3%)
Remove unused imports	3 (2%)
Adapt code to previous refactoring	3 (2%)
Others	2 (1%)
All reviews	136

6.3.4 RQ4: How do changes that employ refactoring operations evolve during code review?

RQ1–3 consider code reviews as single data points when investigating refactoring operations, i.e., a certain code review either performs a certain refactoring operation during its reviewing cycle or not. However, a code review is composed of a set of different revisions that evolve and change according to the feedback provided by other developers. It is possible that developers introduce a refactoring operation in the second iteration of the code change, for example. Similarly, refactoring operations originally proposed in the first iteration of the code review might be reverted and/or disconsidered before the review is integrated into the system. Hence, this research question expands the study of refactoring operations by considering each revision within a review individually. We investigate how refactoring operations evolve during the process of code review.

To do so, we perform a sequential comparison of all revisions in a code review in the context of the refactoring operations performed by each revision. Consider a code review with three revisions, for example. We sequentially compare the refactoring operations performed in the second revision to the operations performed in the first revision. Next, we compare the refactoring operations performed in the third revision to the ones performed in the second one.

At the end of this procedure, we can observe one of the following five refactoring evolution patterns for each review. First, when we observe a code review in which all revisions present the exact same set of refactoring operations, we consider this review to have the *'same'* refactoring evolution pattern. Second, when a subsequent revision performs at least one refactoring operation that was not performed by the previous revision, this code review is considered to have a *'new'* refactoring evolution pattern. Similarly, code reviews are considered to have a '*delete'* refactoring evolution pattern when a subsequent revision do not perform a refactoring operation that was performed by the previous revision. In the case where we observe both introduction and removal of refactoring operations in subsequent revisions, we consider this code review to have a *'both'* refactoring evolution pattern. Finally, some reviews are composed by a single revision, in which we consider this review to have a *'single'* refactoring evolution pattern.

Figure 6.2 presents the distribution of the different refactoring evolution patterns described

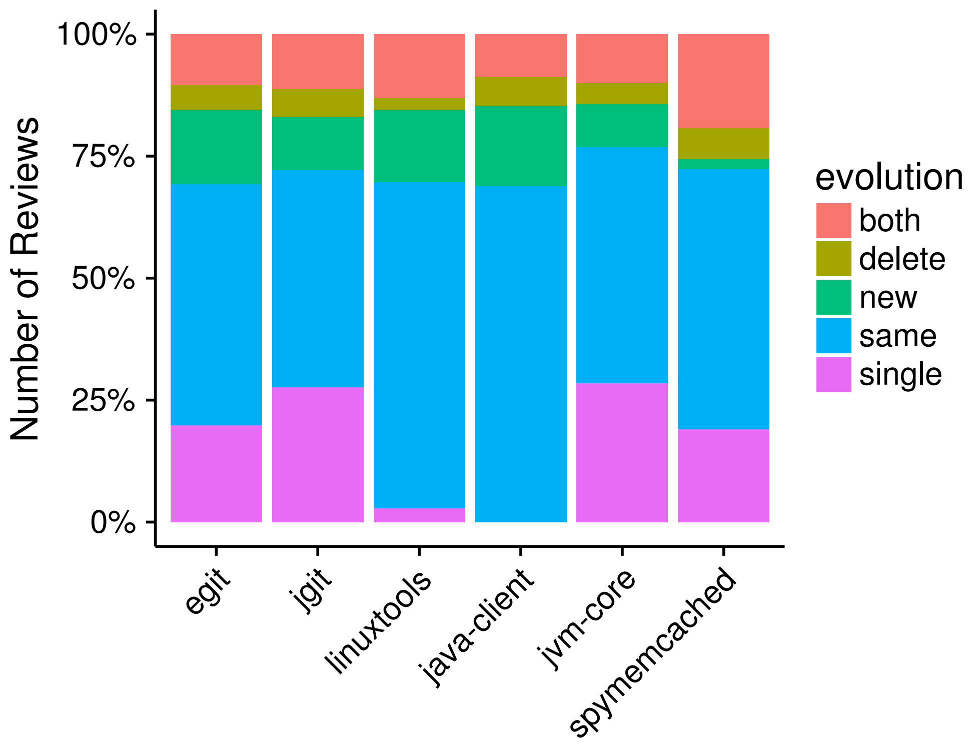


Figure 6.2: Distribution of refactoring evolution patterns grouped by system.

above for each system under study. As one can see, for all systems considered in this investigation, around 75% of the reviews present either a *same* or *single* refactoring evolution pattern. This indicates that only one out of four code reviews exhibit any change in its refactoring operations during the reviewing process. For the 25% of reviews in which we observe a refactoring evolution, *new* refactorings appear to be the most common pattern. Differently, sole deletions of refactorings during code review is the least observable evolution pattern, where most of the deletions are associated with additions as seen by the number of reviews in which we observe the *both* pattern.

Figure 6.3 combines the refactoring evolution observations for all systems and presents them grouped by different intents. As previously observed, most code reviews do not exhibit changes in the refactoring operations performed during the reviewing process. However, slight differences can be observed between different intents. Feature-related reviews, for example, tend to present the highest rate in refactoring change during code review, where around 30% of reviews present a *new, deleted* or *both* patterns. Differently, reviews with an intent of refactoring tend to largely remain the same throughout reviewing, with only 23% of the reviews presenting any change in the refactoring operations.

