AN EMPIRICAL STUDY OF THE INTERPLAY BETWEEN ARCHITECTURE AND SOFTWARE QUALITY USING EVOLUTIONARY HISTORY OF SOFTWARE

by

Ehsan Kouroshfar
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
the Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

_____	Dr. Paul Ammann, Dissertation Director
_____	Dr. Sam Malek, External Committee Member
_____	Dr. Jeff Offutt, Committee Member
_____	Dr. Thomas LaToza, Committee Member
_____	Dr. Houman Homayoun, Committee Member
_____	Dr. Sanjeev Setia, Department Chair
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Spring Semester 2016
George Mason University
Fairfax, VA

An Empirical Study of the Interplay between Architecture and Software Quality using Evolutionary History of Software

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Ehsan Kouroshfar
Master of Science
Sharif University of Technology, 2009
Bachelor of Science
Amirkabir University of Technology, 2006

Director: Paul Ammann, Associate Professor
Department of Computer Science

Spring Semester 2016
George Mason University
Fairfax, VA

Dedication

To my beloved parents, Nahid, and Kourosh.

Acknowledgments

It is a pleasure to thank those who made this dissertation possible through their support, guidance, encouragement and inspiration.

My deepest gratitude is to my adviser Dr. Sam Malek. Not only has Sam nurtured my technical and communication skills, he also supported me as a friend in tough times. It would have not been possible to finish this dissertation without his patience and continuous support.

My special appreciation goes to Dr. Paul Ammann for administering the role of dissertation director after Sam joined the University of California, Irvine. I thank Dr. Jeff Ouffut for detailed review and excellent comments on my proposal and dissertation documents. I also acknowledge the other members of my committee, Dr. Thomas Latoza and Dr. Houman Homayoun for their feedback and devoting time from their busy schedules.

I thank Dr. Joshua Garcia for his immense contributions. He helped me finish this research. I also thank my other collaborators Dr. Mehdi Mirakhorli, Dr. Hamid Bagheri, Dr. Yuanfang Cai, and Lu Xiao for their contributions.

I thank Dr. Audris Mockus for giving me the opportunity to work with him at Avaya where I could apply my research expertise in an industrial setting.

I thank my former colleagues Dr. Naeem Esfahani and Dr. Ahmed Elkhodari for their help and support when I joined Mason.

I have been fortunate to have the support of great friends at Mason. I thank Nariman Mirzaei, Pouyan Ahmadi, and Saba Neyshabouri for their support and companionship in all the ups and downs I faced during the last few years.

I am forever grateful to my parents, Nahid and Kourosh, for all their sacrifices, love, and support. Without their help, I could not have started the PhD, let alone finished it. Their presence in my life and support of my decisions has been an incredible blessing. I thank my younger brother Erfan who has supported my parents in my absence. Finally I thank my best friend, Julia, for all her love, support, and encouragement during the final years of my PhD.

Table of Contents

Page

List of Tables . . . viii
List of Figures . . . ix
Abstract . . . x
1 Introduction . . . 1
2 Research Problem . . . 6
2.1 Problem Statement . . . 6
2.2 Research Hypotheses . . . 7
3 Related Work . . . 11
3.1 Defect Prediction . . . 11
3.2 Architectural Evolution and Decay . . . 17
3.3 Architectural-Quality Metrics . . . 17
4 The Impact of Software Architecture on Defect Proneness of Software Systems . . . 20
4.1 Methodology Overview . . . 20
4.2 Obtaining Surrogates for Architectural Module View . . . 21
4.2.1 Package View . . . 22
4.2.2 Bunch View . . . 23
4.2.3 ArchDRH View . . . 26
4.2.4 LDA View . . . 28
4.2.5 ACDC View . . . 28
4.3 Measuring Effects of Co-change Dispersion . . . 30
4.3.1 Metric Definition . . . 31
4.3.2 Underlying Characteristics of the Data . . . 33
4.3.3 Analysis Method . . . 34
4.4 Executing the Analysis . . . 35
4.5 Results of the Study . . . 38
4.5.1 Results for RQ1.a . . . 38
4.5.2 Results for RQ1.b . . . 42
4.5.3 Results for RQ1.c . . . 43
4.6 Discussion . . . 45

4.6.1 Role of Architecture in Maintenance . . . 45
4.6.2 Building Better Defect Predictors . . . 47
4.6.3 Architectural Bad Smell Predictors . . . 47
4.6.4 Empirical Research . . . 48
4.7 Threats to Validity . . . 48
4.7.1 Construct Validity . . . 49
4.7.2 External Validity . . . 51
5 Architectural Decay Prediction from Evolutionary History of Software . . . 52
5.1 Prediction Model Construction . . . 52
5.1.1 Obtaining Architectural Modules . . . 53
5.1.2 Regression Analysis Selection . . . 55
5.1.3 Dependent Variables . . . 56
5.1.4 Independent Variables . . . 58
5.2 Experimental Setup . . . 60
5.2.1 Projects Studied and Data Collection . . . 61
5.2.2 Data Splitting and Evaluation Metrics . . . 63
5.3 Experimental Results . . . 66
5.3.1 Results for RQ2.a . . . 68
5.3.2 Results for RQ2.b . . . 72
5.3.3 Results for RQ2.c . . . 75
5.3.4 Results for RQ2.d . . . 76
5.4 Discussion . . . 79
5.5 Threats to Validity . . . 84
5.6 Tool . . . 85
5.6.1 Data Collection . . . 86
5.6.2 Model Construction . . . 88
6 Challenges and Suggestions for the Community . . . 97
6.1 Challenges and Limitations . . . 97
6.1.1 The Lack of the Availability of Software Architecture Information . 97
6.1.2 Tracing Defects to Architecture . . . 98
6.1.3 Obtaining Architectural Modules . . . 99
6.2 Opportunities and Suggestions . . . 100
6.2.1 Creating a Repository for Software Architecture . . . 100
6.2.2 A Comprehensive Tool Suite for Architecture Recovery . . . 101
6.2.3 Bringing Software Architecture to Software Engineers' Every Day Life 101

7 Conclusion 103
7.1 Contributions 104
7.2 Future Work 105
Bibliography 106

List of Tables

Table		Page
4.1	Studied Projects and Release Information.	36
4.2	Regression Results for Architectural Views of (a) Bunch, (b) ArchDRH, (c) ACDC, (d) High-Level Package, (e) Low-Level Package, and (f) LDA.	39
4.3	Regression Results for Hadoop and Using the Ground-Truth Architecture.	40
4.4	Correlation Coefficients Between Defects and the Metrics for Cross-Module Co-changes (CMC), Intra-Module Co-changes (IMC), and Number of Co-changed Files (NCF). (Correlations Significant at the 0.01 Level are Highlighted)	41
4.5	Regression Results for Bunch View Including Num-Cochanged-Files.	44
4.6	Regression Results Using Random Clusters.	50
5.1	Studied Projects and Release Information.	62
5.2	Prediction of CF for Packages in HBase (Version 0.92).	73
5.3	Factors Contributing to Each Model	78
5.4	Prediction of Defects and LO for Packages in Hive (Version 0.8.1)	96

List of Figures

Figure		Page
4.1	Overview of the Experimental Method.	21
4.2	Class Dependency Analyzer.	24
4.3	A Module Dependency Graph for a Compiler (Reproduced from [71] With the Approval from the Authors).	26
4.4	The Partitioned MDG for a Compiler (Reproduced from [71] With the Approval from the Authors).	27
4.5	Architectural Module View Surrogates of a System: (a) Package View, and (b) Cluster View.	32
5.1	Overview of My Approach for Architectural-Quality Metric Prediction.	53
5.2	ROC Curve for Defect Prediction.	65
5.3	Number of Architectural Modules.	67
5.4	Percentages of Existence of Architectural-Quality Metrics.	68
5.5	AUC Performance Defects.	69
5.6	Spearman Correlation for Ranking Defective Modules.	70
5.7	AUC Performance Architectural Smells.	71
5.8	Spearman Correlation Cluster Factor.	72
5.9	Percentages of Changes of Architectural Smells.	74
5.10	Percentages of Extreme Changes of Architectural Smells	76
5.11	AUC Performance for Architectural Smell Emergence.	77

Abstract

AN EMPIRICAL STUDY OF THE INTERPLAY BETWEEN ARCHITECTURE AND SOFTWARE QUALITY USING EVOLUTIONARY HISTORY OF SOFTWARE

Ehsan Kouroshfar, PhD

George Mason University, 2016

Dissertation Director: Dr. Paul Ammann

Conventional wisdom suggests that a software system's architecture has a significant impact on its evolution. Well-designed software architecture employs the principle of separation of concern to allocate different functionalities and responsibilities to different architectural elements comprising the system [42, 49] and it is easier to make changes to a software system that has a well-designed architecture. Conversely, bad architecture, manifested as architectural bad smells [42], can increase the complexity, possibly leading to poor software quality [49].

However, a software system's architecture is known to commonly undergo the phenomenon of *architectural decay* [82], where changes and design decisions are added to the system which may break the initially designed system's software architecture. Architectural decay has a negative impact on maintaining the system and results in defects and architectural problems in the system. Thus detecting and preferably avoiding decay will save considerable time and other resources from developers and stakeholders in a system.

This dissertation targets empirical research in the domain of architecture-based software maintenance. It benefits from both fields of software architecture and mining software repository. The mining software repository (MSR) field investigates the rich data in source code repositories and defect repositories to uncover interesting information about software systems. For example data in source code repositories can be linked with data in defect repositories to observe what kind of changes would result in more defects. This would help to warn practitioners and developers about risky changes based on prior changes and faults.

In this research, I first investigate the impact of software architecture on defects from evolutionary history of software. To do that, I designed an empirical study to see whether there is a difference between types of changes made to a software system from software architecture perspective. Specifically I wanted to investigate the impact of co-changes involving several architectural modules versus co-changes localized within a single module. This provided empirical evidence for the importance of considering of software architecture while making changes to a system.

Next, I construct novel models that predict the quality of an architectural element by utilizing multiple architectural views (both structural and semantic) and architectural metrics as features for prediction. Using these models, I accurately predict low architectural quality, i.e., architectural decay in software systems. Engineers can significantly benefit from determining which architectural elements will decay before that decay actually occurs. Forecasting decay allows engineers to take steps to prevent decay, such as focusing maintenance resources on the architectural elements most likely to decay.

This research underlines the importance of software architecture in the construction and maintenance of software.