This adds evidence to an observation we made in Chapter 5 in which we noticed that reviews with a refactoring intent tend to follow a 'one and done' behaviour, where most of them are integrated as they are proposed. Reviews that add or enhance features, on the other hand, tend to be iterative, where changes and adaptations of the code change are commonly observed. Bug

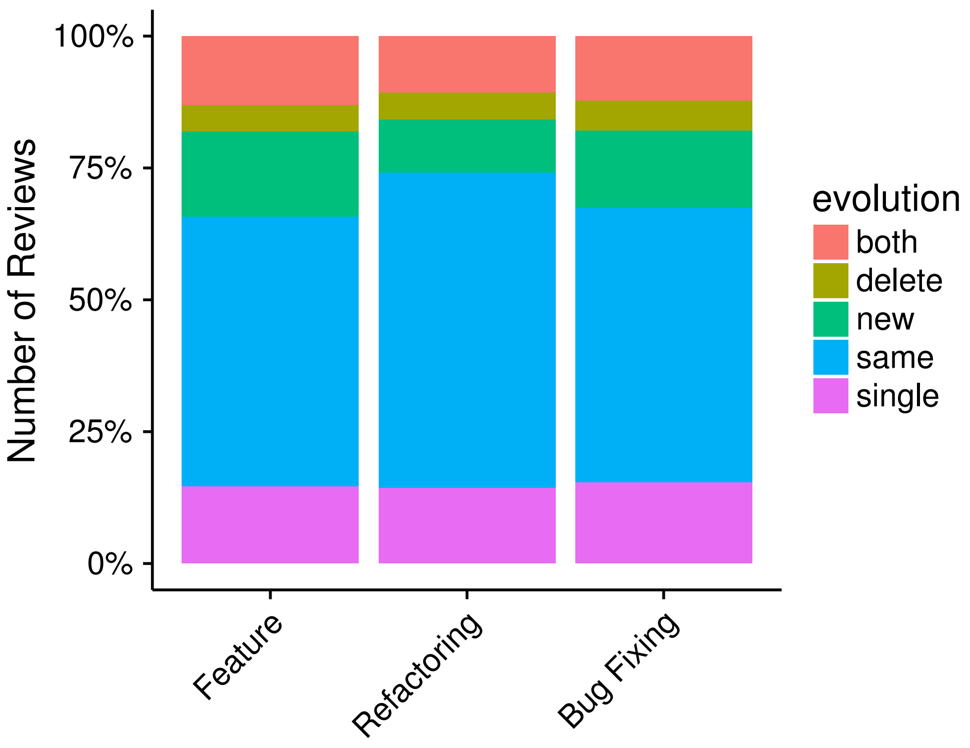


Figure 6.3: Distribution of refactoring evolution patterns grouped by intent.

fixing reviews tend to behave similarly to feature-related ones but with a small difference in the *single* pattern. Reviews with a single revision appear to be more frequent on bug fixing reviews in comparison to feature reviews.

Finally, we investigate which refactoring operations are more likely to change and evolve during code review. To do this, we perform an analysis of subsequent revisions similar to the one described previously. However, for each pair of revisions, we extract the number and type of refactoring operations that were either added or removed between revisions.

Hence, for the five more popular refactoring operations as identified in RQ2, Figure 6.4 presents the number of refactorings that were added and removed during code review for all systems and all intents considered in this investigation. As one can see in the figure, extract and rename method refactorings tend to be more often added during code review than removed. This observation was expected since these refactorings are mostly used in the implementation and enhancement of features, which represent the most iterative code reviews, as previously discussed.

Differently, extract superclass, inline method, and move class tend to be equally added and removed during code review. In an interesting observation, the number of times move class refactorings are added and removed during code review is considerably greater than extract superclass and inline method, even though the overall number of move class operations is smaller than the two others. This indicates that move class operations are more volatile, and developers tend to be more careful when integrating reviews that perform this type of refactoring.