Chapter 1: Introduction

Software maintenance is a set of activities associated with the modification of a software product after it has been delivered to end-users. These activities include modifications made to fix defects (corrective maintenance), modifications performed to cope with changes in the software environment (adaptive maintenance) and modifications that address new requirements or improve software quality (perfective maintenance). Maintaining large software systems is hard and expensive. Making post release changes requires not only the understanding of the part of the system that needs to be changed but also the impact of the changes to its dependencies that might be affected by the change.

Software engineers have developed numerous abstractions to deal with the complexity of implementing and maintaining software systems. One of those abstractions is software architecture, which is particularly effective for reasoning about the system's structure, its constituent elements and the relationships among them. Software architecture enables the engineers to reason about the functionality and properties of a software system without getting involved in low-level source code and implementation details.

At the outset of any large-scale software construction project is an architectural design phase. The architecture produced at this stage is often in the form of Module View [22], representing the decomposition of the software system into its implementation units, called *architectural modules*, and the dependencies among them[1]. This architecture serves as a high-level blueprint for the system's implementation and maintenance activities.

Well-designed software architecture employs the principle of separation of concern to

[1]The notion of *architectural module* should not be confused with *module* traditionally used in the literature to refer to files or classes. Here, I use the notion of module to mean architecturally significant implementation artifacts, as opposed to its typical meaning in the programming languages. Architectural modules represent the construction units (subsystems), and therefore, are also different from *software components* that represent the runtime units of computation in the *Component-Connector View* [22].

allocate different functionalities and responsibilities to different architectural elements comprising the system [42, 49]. Conventional wisdom suggests that it is easier to make changes to a software system that has a well-designed architecture. Conversely, bad architecture, manifested as architectural bad smells [42], can increase the complexity, possibly leading to poor software quality [49]. In particular, scattered functionality, a well-known architectural bad smell, increases the system's complexity by intermingling the functionality across multiple architectural modules. While, certain level of concern scattering is unavoidable due to non-functional concerns (e.g., security), a good architecture tries to minimize it as much as possible.

Monitoring the complexity of making changes to an evolving software system and measuring their effects on software quality is essential for a mature software engineering practice. It has been shown that the more scattered are the changes among a software system's implementation artifacts such as source files and classes, the higher is the complexity of making those changes, therefore the higher is the likelihood of introducing faults [48]. In addition, *co-changes* (i.e., multiple changed files committed to a repository at the same time) have shown to be good indicators of logically coupled concerns [38], which are known to correlate with the number of defects [13, 30].

The first contribution of this research is presenting an empirical method designed for investigating yet unexplored but important software engineering research questions to better understand the impact of architecturally dispersed co-changes on software qualities. Specifically, I investigated whether co-changes involving several architectural modules (cross-module co-changes) have a different impact on software quality than co-changes that are localized within a single module (inner-module co-changes). As part of this research, I contributed two new metrics to quantify the differences between cross-module and inner-module co-changes. Two insights seem to suggest that not all co-changes have the same effect. First, an architectural module supposedly deals with a limited number of concerns, and thus co-changes localized within an architectural module are likely to deal with fewer

concerns than those that crosscut the modules. Second, it is reasonable to assume in a large-scale software system, the developers are familiar with only a small subset of the modules, and thus the more crosscutting the co-changes, the more difficult it would be for the developer to fully understand the consequences of those changes on the system's behavior. Since in reality many software systems do not have a complete and updated documentation of their software architecture, my method introduces the concept of *"Surrogate Architectural Views."* Surrogate views are obtained through a set of diverse reverse engineering methods to approximate the system's architecture for this experimental study.

The second contribution of this research is constructing novel methods that predict the quality of architectural elements. In a software system's life cycle, software maintenance tends to dominate other activities in terms of time, effort, and cost. Throughout that life cycle, a major artifact that must undergo maintenance is a software system's architecture, which determines the key properties of a software system. Architectural elements abstract away unnecessary complexity (e.g., details of source-code constructs), allowing engineers to focus on higher-level design decisions. However, a software system's architecture is known to commonly undergo the phenomenon of *architectural decay* [82], where design decisions are added to and may even violate an architecture, leading to defects and other major architectural problems.

Although decay is typically treated once its detrimental effects (e.g., highly defective module or one that is highly resistant to change) are detected in a system, engineers can benefit from stemming architectural decay before such effects occur. To make such a determination, engineers must be able to predict which architectural elements are most likely to undergo decay, so that they can allocate resources to those elements in the most effective manner. Previous work has produced models for predicting only defects for packages or directories [50,94,112]. However, defects are not the only forms of architectural decay [42,43]. Futhermore, packages represent a structural architectural view [58]. Although such a view is valuable for determining decay, a semantic architectural view is needed to identify decay involving the concerns of architectural elements.

To stem architectural decay, techniques need to be constructed that predict a variety of constructs related to architectural quality, including indicators of architectural decay, i.e., *architectural bad smells* [42, 43], and the *quality of an architecture's modularization* [71]. Architectural bad smells, which are patterns of architectural constructs that may negatively affect the maintenance of software systems, reduce the quality of a software system's architecture but do not constitute an error that should be fixed in all cases, unlike a defect. Determining that an architectural module is decaying, even before it is involved in an architectural smell or exhibits low modularization quality, can reduce maintenance time and effort.

To forecast architectural decay, I construct *novel models that predict the quality of an architectural element* (i.e., architectural module) by utilizing multiple architectural views (both structural and semantic) and architectural metrics as features for prediction. To obtain multiple architectural perspectives, I utilize two module-level views: a package-level view and a semantic view, obtained by leveraging an information retrieval-based technique [40, 44] shown to work accurately based on the latest evaluations of techniques for recovering a software system's architecture [40, 64]. My architectural-quality prediction models use an effective set of prediction metrics (i.e., file-level metrics, smell-based metrics, and architectural metrics) and metrics for representing architectural quality at the module level (i.e. defects, smell-based metrics, and modularization quality). Each architectural view provides an alternative perspective that can be used to prioritize architectural modules and allocate resources to them for maintenance purposes.

The remainder of this research proposal is organized as follows. In Chapter 2, I describe the problem and specify the scope of this thesis. In Chapter 3, I describe the prior research. In Chapter 4, I discuss the empirical study that shows the impact of software architecture on defect proneness of software systems. In Chapter 5, I discuss the approach for architectural decay prediction from evolutionary history of software and its evaluation results. In Chapter 6, I discuss some of the limitations in my research followed by some suggestions for the community. Finally, I conclude this dissertation with a summary of contributions and

future work in Chapter 7.

Chapter 2: Research Problem

In this chapter, I present the research problem, specific hypotheses, and research questions that will be the focus of this thesis.

2.1 Problem Statement

Software architecture is effective for reasoning about the system's structure, its constituent elements and the relationships among them. It allows engineers to reason about the functionality and properties of a software system without getting involved in low-level source code and implementation details. Bad architectural design, often manifested as architectural bad smells, can increase the complexity, possibly leading to poor software quality.

In practice, however, software architecture is being discounted, in particular by the open source community that has traditionally placed a premium on the code rather than the underlying architectural principles holding a system together. This is not to say such systems are devoid of architecture, but that the architecture is not explicitly represented and maintained during the system's evolution.

There are numerous studies on defect prediction models [48, 76, 78, 97, 110], which help to identify the files with the highest probability of defects. Project managers could use that information to assign the resources more efficiently (e.g for testing purposes in order to find defects in early stages). However, software architecture information is often ignored in building defect prediction models. There are some studies to find defective modules but almost all of those studies are based on a system with some sort of known modules. For example Nachiappan et. al [77] executed their study using a number of Microsoft systems and considered binaries as architectural modules. Some other studies use Java packages as architectural modules [94, 95]. On the other hand, there are numerous architectural bad

smell and decay metrics that can help discover architectural problems in software systems. They, however, have not been used by traditional defect prediction models.

To solve this problem, I introduce new metrics that include both change history and architectural information. I use those metrics to empirically show the impact of architecture on software defects. Subsequently, I build an architectural level defect predictor and investigate its usefulness to detect architecture bad smells and architecture decay in a system. **My dissertation (1) provides empirical evidence of the importance of software architecture on defect proneness of software systems, and (2) devises an approach for using evolutionary history of a software system to detect defective modules and architectural problems.**

2.2 Research Hypotheses

This research investigates two overarching hypotheses, each of which is comprised of several research questions.

Hypothesis 1: I hypothesize that software architecture has an impact on defect proneness of software systems.

This hypothesis is investigated by considering the change history of the software. I investigate three research questions to verify hypothesis 1:

> ***RQ1.a:*** *Are co-changes dispersed across multiple architectural modules more likely to have defects than co-changes localized within an architectural module?*

A positive answer to this question will enable the practitioners (software architects and developers) to use co-change dispersion metrics to assess quality of software, cope with architectural degradation, and also focus on important co-changes or architecturally significant changes first.

RQ1.b: *Do different surrogates for module views exhibit different results in terms of the relationship between co-change dispersion and defects? If so, which surrogate module view provides a better estimate of software defects?*

If the co-change dispersions measured from the various surrogate module views are different in their ability to reveal software defects, practitioners would need to use the views that best reveal defects to further inspect the root causes of the problems; otherwise, it would make more sense to use the view that is easier to obtain.

RQ1.c: *Does a metric that differentiates cross-module co-changes have higher correlation with defects than a co-change metric that does not take into account the architecture?*

If that is the case, then using a metric that distinguishes between the different types of co-change could produce more accurate fault prediction models. The co-change differences, from a software architectural perspective, is a factor that has been largely ignored in the prior research.

Assuming the research is able to empirically corroborate the first hypothesis, a natural question that follows is how such information can be used to improve the software engineering practice. This leads to the second hypothesis:

Hypothesis 2: I hypothesize it is possible to build a prediction model that can help the engineers identify the defective modules and architectural problems in a software system from its change history.

To investigate the second hypothesis, I construct multiple architectural-quality prediction models. Then I seek to answer research questions that assess the effectiveness of proposed architectural-quality prediction models. To that end, I study different regression models, the extent of change of each architectural-smell metric, the ability of models to predict the emergence of an architectural smell, and the metrics that work best for each of the models. Consequently, I study the following research questions to verify hypothesis 2:

RQ2.a: *What is the performance of each prediction model for the different architectural-quality metrics?*

I produce a different prediction model for each architectural-quality metric. To ensure high performance of these prediction models, I intend to determine the most effective regression models for making these predictions. Note that *performance* in this context means the correctness of a prediction model—i.e., performance in the sense used in prediction-model literature.