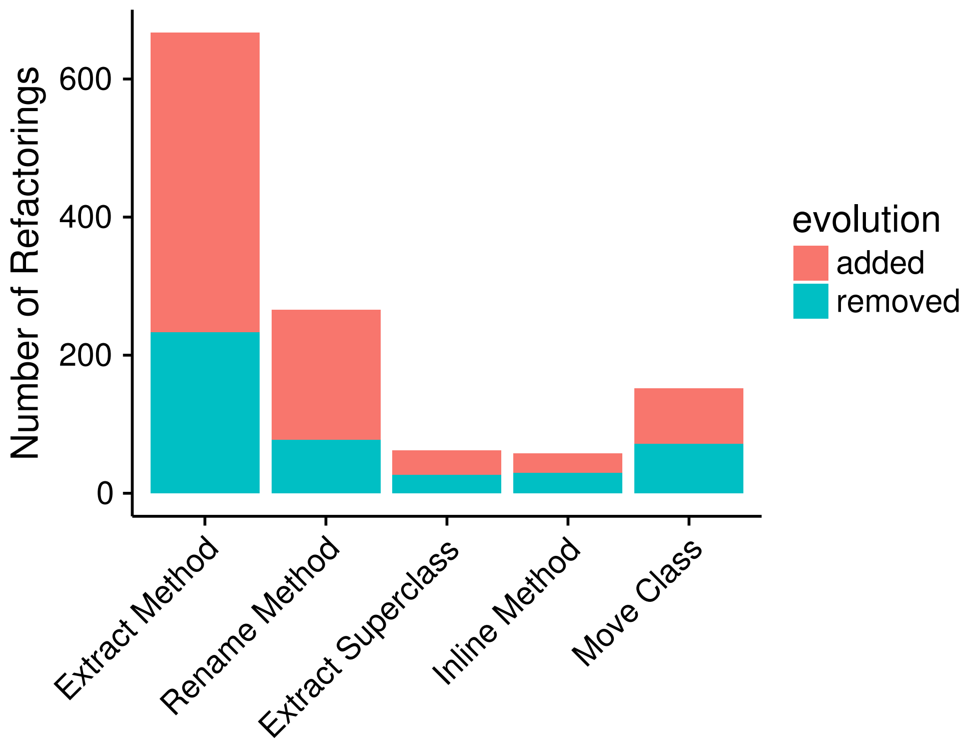


Figure 6.4: Distribution of added and removed refactoring operations during code review.

As an answer to RQ4, around 75% of refactoring operations are not altered during code review. This observation holds for all systems and intents considered in this investigation. For the reviews that do change refactoring operations during code review, developers tend to more often add refactorings than remove them. Finally, different refactoring operations tend to be equally added and removed during code review. However, extract and move method, in particular, tend to be more added than removed, which indicates the iterative nature of feature-related changes.

6.4 Threats to the Validity

Internal validity: We employ RefMiner (Tsantalis et al., 2018) to detect the refactoring operations performed during code review. The current version of the tool is only able to identify 15 refactoring operations even though some refactoring catalogues report more than 70 refactoring types. Thus, it is possible that some of the reviews we considered not to perform any refactoring actually employed operations that are not supported by RefMiner. However, the 15 refactoring operations detected by RefMiner represent the most popular refactoring operations as evaluated by a series of empirical studies (Murphy-Hill et al., 2012; Negara et al., 2013; Silva et al., 2016). Hence, we expect the number of non-identified refactoring operations to be minimal.

Automated refactoring detection is not an easy task, especially in the presence of *floss* refactorings. Hence, RefMiner might report false negatives and fail to report true positives. In its original publication (Tsantalis et al., 2018), RefMiner was found to achieve an overall precision and recall of 97.96% and 87.20%, respectively. Thus, we consider RefMiner to be acceptable for the empirical investigation we performed.

Manual classifications are naturally subjective to bias. To mitigate this threat, we employed a two-stage manual classification procedure. In the first stage, all reviews were separately classified by two researchers. In the second stage, for all reviews in which a disagreement was found, both researchers 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. Refactoring operations are heavily attached to the programming language and paradigm in which these are performed. Thus, the results and observations reported in this work may not be generalisable for systems developed in other languages.

The identification of refactoring operations was performed by RefMiner. One might expect different results when employing different tools for automated refactoring detection, such as RefDiff (Silva and Valente, 2017) or RefFinder (Kim et al., 2010). However, RefMiner was evaluated as the tool with the highest precision and recall when compared to previous refactoring detectors.

6.5 Related Work

The first grasps on the developers' intentions and motivations when performing refactoring emerged in the studies by Murphy-Hill et al. (2012) and Negara et al. (2013), in which the authors investigated the differences between manual and automated refactoring. These papers observed that refactorings are commonly performed in conjunction with other types of changes, mostly as preparation for the introduction and/or enhancement of features and bug fixing. This phenomenon has become known as *floss* refactoring. These findings indicated that we need to better understand how developers approach refactoring so that we can better support such activity.

Silva et al. (2016) monitored Github projects to detect refactorings performed in 124 software systems. In the following of these changes being incorporated into the systems, the authors performed email interviews and surveys with the authors of the changes to assess their motivations behind the refactorings. Extracting a method was mentioned as the most common refactoring operation, where the main motivation is the preparation for new feature developments. Our study investigated the developers' intentions behind refactoring operations in the context of code review, thereby avoiding any biases that may be incurred by interviews and surveys.

In a different study, Palomba et al. (2018) collected a large dataset of software changes that employed refactoring operations and identified the developers' intents behind each change. An odds ratio analysis was then performed to assess the likelihood of certain refactoring operations to be used in feature-related, bug fixing or maintenance tasks. The classification of a software change into one of the three intents was performed automatically by employing a text-based analysis that originally scored an approximate precision of 61%. Our classification of the developers' intents behind each code review was performed manually, where two researchers assessed each review separately and resolved any conflicts at a later stage.

In Chapter 5, we also performed an investigation on the developers' intents behind software changes during code review. The authors automatically identified significant architectural changes and investigated the context in which such changes were performed. A similar investigation was carried on the context of code smells. Tufano et al. (2017) identified the developers' intents behind commits that introduced and removed code smells in order to better understand how smells are created and fixed in software systems.

6.6 Conclusion

Software refactoring is the activity of modifying software with the goal of improving structure while maintaining its behaviour and functionality. Refactoring has been associated with improvements in software quality that include, but are not limited to, maintainability, understandability, and reusability. However, recent empirical investigations have shown that developers commonly face and perform refactoring operations differently than what it was previously believed in research. Thus, this investigation has performed a study aimed at better understanding the context and motivations in which developers perform software refactoring.

To do this, we used code review data and automatically identified occasions in which developers performed refactoring operations during code review. We followed this automated procedure with a manual analysis of all code reviews that performed refactoring operations. The manual analysis and classification consisted of two researchers identifying the developers' intents behind each change, such as the development of new features, bug fixing, and refactoring.

Our data shows that refactoring operations are most often used in code reviews that implement new features or enhance existing features, accounting for 63% of the code changes we studied. Only in 31% of the code reviews that employed refactoring operations the developers had the actual intent of refactoring the system. Such observations indicate that developers often prefer to mix the refactorings with other changes instead of submitting a code review that only performs refactoring.

We noticed that extract method is the most common refactoring operation employed for software changes when considering all different intents. However, for reviews in which the developers had the intention of refactoring, the usage of different refactoring operations is more evenly distributed. Moreover, we observed that some refactoring operations, such as move class, are more often employed when developers have the intent of refactoring the system.

Interestingly, we noticed a non-negligible number of code reviews in which the developers have the intention of refactoring but do not employ refactoring operations. After a careful analysis of each particular case, we observed that this phenomenon mostly occurs in reviews that clean up dead code, which is an operation that is not currently supported by the tool we used to detect refactorings. For some other reviews, the refactorings were deeply tangled with other changes, so that the tool could not identify the refactoring operations.

Finally, we investigated how refactoring operations evolve during code review. We noticed that around 75% of refactoring operations remain unchanged during the entire review process for all systems under study. We observed a similar result when grouping the code reviews by different intents. However, feature-related changes tend to present a higher rate of refactoring evolution while refactoring-related changes presented a lower rate of refactoring evolution.

6.7 Conclusions From the Chapter

In this chapter, we investigated low-level software restructuring through the analysis of refactoring operations during code review. We observed that developers most often perform refactoring operations when implementing new features or enhancing existing features. Surprisingly, we noticed cases in which developers had the intention of refactoring but did not employ any refactoring operation detected by automated tools. Finally, our data suggests that extract method is the refactoring operation that is mostly performed in the course of code review.

The findings have extended the empirical knowledge on how developers perform and perceive refactorings. By using a new methodology of code review analysis, we complemented previous observations on the concepts of *floss* refactorings and the intents behind refactoring operations. Moreover, we provided new insights on occasions in which developers have the intent of refactoring the system but do not employ any refactoring operation. Finally, we performed the first investigation on the evolution of refactoring operations during code review.

7 Conclusion and Future Work

The complexity of software systems increases as the systems evolve. As architectural debt is built into the system, maintenance effort and bug-proneness tend to increase. The accumulation of architectural debt leads to architectural degradation, which in turn might require complete re-implementations of software systems. Although automated approaches to assist developers in restructuring a software system have been proposed, there is no evidence of widespread adoption by practitioners. Moreover, these approaches are not integrated and evaluated within the current models and practices adopted by software engineering practitioners.

Software code review is a widespread practice in today's software development industry. The lightweight and asynchronous process of modern code review allows developers to peer review changes performed by other developers in the system and provide feedback. Code review has been shown to facilitate software comprehension and code quality. In addition, there is empirical evidence to suggest that the process of code review is able to identify bugs related to maintainability and evolution.

In this context, this thesis set out with the goal to *study software restructuring at different granularity levels to better understand how software developers perform restructuring on their daily basis*. Hence, this thesis presented a series of empirical study, where each of which tackled a different objective to reach this goal.

7.1 Summary of Contributions

Chapter 3 aimed at investigating automated approaches for architectural improvement in the context of software evolution. By studying official longitudinal releases of software systems, we observed that developers hardly perform large-scale restructurings. Hence, although the architectural structure implemented by the systems' developers tend to largely respect common metrics used by automated restructuring techniques, our data suggests that such techniques would cause a large disruption to the systems' original structure; thereby undermining their applicability. Thus, as one of the contributions in this chapter, we proposed a new state-of-the-art technique for search-based modularisation that searches for compromises between structural improvement and disruption.

As previously mentioned, the code review process is widely adopted by practitioners. Thus, Chapter 5 carried on a study in which we investigated how developers perform architectural changes on their daily basis. We observed that most of the software changes that significantly impact the systems' structure have the intent of implementing a new feature and/or enhancing an existing feature. Moreover, our data suggests that developers do not often discuss the systems' architecture during code review for changes that impact the structure. However, for the code reviews in which we found evidence of architectural discussion, we noticed the architectural improvements tend to be more pronounced than in reviews where developers do not discuss the architecture. These observations broadened the empirical knowledge of how developers perceive and perform architectural changes during code review, suggesting that developers will greatly benefit from tools and approaches that automatically made them aware of the architectural impact of their changes.

Finally, Chapter 6 tackled the last objective for this thesis, which was the study of how developers employ refactoring operations on their daily basis. By using a new methodology for the study of the developers' intents behind refactoring, we complemented existing studies in showing that most refactoring operations are performed in conjunction with other changes, i.e., *floss* refactoring. Moreover, we provided new insights on software changes in which developers have the intention of refactoring the system but do not employ any refactoring operation. Finally, we observed how refactoring operations tend to evolve during code review. These findings enhanced the empirical knowledge on the context and conditions in which developers perform refactoring operations. Such new insights should be taken into account by both academic and industrial practitioners in the development and/or enhancement of tools for automated refactoring support.

Thus, through the study of software restructuring at different levels of granularity, we extended the state-of-the-art on the understanding of how developers perform restructuring activities in real-world software systems. The observations and findings reported in this thesis will serve as a groundwork for the development and improvements of automated tools to assist developers when performing software restructuring in both large-scale architectural changes and low-level refactorings.