RQ2.b: *What is the amount of change across releases for each architectural-smell metric?*

To better understand the applicability of my models for predicting architectural smells, the architectural-smell metrics I predict should exhibit change. To that end, I must determine the extent of change for each architectural-smell metric in my research. As a result, I investigate amount of change across releases for each architectural-smell metric.

RQ2.c: *Can we effectively predict architectural-smell emergence between two consecutive releases?*

Potentially, predicting architectural smells is most effective in the case of *smell emergence*, i.e., the addition of smells to a software system. For example, if a module has not had a type of smell in the current release but will have that smell in the next release, my models should predict this occurrence, allowing an engineer to take preventive measures to stem that decay. To that end, I aim to investigate if I can effectively predict architectural-smell emergence between two consecutive releases:

RQ2.d: *What are the important metrics for predicting each architectural-quality metric?*

Although I select prediction metrics that intuitively determine architectural quality, the exact combinations of metrics that best predict architectural quality must be assessed empirically. For my research, I select combinations of metrics that are (1) obtained at the file level and aggregated to modules, and (2) are architectural in nature. Thus, as a final research question, I investigate the importance of prediction metrics for predicting each architectural-quality metric.

Chapter 3: Related Work

I overview prior work covering three areas: defect prediction, one of the most commonly studied prediction models in software-engineering literature; studies focused on architectural evolution or architectural decay; and studies concerned with architectural-quality metrics.

3.1 Defect Prediction

Several studies have shown that metrics mined from code change history can be effective in locating defect-prone code areas [46, 73, 75]. Previous research has also investigated the relationships between code dependency and software quality [15, 21]. Yet another group of studies has investigated the relationships between change coupling (or change dependency) and software quality [27, 97]. My research, on the other hand, is different and new as it investigates the effects of change coupling together with syntactic dependency from an architectural perspective.

Previous studies provided empirical evidence that code that had changed frequently or had a large change in the past tends to have more defects than other code areas [73, 75]. Metrics that measure code dependency are also known to be useful indicators of defect-prone code areas [19]. While code dependency can represent some level of logical relationships between code elements, change coupling metrics are known to be useful to find hidden logical dependency between code elements [38].

Change coupling as an approximation of logical coupling provided many useful applications. Gall et al. [38] proposed the idea of logical coupling that can be identified from the change history. They identified that there is a stronger logical dependency between the changed subsystems when those systems change together in a long subsequence of releases. Such logical coupling is not always obvious from code dependency analysis.

Wong et al. [106] proposed an approach to detect object-oriented modularity violation by comparing the expected change coupling and actual change coupling. They identified expected change coupling using structural coupling identified based on the Baldwin and Clark's design rule theory and identified actual change coupling from software revision history.

Breu and Zimmerman used co-changes to identify cross-cutting concerns [13]. If a call to the same method is made in multiple code locations within a single code change (e.g., lock and unlock), it indicates an aspect. The idea is that a code change is likely to introduce a crosscutting concern if various locations are modified within a single code change. This study did not consider the architecture of the system and also did not correlate the co-changes with defects in the system.

Figueiredo et al. [37] presents a catalog of crosscutting concern patterns recurrently observed in software systems and they analyzed instances of the crosscutting patterns in object-oriented and aspect-oriented versions of three evolving programs and show that a certain category of crosscutting patterns seems to be good indicator of harmful instabilities. This study did not consider the architecture of the system.

Eick et al. used increases in change coupling over time as an indicator of code decay [31, 32]. Since change coupling can be an evidence of concern scatteredness [13], studies on concern scatteredness are relevant to my study. Eaddy et al. [30] showed that the degree of concern scattering and the number of defects are strongly correlated. In their study, a concern is an item from a non-executable specification, such as requirements specification or design. The biggest difference between their study and mine is that they manually mapped concerns and program elements, such as classes and methods, to find concern scatteredness, while I use co-changes as an indirect indication of concern scatteredness. In addition, their metrics are at class level while my approach works at the architectural level, providing useful feedback that could be used for detecting bad smells in the architecture.

In a more recent study, Walker et al. [103] mined the patch history of the Mozilla project to examine whether crosscutting concerns exist. Different from prior studies, they found

that 90% of patches show little or no evidence of scattering at file and module level. However, the study was conducted on only one project and their module level analysis was based on directory structure, while my study was conducted on five projects, four open-source and one commercial, and analyzed from five different architectural perspectives. Walker et al.'s study was also not concerned with the impact of changes on the quality of software.

D'Ambros et al.'s study [27] is closer to mine in that they identified the relationships between change coupling and defects. However, they performed the study at the class level, while my focus is at the architectural level. They did not distinguish between the change coupling of classes from different architectural modules and same module.

Cataldo et al. [19] investigated the impact of three types of dependencies on software failure. The dependencies they investigated include syntactic software dependencies based on analysis of source code (e.g., coupling and cohesion), logical dependencies based on change coupling, and work dependencies based on human and organizational factors. The results suggest that all three types of dependencies are indicative of defects and their impact is complementary. Shihab et al. [97] showed that the number of co-changed files is a good indicator of defects that appear in unexpected locations (surprise defects). Hassan [48] predicted defects using the entropy (or complexity) of code changes. It was shown that the more spread the changes, the higher is the change complexity. Unlike that work, I examine the nature of logical coupling from an architectural point of view.

Offutt et al. [79] presents techniques for measuring couplings in object-oriented relationships between classes. They specifically focused on types of couplings that are not available until after the implementation is finished and presents a tool that measures couplings between classes in Java packages. Unlike their study, I examine the change coupling of classes by considering the architecture.

Poshyvanyk et al. [83] introduced a new set of coupling measures for software systems, measuring conceptual coupling of classes. It is based on measuring the degree that identifiers and comments from different classes are similar to each other. They used information retrieval techniques to measure conceptual coupling and compared it to nine other coupling

metrics in a case study of Mozilla web browser and proved that conceptual coupling is a better predictor of defects. In another study Bavota et al. [8] investigated how class coupling (captured by structural, dynamic, semantic and logical coupling) aligns with developers' perception of coupling. They conducted the study on three open source systems and involved 64 students, academics and industrial practitioners. They found out that the semantic coupling measure is a better estimator of the mental model of the developer than other coupling measures. None of these studies considered the architecture of the system.

Nagappan and Ball [75] used code dependency metrics and code change history metrics to predict failure-proneness. However, they did not examine change-coupling effects. Additionally, they used Windows Server 2003 as a project under study, the source code and architecture of which is not publically available.

Rahman and Devanbu [88] compared the performance of code metrics (e.g., size and complexity) and process metrics (e.g., number of changes, number of developers) using logistic regression and showed that code metrics, despite widespread use in defect prediction literature are generally less useful than process metrics for prediction. In another paper [87], they considered the impact of code ownership and developer experience in on software quality. They found that the implicated code (code that is modified to fix a bug) is more associated with a single developer's contribution. They also found that author's specialized experience in the target file is more important than general experience. Unlike their study, I consider the architectural information in defect prediction.

There are a lot of studies that use different data mining and statistical methods and try to predict the location or number of faults in a software system. Ostrand et al. [81] developed a negative binomial regression model based on code of the file in the current release and fault and change history of the file from previous release. They tried to predict the number of faults for each file in the next release and showed that the 20 percent of the files with the highest predicted number of faults contained on average, 83 percent of the faults that were actually detected. They did not use architectural information in defect prediction.

Menzies et al. [67] used different predictors based on some static code attributes and showed that how the attributes are used to build the predictors is much more important than which attributes are used. Their results indicate that a naive Bayes data miner with a log-filtering preprocessor outperforms a rule base or decision-tree learning method. They also concluded that the best set of attributes for defect prediction varies from project to project and suggested that instead of using a particular subset of attributes for all projects, defect predictors should be built on using all available attributes, followed by subsetting methods to find the most appropriate subset for each specific project and domain. Unlike my research, they did not use architectural-level metrics.

Kim et al. [51] used an SVM classifier to determine whether a new software change is more similar to prior buggy changes or clean changes. They showed that the trained classifier can classify changes with 78 percent accuracy. They identified the bug-introducing and clean changes at the file level by tracing backward in the revision history and used that as one of the attributes to train the classifier. Unlike their study, I examine the changes from an architectural point of view and also I use different classifiers.

Turhan et al. [100] investigated the applicability of cross-company (CC) data for building defect predictors using static code metrics. They observed that defect predictors learned from within-company (WC) data outperformed the ones learned from CC data. They proposed a two phase approach for building defect predictors in companies: in phase one companies should apply analogy-based learning (i.e. nearest neighbor filtering) to CC data and initiate defect prediction process with that and at the same time start collecting WC (local) data. Once they have enough WC data, they should switched to phase two and use predictors learned from WC data. Unlike their study, I build prediction models at architectural-level.

While most of the defect prediction studies are at the file-level, some studies focus on the subsystem level. Mockus and Weiss [73] found that in a large switching software system, the number of subsystems modified by a change can be a predictor of whether the change results in a fault, but the definition of subsystem was not explained, making it difficult to

generalize their observation to other projects. Nagappan et al. [77] used post-release defect history of five Microsoft software systems and found that failure-prone software entities are statistically correlated with code complexity measures. They also found that there is no single set of complexity metrics that could be used universally and it depends on the project. Unlike my research, they chose binary files within Windows as subsystems, making it difficult to generalize their observation to other projects.

Zimmermann and Nagappan [110] investigated the architecture and dependencies in Windows Server 2003, demonstrating how the complexity of a subsystem's dependency graph can be used to predict the number of failures. For each subsystem they measured some graph complexity and density measures and used those as parameters in regression analysis and principle component analysis to predict number of failures for each subsystem. They chose binary files within Windows as subsystems.

Several studies used packages as modules. Martin and Martin [66] introduced the Common Closure Principle (CCP) as a design principle about package cohesion. This principle implies that a change to a component may affect all the classes in that component, but should not affect other components. Although the authors introduce CCP as a guideline for good decomposition of architecture, they do not investigate the impact of it on software defects. Zimmermann et. al [112] showed that complexity metrics are indicators of defects in Eclipse using files and packages. Kamei et. al [50] showed that package-level predictions do not outperform file-level predictions when the effort needed to review or test the code is considered. Schroter et. al [94] showed that import dependencies can predict defects using both files and packages. Bouwers et. al [12] investigated twelve architecture metrics for their ability to quantify the encapsulation of an implemented architecture and used packages for evaluation. The biggest difference between these studies and my research is that, I extract both file-level and architectural-level metrics and use the prediction models to identify architectural problems. I also use architectural recovery techniques for identifying modules.