7.2 Summary of Future Work

Many aspects of the research conducted in this thesis point to new research directions and extensions. Some of these are discussed next.

Automatic classification of a review's intent

In chapters 5 and 6 we manually analysed data from code review to identify the developers intent behind software changes. Although our methodology follows the best principles of such analysis, this method does not scale for a study of larger proportions. Hence, the automatic classification of the intent of a software change would greatly enhance empirical studies such as the ones presented in this thesis.

Integration of search-based software modularisation into code review

In chapter 3 we proposed an approach based on search-based modularisation to suggest solutions that improve the structural architecture of the system while preserving its original structure. In the following chapters we presented several insights on the needs and behaviours of developers when performing architectural changes and refactoring. All this knowledge could be used to integrate the proposed approach in a tool to be used by practitioners during the code review process.

Usage of different metrics of cohesion and coupling

In chapters 3 and 5 we employed structural measurements of cohesion and coupling. However, other metrics have been proposed in the literature, such as semantical and historical cohesion/-coupling. Hence, future studies may build upon the methodology and data we make available in this thesis to extend such studies by incorporating different metrics of cohesion and coupling.

Bibliography

H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui. Automatic Package Coupling and Cycle Minimization. In *2009 16th Working Conference on Reverse Engineering*, pages 103–112. IEEE, 2009.

H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, and S. Ducasse. Towards automatically improving package structure while respecting original design decisions. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 212–221. IEEE, oct 2013.

A. Ackerman, L. Buchwald, and F. Lewski. Software inspections: an effective verification process. *IEEE Software*, 6(3):31–36, may 1989.

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*, pages 1–35, nov 2017. ISSN 1382-3256.

J. Al Dallal and L. C. Briand. An object-oriented high-level design-based class cohesion metric. *Information and Software Technology*, 52(12):1346–1361, dec 2010. ISSN 09505849.

M. Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and software technology*, 51(9):1319–1326, 2009.

E. Ammerlaan, W. Veninga, and A. Zaidman. Old habits die hard: Why refactoring for understandability does not give immediate benefits. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 504–507. IEEE, 2015.

N. Anquetil and T. Lethbridge. Experiments with clustering as a software remodularization method. In *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, pages 235–255. IEEE Comput. Soc, 1999.

A. Arcuri and L. Briand. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3): 219–250, may 2014.

A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE, may 2013.

V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *35th International Conference on Software Engineering (ICSE)*, may 2013.

J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

M. Barros and F. Farzat. What Can a Big Program Teach Us about Optimization? In *Proceeding of the International Symposium on Search Based Software Engineering (SSBSE'13)*, pages 275–281. 2013.

M. d. O. Barros. An analysis of the effects of composite objectives in multiobjective software module clustering. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, page 1205, New York, New York, USA, 2012. ACM Press.

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*, 57(1):684–704, jan 2015.

G. Bavota, F. Carnevale, A. De Lucia, M. Di Penta, and R. Oliveto. Putting the Developer in-the-Loop: An Interactive GA for Software Re-modularization. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7515 LNCS, pages 75–89. 2012.

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 *2013 35th International Conference on Software Engineering (ICSE)*, pages 692–701, San Francisco, may 2013. IEEE.

G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology*, 23(1):1–33, feb 2014.

G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14, 2015.

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*, page 354, New York, New York, USA, 2011. ACM Press.

F. Beck and S. Diehl. On the impact of software evolution on software clustering. *Empirical Software Engineering*, 18(5):970–1004, oct 2013.

M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, pages 202–211, New York, New York, USA, 2014. ACM Press.

J. Bieman and L. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, 1994.

J. M. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of the 1995 Symposium on Software reusability - SSR '95*, pages 259–262, New York, New York, USA, 1995. ACM Press. ISBN 0897917391.

C. Bonja and E. Kidanmariam. Metrics for class cohesion and similarity between methods. In *Proceedings of the 44th annual southeast regional conference on - ACM-SE 44*, page 91, New York, New York, USA, 2006. ACM Press. ISBN 1595933158.

A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *International Symposium on Foundations of Software Engineering*, 2014.

L. Briand, J. Daly, and J. Wust. A unified framework for cohesion measurement in object-oriented systems. In *Proceedings Fourth International Software Metrics Symposium*, volume 3, pages 43–53. IEEE Comput. Soc, 1998.

L. Briand, J. Daly, and J. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

N. Brown, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, N. Zazworka, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, and R. Nord. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*, page 47, New York, New York, USA, 2010. ACM Press.

W. H. Brown, R. C. Malveau, and T. J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley, 1998.

I. Candela, G. Bavota, B. Russo, and R. Oliveto. Using Cohesion and Coupling for Software Remodularization : Is It Enough ? *ACM Transactions on Software Engineering and Methodology*, 25(3):1–28, 2016.

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*, pages 465–475, New York, New York, USA, 2017. ACM Press.

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

J. Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2nd edition, 1988.

Couchbase. Couchbase's Open Source Community. `https://developer.couchbase.com/open-source-projects`, 2018. Accessed in: September 2018.

S. Counsell, S. Swift, and A. Tucker. Object-oriented cohesion as a surrogate of software comprehension: an empirical study. In *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 161–172. IEEE, 2005.

W. Cunningham. The WyCash portfolio management system. In *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92) (Addendum)*, volume 4, pages 29–30, New York, New York, USA, apr 1993. ACM Press.

L. de Silva and D. Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, jan 2012.

J. T. de Souza, C. L. Maia, F. G. de Freitas, and D. P. Coutinho. The Human Competitiveness of Search Based Software Engineering. In *2nd International Symposium on Search Based Software Engineering*, pages 143–152. IEEE, sep 2010.

K. Deb. Multi-objective optimization. In *Search methodologies*, pages 403–449. Springer, 2014.

K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, apr 2002.

D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen. Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering*, 34(3): 321–335, 2008.

D. Doval, S. Mancoridis, and B. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *STEP '99. Proceedings Ninth International Workshop Software Technology and Engineering Practice*, pages 73–81. IEEE Comput. Soc, 1999.

M. Duchrow. Programmer's friend class dependency analyzer. `http://www.dependency-analyzer.org/`, 2018. Accessed in: September 2018.

Eclipse. Eclipse's Open Source Community. `https://eclipse.org/projects`, 2018. Accessed in: September 2018.

S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1): 1–12, 2001.

K. El Emam, S. Benlarbi, N. Goel, and S. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7): 630–650, jul 2001.

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 2015*, pages 50–60, New York, New York, USA, 2015. ACM Press.

M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

L. Fernández and R. Peña. A Sensitive Metric of Class Cohesion. *Information Theories and Applications*, 13:82–91, 2006.

M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1037–1039. ACM, 2011.

S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, pages 222–232. IEEE Press, 2012.

M. Fowler. Technicaldebtquadrant. `http://martinfowler.com/bliki/TechnicalDebtQuadrant.html`, 2009. Accessed in September, 2018.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

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)*, pages 552–555. IEEE, nov 2011.

J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic. Obtaining ground-truth software architectures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 901–910. IEEE, may 2013.

D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in software engineering and knowledge engineering*, pages 1–39. World Scientific, 1993.

M. W. Godfrey and E. H. Lee. Secrets from the monster: Extracting mozilla's software architecture. In *Proceedings of Second Symposium on Constructing Software Engineering Tools (CoSET'00)*, 2000.

G. Gousios and A. Zaidman. A dataset for pull-based development research. In *Working Conference on Mining Software Repositories*, MSR, 2014.

M. Hall and P. McMinn. An analysis of the performance of the bunch modularisation algorithm's hierarchy generation approach. In *4 th Symposium on Search Based-Software Engineering*, page 19, 2012.

M. Hall, N. Walkinshaw, and P. McMinn. Supervised software modularisation. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 472–481. IEEE, sep 2012.

M. Hall, M. A. Khojaye, N. Walkinshaw, and P. McMinn. Establishing the Source Code Disruption Caused by Automated Remodularisation Tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 466–470. IEEE, sep 2014.

K. Hamasaki, R. G. Kula, N. Yoshida, A. E. C. Cruz, K. Fujiwara, and H. Iida. Who does what during a code review? datasets of oss peer review repositories. In *Working Conference on Mining Software Repositories*, MSR, 2013.

M. Harman, R. Hierons, and M. Proctor. A New Representation And Crossover Operator For Search-Based Optimization Of Software Modularization. In *Genetic and Evolutionary Computation Conference*, 2002.

M. Harman, S. Swift, and K. Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, page 1029, New York, New York, USA, 2005. ACM Press.

M. Harman, P. McMinn, J. T. de Souza, and S. Yoo. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In B. Meyer and M. Nordio, editors, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7007 of *Lecture Notes in Computer Science*, pages 1–59. Springer Berlin Heidelberg, 2012.

J. Henkel and A. Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th international conference on Software engineering*, pages 274–283. ACM, 2005.

M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*, pages 25–27, 1995.

J. Huang and J. Liu. A similarity-based modularization quality measure for software module clustering problems. *Information Sciences*, 342:96–110, may 2016.

J. Huang, J. Liu, and X. Yao. A multi-agent evolutionary algorithm for software module clustering problems. *Soft Computing*, feb 2016.

K. Jeet and R. Dhir. Software Architecture Recovery using Genetic Black Hole Algorithm. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–5, feb 2015.

K. Jeet and R. Dhir. Software module clustering using bio-inspired algorithms. *Handbook of Research on Modern Optimization Algorithms and Applications in Engineering and Economics*, page 445, 2016.

Y. Jiang, B. Adams, and D. M. German. Will my patch make it? and how fast? case study on the linux kernel. In *Working Conference on Mining Software Repositories*, MSR, May 2013.

C. Kapser and M. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *2006 13th Working Conference on Reverse Engineering*, pages 19–28. IEEE, 2006.

M. G. Kendall. *Rank correlation methods*. Griffin, 1948.

N. Kennedy. Google mondrian: web-based code review and storage. `https://www.niallkennedy.com/blog/2006/11/google-mondrian.html`, 2006. Accessed in September, 2018.

M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 371–372. ACM, 2010.

M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160. ACM, 2011.

B. Kitchenham, S. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, 1995. ISSN 00985589.

P. Kruchten, R. L. Nord, and I. Ozkaya. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6):18–21, nov 2012.

P. B. Kruchten. The 4+ 1 view model of architecture. *IEEE software*, 12(6):42–50, 1995.

A. C. Kumari and K. Srinivas. Hyper-heuristic approach for multi-objective software module clustering. *Journal of Systems and Software*, 117:384–401, jul 2016.