3.2 Architectural Evolution and Decay

Several studies are concerned with architectural decay across multiple versions of a software system. None of the following studies aim to predict architectural quality or decay.

Two studies have examined architectural decay by using the reflexion method [74], a technique for comparing descriptive architectures (i.e., architectures as designed by its architects) and recovered architectures (i.e., architectures as represented by implementation-level artifacts). Brunet et al. [16] studied the evolution of architectural violations from four subject systems. Rosik et al. [89] conducted a case study using the reflexion method to assess whether architectural drift, i.e., unintended design decisions, occurred in their subject system and whether instances of drift remain unsolved.

Four additional studies investigate different facets of architectural decay. Hassaine et al. [47] present a recovery technique, that they use to study decay in three systems. van Gurp et al. [102] conduct two qualitative studies of software systems to better understand the nature of architectural decay and how to prevent it. D'Ambros et al. [26] present an approach for studying software evolution that focuses on the storage and visualization of evolution information at the code and architectural levels. Mo et al. [72] study patterns of recurring architectural problems at the file and package level, finding evidence of proneness to errors and changes for such entities involved in such patterns. Unlike these studies, I build architectural-level prediction models to predict which modules would have decay in the future. I also use multiple indicators of architectural decay such as number of defects in each module, architectural bad smells, and modularization quality for identifying decay in modules.

3.3 Architectural-Quality Metrics

A variety of metrics have been established in the software-engineering literature that quantify architectural quality and are applicable to architectural modules. Most of the metrics

focus on representing coupling and cohesion between architectural entities. Other metrics consider the concerns (i.e., concepts, roles, or responsibilities) of the software system. Furthermore, some metrics have been applied to studies of architectural evolution.

Several studies focus on coupling and cohesion metrics for architectural modules. Allen and Khoshgoftaar [4] define coupling and cohesion metrics based on information theory. Briand et al. [14] present coupling and cohesion metrics based on object-oriented design princples. Sarkar et al. [92, 93] defined a series of metrics concerned with quality at the module and object-oriented levels. Most of these metrics highly overlap with previous metrics and are based on coupling and cohesion. Many of these metrics overlap with constructs measured by our selected metrics, while others are dependent on specific technologies or are not fully automatable—precluding their inclusion in my study.

Sant'Anna et al. [91] present architectural metrics based on concerns. These metrics are highly similar to concern-based architectural smells and focus on aspect-oriented systems. They do not provide mechanisms for identifying concerns that are not aspect-oriented, precluding the use of these metrics for my study.

Wermelinger et al. [105] apply architectural-decay metrics across multiple releases of Eclipse, with a focus on coupling, cohesion, and stability metrics. Sangwan et al. [90] apply architectural complexity metrics to multiple versions of Hibernate. Finally, Zimmerman et al. [109] propose that true coupling is determined by studying revision histories and code-level entities rather than the decomposition of modules or files. None of this previous work aims to predict architectural quality, which is the focus of my research.

In summary, while the majority of existing studies on defect prediction are at the file level, my study is at the architectural level. I further examine other indicators of architectural decay and quality other than defects (i.e., architectural smells and modularization quality). Furthermore, existing studies of prediction models at the subsystem level used either packages as architectural modules or other pre-defined modules (e.g. studies on Windows that used binaries as architectural modules). In this dissertation, I use packages and

recovery techniques for identifying modules from source code. These recovered architectural views enable us to build architectural prediction models for any system, even if a ground-truth architecture is unavailable.

Chapter 4: The Impact of Software Architecture on Defect Proneness of Software Systems

This chapter discusses the methodology to investigate the first hypothesis (See 2.2). It discusses the empirical studies and the techniques I used, followed by the results [53, 55].

4.1 Methodology Overview

The method designed and implemented to run the empirical study involves four components represented by rectangles in Figure 4.1. The first component is *Co-change Extractor*, which searches source code repositories and retrieves the groups of files that have been changed together. It identifies the co-changes by going through the developer commits to the SVN repository and extracting the groups of files in the same commit transaction that have been modified together. The current implementation of the *Co-change Extractor* component uses *SVNKit*, a Java toolkit providing APIs to access and work with subversion repositories. This component has a modular design, and can be easily extended to support other source code repositories as well.

The second component is *Defect Extractor*, which parses the commit logs of projects and identifies the software changes that introduced the defects in the system. *Defect Extractor* and *Co-Change Extractor* components are synchronized with each other, to implement an *n-months data collection* approach, where the co-changes are extracted from the first n-months after a certain release and the introduced faults are retrieved from the next n-months after the co-changes are retrieved. While *Co-change Extractor* component obtains the information of co-changes from the source code repository, the *Defect Extractor* component retrieves the information from the next n-months, and finds which of the original co-changed files introduced defects in the next n-months time slice.

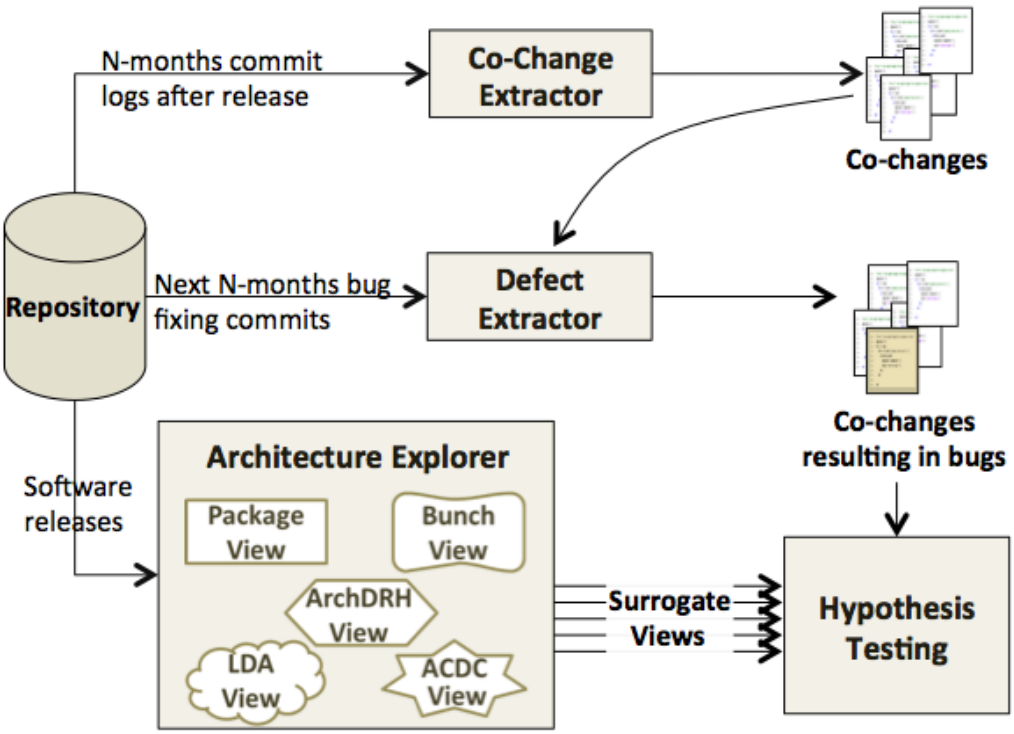

Figure 4.1: Overview of the Experimental Method.

To examine the effects of co-change dispersion among the system's architectural modules on software defects I incorporated a third component, *Architecture Explorer*, which reconstructs the module view of architecture for the experiments. At the state of practice, there is no reverse engineering technique that can produce the "ground truth" architecture [41]. *Architecture Explorer* component thus utilizes different reverse engineering approaches and obtains several *Surrogate Views* that approximate the system's architecture. The surrogate views are then used in the last experimental module, *Hypothesis Testing*, where the effects of software co-change dispersion are examined from an architectural perspective.

4.2 Obtaining Surrogates for Architectural Module View

In this section, I describe the architectural representations that I used in the study.

Comprehending the architecture and architecturally significant issues of any complex

system requires looking at the architecture from different perspectives [7, 59]. These perspectives are known as *architectural views*, each dealing with a separate concern. According to Clements et al. [22], three view types are commonly used to represent the architecture of a system: *Module View*, *Component-and-Connector View*, and *Allocation View*. Module View shows units of implementation, Component-and-Connector View represents a set of elements that have runtime behavior and interactions, and Allocation View shows the relationships between software and non-software resources of development (e.g., team of developers) and execution environment (e.g., hardware elements).

Since this study is concerned with the construction and evolution of software, and not its runtime or deployment characteristics, Module View is the relevant view to focus on. Module View determines how a system's source code is decomposed into units and it provides a blueprint for construction of the software.

In reality, many projects lack trustworthy architecture documentation, therefore, I used different techniques to reverse engineer five surrogate models that approximate such architecture: *Package View*, *Bunch View* [71], *ArchDRH View* [18], *LDA View* [9], and *ACDC View* [101]. Although there might have been various techniques to reconstruct the architecture, I chose those that have the highest degree of automation, and therefore are applicable to the context of the empirical study.

4.2.1 Package View

An intuitive approximation of the system's architecture in Module View is the Package View, where packages represent the system's architectural modules. It is reasonable to assume the package structure is a good approximation of the decomposition of the system into architecturally significant elements, as packages are created by the developers of the system. In fact, package structuring has been used as a decomposition reference in prior research as well [10]. Therefore, one can say that package structuring of a Java project is representative of Module View architecture in which each architectural module consists of several Java classes (as a package) and the relation between them is *is-part-of*. There could be different

decomposition layers when we are looking at the package structuring. It can be seen as a tree considering each class as a leaf that is part of a package, which itself may be part of a bigger package, with a top package as the root. Package view is considered at two levels. In the high-level package view, we consider each of the top-level directories as one of the architectural modules. For example, in *OpenJPA*, we consider each subfolder of the project (i.e., *org.apache.openjpa.jdbc*, *org.apache.openjpa.kernel* and *org.apache.openjpa.persistence*) as an architectural module. In the low-level view, architectural modules are represented by enclosing directories of each file.

4.2.2 Bunch View

Bunch [71] is a reverse engineering tool that produces clusters based on the dependencies among the classes. Bunch is a fast, scalable and easy-to-use tool. Prior research has shown that it is among the best available tools for reverse engineering the system's architecture [107]. Bunch relies on source code analysis tools to transform the source code to a directed graph. I used *Class Dependency Analyzer (CDA)*[1] which is a tool that extracts dependencies between Java classes. CDA takes Jar file of a project as input. Figure 4.2 shows the Class Dependency Analyzer tool when is loaded with version 1.6.0 of Apache Camel project. You can export the dependencies between classes in the form of *Object Dependency Exploration Model (ODEM)* which is in XML format. Dependencies between classes that are shown in the ODEM file are binary relations that are supported by programming languages like procedure invocation, variable access, and inheritance. Next I wrote an script to convert the ODEM format to one that are accepted by Bunch. Each line of the input file for Bunch has the following format:

```
file1 file2
```

This shows that file1 has a dependency on file2.

Basically, this input is a graph that represents the source code artifacts and their relation dependencies, called Module Dependency Graph (MDG). The clustering output of

[1] http://www.dependency-analyzer.org/

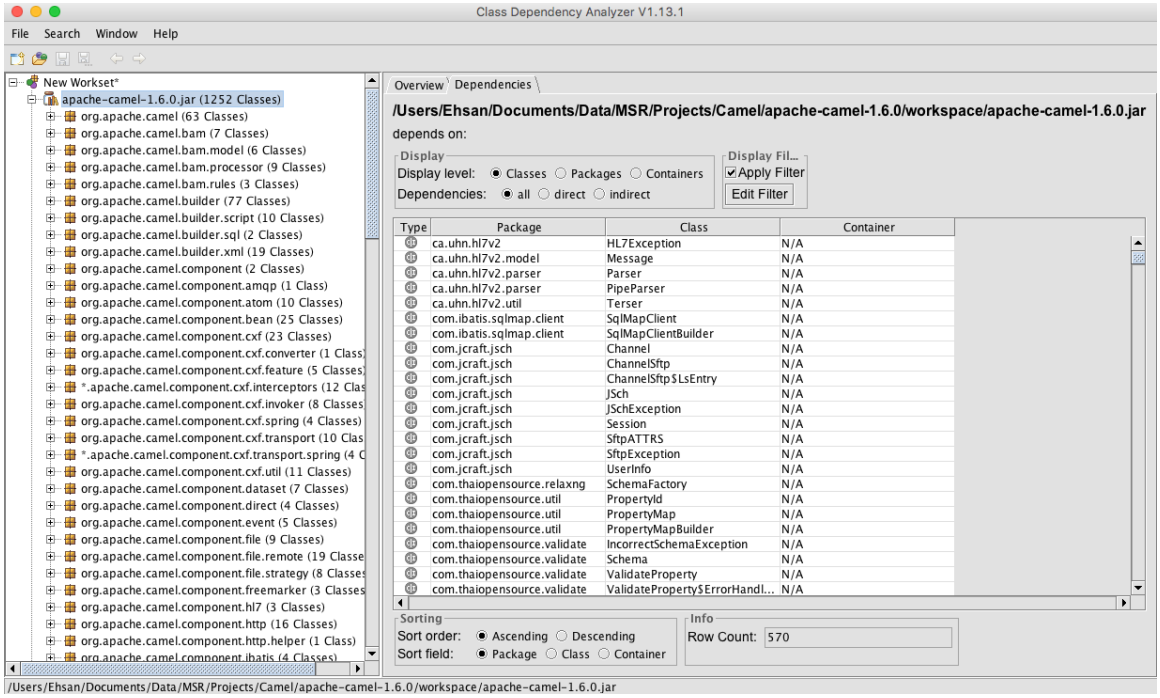

Figure 4.2: Class Dependency Analyzer.

Bunch is a representation of Module View architecture, where the elements of this architecture correspond to the clustered classes. These clusters represent *depends-on* and *is-a* relationships in the system.

Bunch's approach to solve the clustering problem is to find a good partition in the Module Dependency Graph. The term *partition* is used in the graph-theoretic sense, that is, the decomposition of a set of elements (i.e., all nodes of graph) into mutually disjoint sets (i.e., clusters). A good partition by Bunch means a partition where highly interdependent classes are grouped in the same architectural module and conversely, independent classes are assigned to separate subsystems.

Since projects can have a large number of classes, finding a good partition involves navigating through a very large search space of all possible partitions. To solve this search problem in an efficient way, Bunch tries to maximize an objective function called *Modularization Quality (MQ)*. MQ determines the quality of a partition as the trade-off between

interconnectivity (dependencies between classes of two different architectural modules) and intraconnectivity (dependencies between the classes of the same module). Therefore Bunch would result in creating of highly cohesive architectural modules that are not coupled excessively.

MQ for a given architecture is defines as [71]:

$$MQ = \sum_{i=1}^{|C|} CF_i$$

where CF_i is the "cluster factor" of module i, representing its coupling and cohesion. CF_i is defined as:

$$CF_i = \begin{cases} 0 & \mu_i = 0 \\ \dfrac{2\mu_i}{2\mu_i + \sum\limits_{\substack{j=1 \\ j \neq i}}^{k} (\epsilon_{i,j} + \epsilon_{j,i})} & \text{if } x < 0 \end{cases}$$

where μ_i is the number of edges within the module, which measures cohesion; and $\epsilon_{i,j}$ is the number of edges from module i to module j, which measures coupling. The CF is defined as a normalized ratio between the total weight of the internal edges (edges within a module) and half the total weight of the external edges (edges that exits or enters a module).

Figure 4.3 shows an example of *Module Dependency Graph* for a small compiler developed at University of Toronto. Figure 4.4 shows the partitions generated by Bunch. Bunch generates four modules for code generation, scope management, type checking and parsing services of the compiler.

Bunch provides three optimization strategies to find a good partition: *genetic*, *hill climbing* and *exhaustive* search. Hill climbing algorithms produce high quality results in a reasonable time and therefore I selected the hill climbing algorithm for clustering.

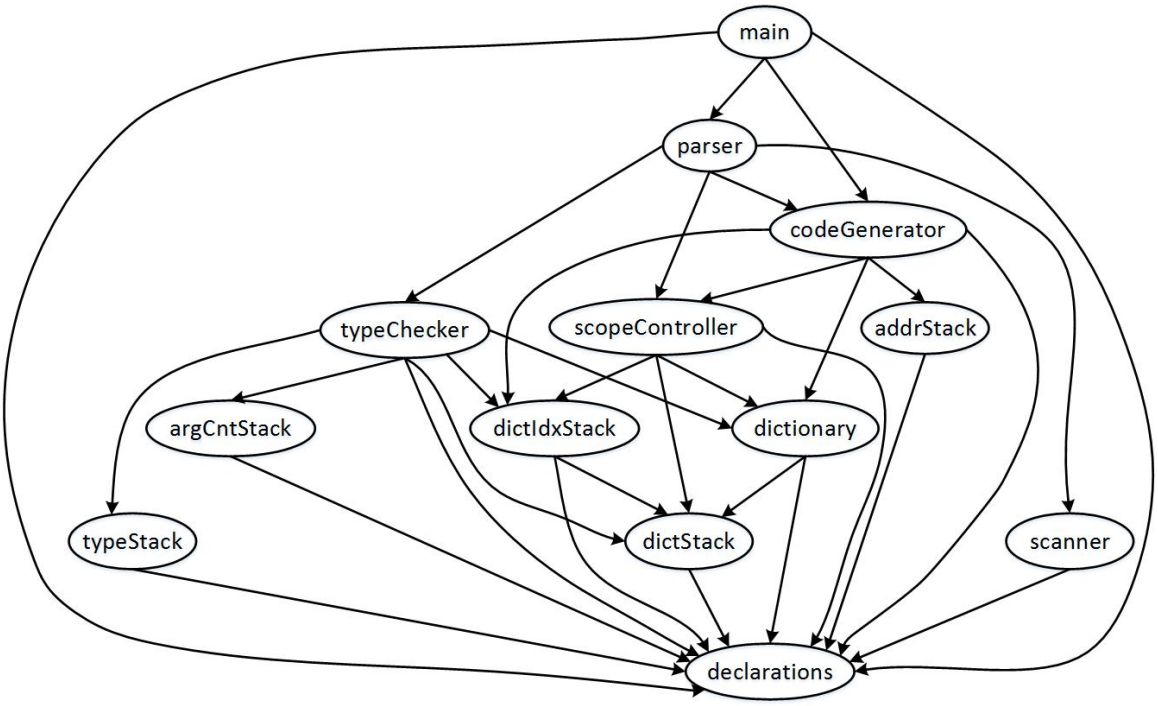

Figure 4.3: A Module Dependency Graph for a Compiler (Reproduced from [71] With the Approval from the Authors).

4.2.3 ArchDRH View

Based on the rationale that well-modularized systems are usually designed with stable *design rules* (they are often implemented in the form of architectural level interfaces that decouple the rest of the system into modules), Cai et al. [18] proposed an architecture recovery algorithm called the Architectural Design Rule Hierarchy (ArchDRH).

For example, if a system employs an observer pattern, then there should be an observer interface in the source code. This interface has a special position in the architecture. First, it decouples subjects from concrete observers: the subjects just need to depend on the observer interface, but not any concrete observers. Second, this interface should be stable because both subjects and observers will be impacted by its changes. In this case, we consider this observer interface as a *design rule*, and the subjects and observers as two independent modules decoupled by the design rule. As another example, the abstract factory pattern

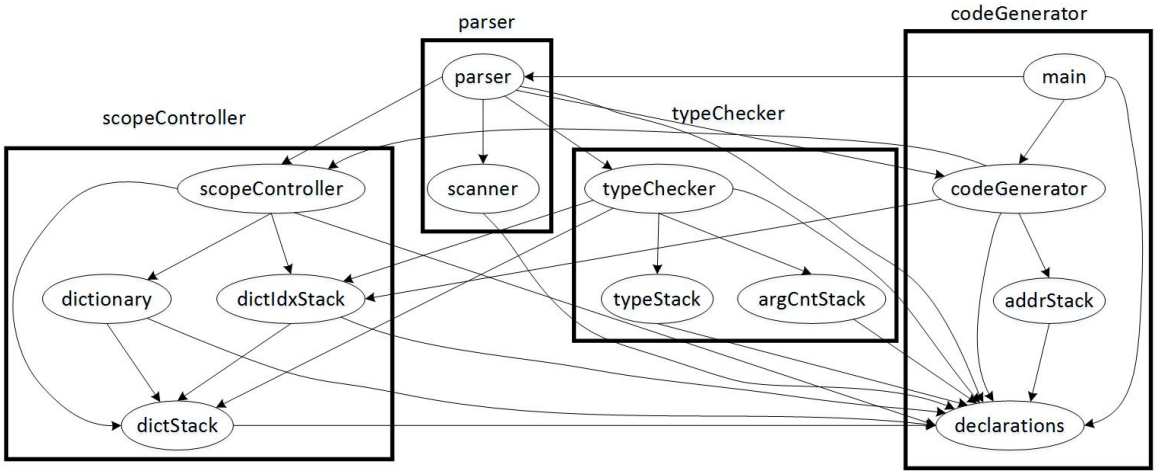

Figure 4.4: The Partitioned MDG for a Compiler (Reproduced from [71] With the Approval from the Authors).

requires creating an abstract factory interface that decouples multiple concrete factories. In this case, the abstract factory interface is one instance of a design rule.