A. C. Kumari, K. Srinivas, and M. P. Gupta. Software module clustering using a hyper-heuristic based multi-objective genetic algorithm. In *Proceedings of the 2013 3rd IEEE International Advance Computing Conference, IACC 2013*, pages 813–818, Ghaziabad, 2013. IEEE.

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.

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 *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, volume 2015-Augus, pages 235–245. IEEE, may 2015.

M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1(C):213–221, jan 1979.

M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*, pages 20–32. IEEE Comput. Soc, 1997.

M. M. Lehman. Laws of software evolution revisited. In C. Montangero, editor, *Software Process Technology*, pages 108–124, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70676-2.

W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, nov 1993.

Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

K. Mahdavi, M. Harman, and R. Hierons. A multiple hill climbing approach to software module clustering. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 315–324. IEEE Comput. Soc, 2003.

A. S. Mamaghani and M. R. Meybodi. Clustering of Software Systems Using New Hybrid Algorithms. In *2009 Ninth IEEE International Conference on Computer and Information Technology*, volume 1, pages 20–25. IEEE, 2009.

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. 6th oInternational Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, number for C, pages 45–52. IEEE Comput. Soc, 1998.

S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 50–59. IEEE, 1999.

M. Mantyla and C. Lassenius. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, may 2009.

A. Martini, J. Bosch, and M. Chaudron. Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology*, 67: 237–253, nov 2015.

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 *Working Conference on Mining Software Repositories*, MSR, 2014.

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*, pages 744–753. IEEE Comput. Soc, 2001.

B. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, mar 2006.

B. Mitchell, M. Traverso, and S. Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, pages 181–190. IEEE Comput. Soc, 2001.

B. S. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem.* PhD thesis, Drexel University, 2002.

B. S. Mitchell and S. Mancoridis. Using Heuristic Search Techniques To Extract Design Abstractions From Source Code. *Genetic and Evolutionary Computation Conference*, pages 1375–1382, 2002.

B. S. Mitchell and S. Mancoridis. Modeling the Search Landscape of Metaheuristic Software Clustering Algorithms. In *Genetic and Evolutionary Computation Conference*, pages 2499–2510, Chicado, IL, USA, 2003. Springer Berlin Heidelberg.

B. S. Mitchell and S. Mancoridis. On the evaluation of the Bunch search-based software modularization algorithm. *Soft Computing*, 12(1):77–93, aug 2007.

W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni. Many-Objective Software Remodularization Using NSGA-III. *ACM Transactions on Software Engineering and Methodology*, 24(3):1–45, may 2015.

R. Mo, Y. Cai, R. Kazman, and L. Xiao. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 51–60. IEEE, may 2015.

R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. Decoupling level. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, pages 499–510, New York, New York, USA, 2016. ACM Press.

A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *icsm*, pages 120–130, 2000.

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 *International Conference on Software Analysis, Evolution, and Reengineering*, 2015.

R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *Balancing Agility and Formalism in Software Engineering*, pages 252–266. Springer, 2008.

M. Mukadam, C. Bird, and P. C. Rigby. Gerrit software code review data from android. In *Working Conference on Mining Software Repositories*, 2013.

E. Murphy-Hill, C. Parnin, and A. P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, jan 2012.

S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *European Conference on Object-Oriented Programming*, pages 552–576. Springer, 2013.

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 ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12*, page 49, New York, New York, USA, 2012. ACM Press.

A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering*, 20(1):47–79, mar 2013.

A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb. Multi-Criteria Code Refactoring Using Search-Based Software Engineering. *ACM Transactions on Software Engineering and Methodology*, 25(3):1–53, jun 2016.

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*, pages 1–21, 2017a.

M. Paixao, M. Harman, Y. Zhang, and Y. Yu. Supporting material for the paper: An empirical study of cohesion and coupling: Balancing optimisation and disruption. `https://mhepaixao.github.io/balancing_improvement_disruption/`, 2017b. Accessed in September, 2018.

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*, 2017c.

F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia. An Exploratory Study on the Relationship between Changes and Refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 176–185. IEEE, may 2017.

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*, 99(September 2017):1–10, jul 2018.

D. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, mar 1979.

D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, dec 1972.

D. L. Parnas, P. C. Clements, and D. M. Weiss. The Modular Structure of Complex Systems. In *International Conference on Software Engineering*, 1984.

S. Pearce. Gerrit code review for git. `https://www.gerritcodereview.com`, 2006. Accessed in: September 2018.

J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *17th European Conference on Genetic Programming*, volume 8599, pages 137–149. 2014.

D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, feb 2009.

K. Praditwong. Solving software module clustering problem by evolutionary algorithms. In *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 154–159. IEEE, may 2011.

K. Praditwong and X. Yao. A New Multi-objective Evolutionary Optimisation Algorithm: The Two-Archive Algorithm. In *2006 International Conference on Computational Intelligence and Security*, pages 286–291. IEEE, nov 2006.

K. Praditwong, M. Harman, and X. Yao. Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering*, 37(2):264–282, mar 2011.

R. S. Pressman. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

ReviewBoard. Reviewboard. `https://www.reviewboard.org/`, 2017. Accessed in September, 2018.

P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 202, New York, New York, USA, 2013. ACM Press.

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

P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, apr 2009.

A. M. Saeidi, J. Hage, R. Khadka, and S. Jansen. A search-based approach to multi-view clustering of software systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 429–438. IEEE, mar 2015.

V. Sales, R. Terra, L. F. Miranda, and M. T. Valente. Recommending move method refactorings using dependency sets. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 232–241. IEEE, 2013.