The strong characteristics of design rules cannot be fully recovered and exploited by using traditional software clustering techniques (e.g. coupling and cohesion). Design rules in a system should not belong to any modules. Instead they dominate subordinate modules and frame the overall structure of a system.

The key features of ArchDRH algorithm are as follows. First, it finds such design rules and gives them a special position in the architecture, rather than aggregating them into subordinating modules as other clustering methods would do. Second, based on the observation that a software system usually has one or more main programs that depend on many other elements, acting as controllers or dispatchers, ArchDRH also separates these controllers and gives them a special position in the architecture.

After that, ArchDRH separates the rest of the system into modules based on the principle of maximize parallelism between modules. Concretely, given the dependency graph formed by the rest of the system, it calculates its connected subgraphs. For a subgraph that is still large, the algorithm further separates design rules and controllers within the subgraph,

and processes the rest of it recursively till a stop condition is met, e.g, all the subgraphs are strongly connected. This way, the algorithm outputs a hierarchical structure, which is called a *design rule hierarchy*.

ArchDRH can be used alone or be combined with other recovering techniques to recover software architecture more effectively. When each module is generated by ArchDRH algorithm, the user can plug in any other recovery techniques depending on the system under study. For example if classes in the system follow strong naming convention, applying ACDC after ArcDRH may result in more accurate architecture.

4.2.4 LDA View

Yet another way to reconstruct the modular decomposition of architecture is to use *Information Retrieval* and *Data Mining* techniques, such as *Latent Dirichlet Allocation (LDA)*, which is a known approach to automatically discover the underlying structure of the data. In the context of software engineering, this method has been used to discover the modular decomposition of a system [9], a conceptual implementation architecture [68], and capturing coupling among classes in OO software systems [45].

LDA analyzes the underlying latent topics, words, and terms used to implement each class/source files and discovers the most relevant topics describing the system. Therefore, based on the similarity of each source file and the discovered topics, it decides which source files should be part of the same module.

Unlike the previous reconstruction approaches, which utilize the structural dependencies between classes to find a potential modularization view, LDA uses the textual similarities between the contents of these classes and clusters them into different modules. The number of reconstructed modules is equal to the number of discovered underlying topics.

4.2.5 ACDC View

The Algorithm for Comprehension-Driven Clustering (ACDC) [101] clusters program entities based on the principle of easing comprehension. ACDC provides meaningful names

for the obtained clusters rather than names such as subsystem01. It also avoids making partitions of a system in a way that majority of classes are located in one cluster while other clusters have very few classes. It is based on the fact that clusters with limited number of classes (up to 20) are more manageable and easier to understand.

ACDC clusters program entities based on a list of subsystem patterns that are observed in manual decomposition of software systems. Following is a list of some of the patterns:

- *Source file pattern:* If a source file contains the definition of one or more procedures, ACDC would group them together into one cluster.
- *Directory structure pattern:* Directory structure of the source code may sometimes correspond to modules.
- *Body-header pattern:* In languages like C++ where a procedure being split between two different files, e.g. a .cpp file and a .h file, ACDC would put them in the same cluster.
- *Leaf collection pattern:* This pattern applies when a set of files are not dependent on each other but serve similar purposes e.g. device drivers. These files are usually leaves of a system's graph and should be placed in the same cluster.
- *Support library pattern:* If several procedures are accessed by majority of its classes, ACDC would group them in the same cluster.
- *Central dispatcher pattern:* If there is a class with a large number of outgoing edges (i.e it depends on a large number of classes), ACDC first disregards that class and its outgoing edges and later reconsiders them in conjunction with other formed modules.
- *Subgraph dominator pattern:* This pattern searches a system's dependency graph to find subgraphs with the following characteristics: subgraph should have a node n_0 (dominator node) in which, there is a path from n_0 to every other node in that subgraph.

The clustering algorithm of ACDC has two stages: In the first stage in creates a skeleton of the final decomposition of the system using these patterns. The second stage aggregates the leftover elements using a technique known as orphan adoption. I will briefly explain these two steps:

- *Skeleton construction:* ACDC first starts by identifying classes according to source file pattern and body-header pattern. Then it identifies the collection of files that are matched with leaf collection pattern and support library pattern. Then it follows the central dispatcher pattern and disregards classes with more than 20 outgoing dependency links. Then it goes through the nodes in the graph to find dominator nodes as explained in subgraph dominator pattern. ACDC checks the nodes from the smallest number of outgoing dependencies to the largest. This results in creating smaller subsystems. When ACDC discovers a set of dominated nodes, it creates a subsystem containing both the dominated set and the dominator node. After checking all the nodes and creating subsystems, ACDC might move subsystems with low cardinality (less than four) to the higher level subsystems.

- *Orphan adoption:* This stage assigns the remaining classes (orphans) to the existing modules. The idea is that orphans should be assigned to the subsystems that have more connectivity to the orphan than any other subsystem.

4.3 Measuring Effects of Co-change Dispersion

The goal of my study is to examine the effects of co-change dispersion from an architectural perspective. To that end, I formulated the three research questions described in Section 2.2. To answer those questions, I define two metrics discussed in the following subsection.

4.3.1 Metric Definition

To answer *RQ1.a*, I compare the number of co-changes made within an architectural module and across multiple architectural modules for a given file. For this purpose, this section defines metrics to quantify the number of co-changes with respect to the system's architectural modules.

Let $S = < F, P_m, C >$ be a project, consisting of a set of files F, structured in a set of modules P_m under the architectural model m, and a set of commits C. Each file is assigned to a module and none of the modules overlap. More formally, the set $P_m \subseteq \mathcal{P}(F)$ is a partition of F under the architectural model m. The relationship between a file and a module in the m architectural model is captured by a function $p_m : F \rightarrow P_m$, and a set of co-changed committed files is identified by a function $h : C \rightarrow \mathcal{P}(F)$.

I can now define the two metrics for *intra-module co-changes (IMC)* and *cross-module co-changes (CMC)* using set cardinality expression.

Definition 1 (CMC)**.** *Number of co-changes for a file, f_i, where the co-changes are made across more than one architectural module:*

$$CMC(f_i) = \mathbf{card}(\{c : C |\ f_i \in h(c)\ \wedge\ \exists\ f_j\ \in\ h(c)\ .\ p_m(f_i)\ \neq\ p_m(f_j)\})$$

Definition 2 (IMC)**.** *Number of co-changes for a file, f_i, where there is at least another co-changed file in the same architectural module:*

$$IMC(f_i) = \mathbf{card}(\{c : C |\ f_i\ \in\ h(c)\ \wedge\ \exists\ f_j\ \in\ h(c)\ .\ p_m(f_i)\ =\ p_m(f_j)\})$$

More intuitively, Figure 4.5 illustrates the differences between these two metrics using two surrogate architectures for a small hypothetical example. Figure 4.5(a) depicts the Package View, which includes two packages and classes inside them, denoting the *is-part-of* relation. Based on this architectural view surrogate, package 1 and package 2 are the

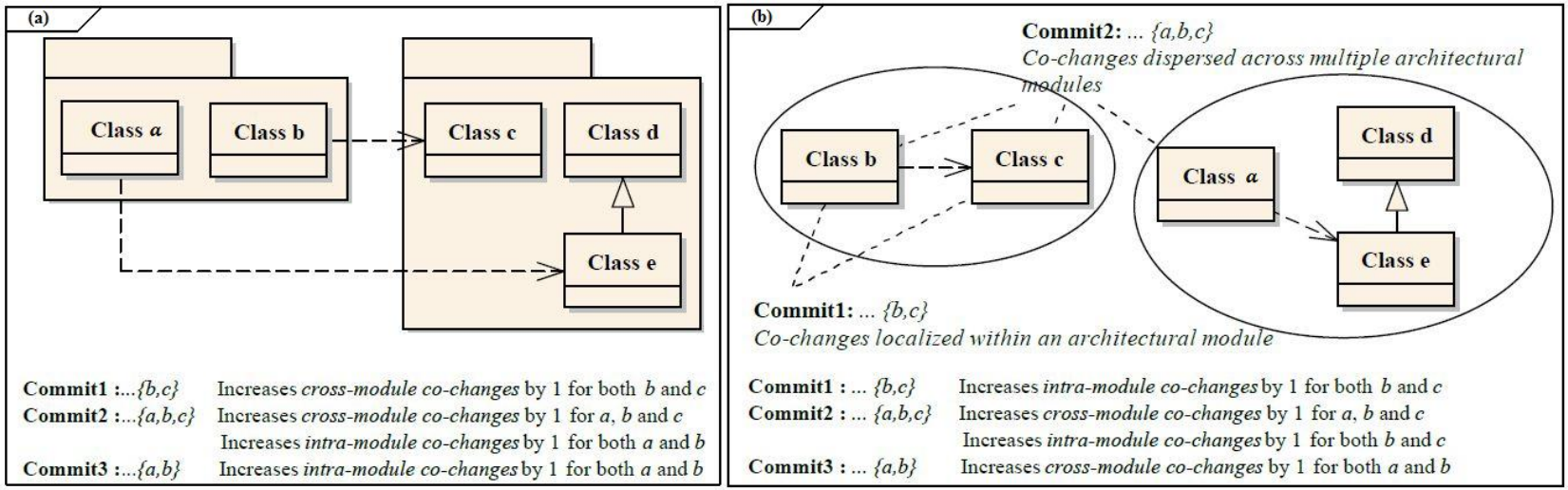

Figure 4.5: Architectural Module View Surrogates of a System: (a) Package View, and (b) Cluster View.

architectural modules of the system. Package 1 includes two classes of a and b; Package 2 includes three classes of c, d and e.

In Figure 4.5(b) an alternative surrogate view is shown, which is obtained by clustering the classes using Bunch (recall Section 4.2.2). Here, there are two *depends-on* relationships (i.e., *a*-*e* and *b*-*c*) and one *is-a* relationship (i.e., *e*-*d*). Based on these dependency relations, Bunch generates two clusters: Cluster 1 includes classes *b* and *c*; Cluster 2 includes classes *a*, *e* and *d*. These clusters here are considered as the architectural modules of the system.

As illustrated at the bottom of Figure 4.5(a) and Figure 4.5(b), suppose that *a*, *b* and *c* are the co-changing files from three commits to the repository: {*a, b*}, {*a, b, c*}, {*b, c*}. All the files in the same set have been changed in a single commit. I am able to calculate the metrics from these commits. For example, in Figure 4.5(b), from the commit {*b, c*}, the values of *IMC* for both *b* and *c* increase by 1, because both of the files are in the same architectural module. From the commit {*a, b, c*}, the values of *CMC* for *a*, *b* and *c* increase by 1, because *a* is in a different architectural module. The *IMC* values for both *b* and *c* also increase by 1, since they are in the same architectural module. Respectively Figure 4.5(a) shows how each metric changes for the Package View surrogate.

I use regression analysis to examine the effect of *cross-module co-changes* and *intra-module co-changes* on the number of defects per file.

4.3.2 Underlying Characteristics of the Data

Before adopting any statistical method to answer the research questions, I examined the nature and statistical characteristics of the mined data. This section presents the characteristics of the dataset and provides rationale for choosing the appropriate statistical methods.

The data used in the experiments are non-negative integers, representing the numbers of faults, therefore can be considered as *count* or *frequency* data. By using Q-Q normal plot, I realized that the data does not follow a normal distribution. Also by using scatter plot of *cross-module co-changes* and *intra-module co-changes* with defects, I observed

that these two metrics do not have a linear relationship with defects. The collected data, moreover, contains numerous zeroes (i.e., files that do not change or do not have defects). The same phenomena have also been observed by other researchers [81, 95, 108]. In fact, it has been shown that the distribution of fault data over modules in real systems are highly unbalanced [108].

4.3.3 Analysis Method

Considering the characteristics of the data (see 4.3.2), I ruled out the option of using Linear Regression as it assumes the defects to be normally distributed [108], which is certainly not the case here. Unlike linear regression, *negative binomial regression (NBR)* makes no assumptions on either the linearity of the relationship between the variables, or the normality of the variables distributions [23]. Therefore NBR is an appropriate technique to relate the number of defects in a source file to the two co-change metrics.

This model is thus applicable to count data and even more importantly addresses circumstances such as over-dispersion [23], as used in previous studies [81, 84].

I want to model the relationships between the number of defects (Y) in the source files and the two metrics. Suppose that y_i represents the number of defects for file i and x_i is a vector of the two metrics (*CMC* and *IMC*) for that file. NBR specifies that y_i, given x_i, has a Poisson distribution with mean $\lambda_i = \gamma_i e^{\beta' x_i}$, where β is a column vector of regression coefficients (β' is the transpose β) and γ_i is a random variable drawn from a gamma distribution [81].

Specifically, the output of this regression model is the vector $\beta = [\beta_1, \beta_2]$, where β_1 is the coefficient of *CMC* and β_2 is the coefficient of *IMC*. By using NBR, the expected number of defects varies as a function of *CMC* and *IMC* in multiplicative factor (unlike the case of linear regression, where the outcome is an additive function of the explanatory variables).

Since the co-changes data have a long tail (some files have a lot of co-changes compared to others), I use the log_2 transformation of the metrics to reduce the influence of extreme

values [23]. Furthermore, as NBR models the natural log of the dependent variable (number of faults), the coefficients can be interpreted as follows: for one unit of change in the independent variable (e.g., *cross-module co-changes*), the log of the dependent variable (number of faults) is expected to change by the value of the regression coefficient (β_1). To make the idea concrete, suppose that the coefficient of *CMC* is 0.8. This means that a unit change in log_2 of *cross-module co-changes* is associated with an increase of 0.8 in natural logarithm of the expected number of defects. In essence, this would result in multiplicative factor of $e^{0.8} = 2.22$ in defects.

I hypothesize that co-changes across different architectural modules are more correlated with defects than co-changes within the same architectural module. If my hypothesis is true, β_1 should be greater than β_2.

4.4 Executing the Analysis

This section describes in detail how I conducted an experimental study to investigate the research questions about impacts of architecture and modularity on bugs.

Projects Studied. The experimental subjects are seven projects from diverse domains, listed in Table 4.1.

The first five projects, i.e., *HBase*, *Hive*, *OpenJPA*, *Camel* and are *Cassandra* are Java-based and open-source, maintained by the Apache Software Foundation. *Hive* is a data warehouse system for Hadoop; *OpenJPA* is an open-source implementation of the java persistence API; *HBase* is a distributed, scalable, big data store; *Camel* is a rule-based routing and mediation engine that provides a Java object-based implementation of the Enterprise Integration Patterns using an API (or declarative Java Domain Specific Language) to configure routing and mediation rules and *Cassandra* is an open source distributed database management system designed to handle large amounts of data across many servers, *Hadoop*, the sixth subject system, is a middleware framework widely used for distributed processing

Table 4.1: Studied Projects and Release Information.

Project	Description	Releases	SLOC	Architecture
HBase	Distributed Scalable Data Store	0.1.0 ,0.1.3 ,0.18.0 ,0.19.0, 0.19.3, 0.20.2, 0.89.20100621, 0.89.20100924, 0.90.2, 0.90.4, 0.92.0, 0.94.0	39K-246K	Recovered
Hive	Data Warehouse System for Hadoop	0.3.0, 0.4.1, 0.5.0, 0.6.0, 0.7.0, 0.7.1, 0.8.1, 0.9.0	66K-226K	Recovered
OpenJPA	Java Persistence Framework	1.0.1, 1.0.3, 1.1.0, 1.2.0, 1.2.1, 1.2.2, 2.0.0, 2.0.0-M3, 2.0.1, 2.1.0, 2.1.1, 2.2.0	153K-407K	Recovered
Camel	Integration Framework based on Enterprise Integration Patterns	1.6.0 ,2.0.M ,2.2.0 ,2.4.0, 2.5.0, 2.6.0, 2.7.1, 2.8.0, 2.8.3, 2.9.1	99K-390K	Recovered
Cassandra	Distributed Database Management System	0.3.0 ,0.4.1 ,0.5.1 ,0.6.2, 0.6.5, 0.7.0, 0.7.5, 0.7.8	50K-90K	Recovered
Hadoop	Distributed Computing Framework	0.19	224k	Ground-truth
System J	An Industrial Product	-	300K	Recovered

of large-scale data across clusters. For the Hadoop project, I had access to its ground-truth architecture, obtained through a manual recovery process by other researchers and verified by the key developers of Hadoop [41]. The last project that I studied is an industrial software project, called *System J*, which is a code-name for a system that has also been the subject of a prior empirical study [95]. It is a two-year old development project, comprised of about 300 KSLOC of Java in 900 files, and structured in 165 Java packages. The system aggregates a certain type of data from many sources and uses it to support both market and operational decision-making at a time granularity of minutes to hours. It has a service-oriented architecture and a transactional database, both implemented with third-party platform technologies.

Data Collection. There are several techniques for linking a bug database and a version archive of a project for finding fix-inducing changes e.g, searching the commit logs for specific tokens like *bugs, fixes,* and *defects* followed by a number [98]. Unfortunately, developers do not always report which commits are defect fixes. Prior work suggests that such links can be a biased sample of entire population of fixed defects [6]. But in the software repositories (Apache Foundation) and the projects studied in this dissertation, the commits that are defect fixes are distinguishable as they specify project name and defect number as a value pair in their commit logs in SVN. For example, all of the defect fixes in HBASE start with "HBASE-bug number" (e.g., HBASE-3172). This enabled the *Defect Extractor* component (recall Section 4.1) to find all defect fixes by just parsing the log of commits in SVN and finding the keyword "HBASE-bug number."

For co-change metrics, I considered a maximum of 30 for the number of files that are changed together in a single commit. The reason is that often when lots of files are changed together, it is due to simple refactoring (e.g., a change in the naming convention or commenting style), which neither requires the developer to understand the impact of changes on the application logic, nor poses a possibility of introducing bugs.

I intentionally chose equal periods of time for collecting both co-changes and bug fixes to have a meaningful comparison of the results. I performed the study using both 3 and

6 months time intervals, which produced consistent results. The results reported here are based on a 3 months time interval. In the first 3 months, I obtain the information of co-changes from the source code repository. Subsequently, in the next three months, I find the files that have been changed to fix bugs. This is repeated for the entire duration of the revision history.

At first it may seem that I could have simply used the time between consecutive releases, but I observed that in many cases, the periods of time between releases are not consistent. For example, the intervals between 4 consecutive releases of HBase project (0.90.3-0.90.6) are 66, 153, and 85 days. If I were to follow the release dates, one data point would be based on collecting the co-changes in 66 days and bug fixes in 153 days, while the next data point would be based on collecting the co-changes in 153 days and the bug fixes in 85 days. Rather, to have unbiased results required for conducting this study, I take the approach of using equal time for collecting both co-changes and bug fixes, which is consistent with previous studies in the literature [76].

Interested readers may access the research artifacts at:

http://www.sdalab.com/projects/ccdispersion.

4.5 Results of the Study

This section describes the results of empirically analyzing the data in the manner described in the previous section.

4.5.1 Results for RQ1.a

To address the first research question—whether co-changes dispersed across architectural modules are more likely to have defects than intra-module co-changes—I use NBR to model the count data against the two metrics I defined (recall Section 5.1.4). I include the file size (LOC) to control for the relationship between the size of files and the number of defects, as it could be argued that the larger files are more likely to have bugs and be a party in cross-module co-changes, thus creating a confounding effect [33, 108].