R. Schwanke, L. Xiao, and Y. Cai. Measuring architecture quality by structure plus history analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 891–900. IEEE, may 2013.

Scitools. Scitools. `https://scitools.com/features`, 2018. Accessed in: September 2018.

O. Seng, M. Bauer, M. Biehl, and G. Pache. Search-based improvement of subsystem decompositions. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, page 1045, New York, New York, USA, 2005. ACM Press.

A. Shahbazian, D. Nam, and N. Medvidovic. Toward predicting architectural significance of implementation issues. In *International Conference on Mining Software Repositories*, MSR, 2018.

A. Shokoufandeh, S. Mancoridis, T. Denton, and M. Maycock. Spectral and meta-heuristic algorithms for software clustering. *Journal of Systems and Software*, 77(3):213–223, sep 2005.

D. Silva and M. T. Valente. Refdiff: detecting refactorings in version histories. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 269–279. IEEE, 2017.

D. Silva, R. Terra, and M. T. Valente. Recommending automated extract method refactorings. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 146–156. ACM, 2014.

D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 858–870, New York, New York, USA, 2016. ACM Press.

C. Simons and J. Smith. Exploiting antipheromone in ant colony optimisation for interactive search-based software design and refactoring. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 143–144. ACM, 2016.

C. Simons, J. Singer, and D. R. White. Search-Based Refactoring: Metrics Are Not Enough. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9275, pages 47–61. 2015.

H. Siy and L. Votta. Does the modern code inspection have value? In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 281–289. IEEE Comput. Soc, 2001.

I. Sommerville. *Software Engineering*. Addison Wesley, 2011.

I. Sutton. Software erosion in pictures – findbugs. `http://structure101.com/blog/2008/11/software-erosion-findbugs/`, 2008. Accessed in September, 2018.

R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology*, 54(12):1297–1307, 2012.

N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331. IEEE, 2008.

N. Tsantalis, M. Mansouri, L. M-Eshkevari, D. Mazinanian, and D. Dig. Accurate and Efficient Refactoring Detection in Commit History. *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, page to appear, 2018.

A. Tsotsis. Meet phabricator, the witty code review tool built inside facebook. `https://techcrunch.com/2011/08/07/oh-what-noble-scribe-hath-penned-these-words/`, 2011. Accessed in September, 2018.

A. Tucker, S. Swift, and X. Liu. Variable grouping in multivariate time series via correlation. *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, 31(2):235–245, apr 2001.

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). *IEEE Transactions on Software Engineering*, 1(c):1–1, may 2017.

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

V. Tzerpos and R. Holt. MoJo: a distance metric for software clusterings. In *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, pages 187–193. IEEE Comput. Soc, 1999.

V. Tzerpos and R. Holt. ACCD: an algorithm for comprehension-driven clustering. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 258–267. IEEE Comput. Soc, 2000.

J. van Gurp and J. Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, mar 2002.

J. van Gurp, S. Brinkkemper, and J. Bosch. Design preservation over subsequent releases of a software product: a case study of Baan ERP. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):277–306, jul 2005.

L. G. Votta. Does every inspection need a meeting? In *Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering - SIGSOFT '93*, volume 35, pages 107–114, New York, New York, USA, 1993. ACM Press.

P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In *International Workshop on Mining Software Repositories (MSR)*, 2008.

Z. Wen and V. Tzerpos. An optimal algorithm for MoJo distance. In *Proceedings of 2003 IEEE International Workshop in Program Comprehension*. IEEE Comput. Soc, 2003.

Z. Wen and V. Tzerpos. Evaluating similarity measures for software decompositions. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 368–377. IEEE, 2004a.

Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 194–203. IEEE, 2004b.

M. Wermelinger, Y. Yu, A. Lozano, and A. Capiluppi. Assessing architectural evolution: a case study. *Empirical Software Engineering*, 16(5):623–666, oct 2011.

B. J. Williams and J. C. Carver. Characterizing software architecture changes: A systematic review. *Information and Software Technology*, 52(1):31–51, jan 2010.

J. Wu, A. Hassan, and R. Holt. Comparison of clustering algorithms in the context of software evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, volume 2005, pages 525–535. IEEE, 2005.

X. Xia, D. Lo, X. Wang, and X. Yang. Who should review this change? putting text and file location analyses together for more accurate recommendations. In *International Conference on Software Maintenance and Evolution (ICSME)*, 2015.

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*, pages 488–498, New York, New York, USA, 2016. ACM Press.

X. Yang, R. G. Kula, N. Yoshida, and H. Iida. Mining the modern code review repositories: A dataset of people, process and product. In *Working Conference on Mining Software Repositories (MSR)*, 2016.

E. Yourdon and L. L. Constantine. *Structured design: Fundamentals of a discipline of computer program and systems design*, volume 5. Prentice-Hall Englewood Cliffs, NJ, 1979.

M. B. Zanjani, H. Kagdi, and C. Bird. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering*, 42(6), June 2016.

Zhifeng Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, volume 2001-Janua, pages 293–299. IEEE Comput. Soc, 2001.

Zhihua Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, pages 194–203. IEEE, 2004.

Y. Zhou and H. Leung. Predicting object-oriented software maintainability using multivariate adaptive regression splines. *Journal of Systems and Software*, 80(8):1349–1361, aug 2007.

T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, jun 2005. ISSN 0098-5589.