Table 4.2: Regression Results for Architectural Views of (a) Bunch, (b) ArchDRH, (c) ACDC, (d) High-Level Package, (e) Low-Level Package, and (f) LDA.

Project	Metrics	Bunch View		ArchDRH View		ACDC View		High-level Package		Low-level Package		LDA View	
		Est	Pr(> |z|)	Est	Pr(> |z|)	Est	Pr(> |z|)	Est	Pr(> |z|)	Est	Pr(> |z|)	Est	Pr(> |z|)
HBase	(Intercept)	-3.59	<2e-16	-3.57	<2e-16	-3.64	<2e-16	-3.71	<2e-16	-3.74	<2e-16	-3.74	<2e-16
	log(IMC)	-0.03	0.562	0.15	0.00169	0.02	0.588	0.13	0.0161	0.15	0.00772	0.14	0.0151
	log(CMC)	0.57	<2e-16	0.40	5.43e-10	0.52	<2e-16	0.40	3.7e-12	0.38	2.39e-09	0.36	9.86e-09
	log(LOC)	0.36	<2e-16	0.37	<2e-16	0.37	<2e-16	0.39	<2e-16	0.39	<2e-16	0.39	<2e-16
Hive	(Intercept)	-4.06	<2e-16	-4.34	<2e-16	-4.06	<2e-16	-3.65	<2e-16	-3.68	<2e-16	-4.21	<2e-16
	log(IMC)	0.15	0.102	0.13	0.171	0.26	0.00315	0.16	0.04	0.05	0.454	0.22	0.0252
	log(CMC)	0.53	1.29e-07	0.67	1.15e-09	0.48	7.76e-06	0.50	1.19e-10	0.72	<2e-16	0.56	4.09e-07
	log(LOC)	0.32	<2e-16	0.33	2.66e-15	0.31	<2e-16	0.28	<2e-16	0.24	<2e-16	0.31	<2e-16
OpenJPA	(Intercept)	-4.47	<2e-16	-4.70	<2e-16	-4.41	<2e-16	-4.37	<2e-16	-4.41	<2e-16	-4.42	<2e-16
	log(IMC)	0.01	0.922	0.19	0.00814	0.05	0.484	0.29	8.88e-05	0.02	0.709	0.05	0.438
	log(CMC)	0.59	6.02e-10	0.44	5.96e-06	0.55	1.02e-08	0.23	0.00526	0.62	4.68e-10	0.54	2.73e-09
	log(LOC)	0.37	<2e-16	0.40	<2e-16	0.37	<2e-16	0.39	<2e-16	0.36	<2e-16	0.37	<2e-16
Camel	(Intercept)	-4.14	<2e-16	-4.18	<2e-16	-4.17	<2e-16	-4.25	<2e-16	-4.30	<2e-16	-4.16	<2e-16
	log(IMC)	0.11	0.00155	0.09	0.00971	0.14	8.70e-05	0.18	9.52e-07	0.14	8.29e-05	0.07	0.0358
	log(CMC)	0.21	4.22e-09	0.25	1.96e-11	0.17	1.92e-05	0.17	4.13e-07	0.22	1.03e-08	0.25	3.62e-10
	log(LOC)	0.52	<2e-16	0.52	<2e-16	0.53	<2e-16	0.53	<2e-16	0.53	<2e-16	0.52	<2e-16
Cassandra	(Intercept)	-3.60	<2e-16	-3.48	<2e-16	-3.54	<2e-16	-3.20	<2e-16	-3.27	<2e-16	-3.28	<2e-16
	log(IMC)	0.20	0.0166	-0.05	0.336799	-0.06	0.418	-0.05	0.55872	-0.01	0.962640	-0.01	0.84988
	log(CMC)	0.41	1.36e-05	0.63	<2e-16	0.69	1.13e-11	0.69	7.75e-12	0.64	1.77e-10	0.64	8.72e-13
	log(LOC)	0.21	1.89e-05	0.18	0.000192	0.19	5.46e-05	0.14	0.00163	0.15	0.000942	0.15	0.00115
System J	(Intercept)	-3.58	<2e-16	-2.89	<2e-16	-2.82	<2e-16	-2.93	<2e-16	-2.98	<2e-16	-	-
	log(IMC)	-0.05	0.68	-0.02	0.79	0.09	0.31	-0.07	0.46	-0.03	0.53	-	-
	log(CMC)	0.82	<2e-16	0.75	<2e-16	0.72	<2e-16	0.78	<2e-16	0.81	<2e-16	-	-

Table 4.2 summarizes the results for the five surrogate models of Bunch, ArchDRH, ACDC, Package, and LDA view. Since we did not have access to the source code of the commercial project, we could not generate the data for the LDA view. Each row shows the regression coefficient for a variable along with the p-value. For example, the regression result for Bunch view in Hive project (Table 4.2) indicates that the coefficient of *IMC* is 0.15 and its significance level is at 89% (the p-value is 0.102), while the coefficient of *CMC* is 0.53 and its significant level is more than 99% (the p-value is 6.02e-10).

We can see from these regression models that for the projects studied, the coefficient of *CMC* is highly significant and is greater than the coefficient of *IMC* in all surrogate views except the high-level package view—which will be discussed in the next subsection. In addition, we can observe that in several instances, the attribution of *IMC* in the model is not even significant. These data support the proposition that *cross-module co-changes* have a bigger impact on the number of bugs than *intra-module co-changes*. We also observe no difference between the open-source projects and the commercial project.

Table 4.3 shows the results of the regression analysis for the ground truth architecture of Hadoop. The results of analyzing the ground truth architecture in this project are in line with those obtained using surrogate models for the other projects (i.e., CMC is highly significant and larger than IMC), thereby, giving us confidence in the validity of my conclusions.

Furthermore I compared the Spearman correlation of CMC and IMC with defects (see table 4.4). As we can see, in all of the projects, CMC has higher correlation with defects. I used the Spearman rank correlation method, since it makes no assumption about the distribution of data, and thus more appropriate for data that is not normally distributed.

Table 4.3: Regression Results for Hadoop and Using the Ground-Truth Architecture.

	Estimate	Std. Error	z value	Pr($>\|z\|$)
(Intercept)	-4.5085	0.9461	-4.77	1.9e-06
log(IMC)	0.9368	0.2614	3.58	0.00034
log(CMC)	1.8020	0.3113	5.79	7.1e-09
log(LOC)	0.0987	0.1237	0.80	0.42524

Table 4.4: Correlation Coefficients Between Defects and the Metrics for Cross-Module Co-changes (CMC), Intra-Module Co-changes (IMC), and Number of Co-changed Files (NCF). (Correlations Significant at the 0.01 Level are Highlighted)

	HBase			Hive			OpenJPA			Camel			Cassandra			System J		
	CMC	IMC	NCF	CMC	IMC	NCF	CMC	IMC	NCF	CMC	IMC	NCF	CMC	IMC	NCF	CMC	IMC	NCF
Bunch	0.39	0.24	0.22	0.33	0.28	0.31	0.30	0.13	0.15	0.20	0.13	0.07	0.33	0.31	0.24	0.49	0.05	0.49
ACDC	0.39	0.21	0.22	0.37	0.29	0.31	0.31	0.13	0.15	0.20	0.11	0.07	0.34	0.27	0.24	0.52	0.33	0.49
ArchDRH	0.38	0.27	0.22	0.33	0.23	0.31	0.26	0.18	0.15	0.21	0.1	0.07	0.33	0.06	0.24	0.52	0.18	0.49
Package High	0.36	0.29	0.22	0.31	0.28	0.31	0.21	0.21	0.15	0.19	0.16	0.07	0.32	0.26	0.24	0.51	0.21	0.49
Package Low	0.36	0.26	0.22	0.38	0.22	0.31	0.32	0.15	0.15	0.22	0.12	0.07	0.32	0.27	0.24	0.52	0.19	0.49
LDA	0.36	0.24	0.22	0.36	0.29	0.31	0.31	0.15	0.15	0.20	0.15	0.07	0.33	0.25	0.24	-	-	-

Conclusion 1: Co-changes crosscutting the system's architectural modules have more impact on defects than co-changes localized within the same architectural modules.

4.5.2 Results for RQ1.b

Up to this point, I described the effect of co-change dispersion across architectural modules using the five surrogate models. In all of the projects that I investigated, co-changes crosscutting multiple architectural modules had a stronger impact on faults than co-changes localized within the same architectural module. But which view is a better predictor of defects and should be used to analyze the effect of co-changes? The answer to this question is relevant, as it helps the practitioners understand which view should be employed for collecting the data in practice.

To that end, I calculated the Spearman correlation between *CMC* and defects in all projects using the five views. Table 4.4 summarizes correlation coefficients between defects and the *CMC* metric calculated for five different surrogate views of each project. The data show consistently similar correlation between *CMC* and defects in all surrogate architectural views except in the high-level package view, where the correlation is relatively lower than other surrogate views. I also observed that (cf. Table 4.2) in two cases—i.e., OpenJPA and Camel high-level package views—*IMC* is even greater than *CMC*. Further analysis showed that the high-level package view is not a proper representation for the architectural modules of a system due to its coarse granularity. For instance, nearly 65% of files in the Camel (version 2.9.1) are located in one of its top level packages (called "components").

The data suggest that developers can use any of the available surrogate views except the high-level packages to monitor the changes being made in the system. In fact, it means that even using the low-level package structure and not any complex reverse engineering methods can be helpful in monitoring the health of a system from its change history (e.g., identify co-changes that may indicate architectural bad smells, as further discussed in Section 4.6).

Conclusion 2: No surrogate view is conclusively better than others, as they all—except the high-level package view—produce similar results in terms of the relationship between co-change dispersion and defects.

4.5.3 Results for RQ1.c

To address the third research question—whether a co-change metric considering architectural modules has higher correlation with defects than one that does not—I compare my CMC architecture-relevant metric with the *num-co-changed-files* (NCF) metric of Shihab et al. [97] to see which one is more correlated with defects. Their metric does not take into account the notion of architectural modules.

Shihab et al. [97], in their extensive study of defect prediction, extracted 15 different metrics from three categories of (1) *traditional metrics* (e.g., file-size), (2) *co-change metrics* (e.g., num-co-changed-files) and (3) *time factors* (e.g., latest-change-before-release) to predict defects. Since some of these metrics are highly correlated, they performed a multicollinearity test to remove the metrics that have overlapping degree of impact. After removing the overlapping metrics, five were left that covered all of the three categories. One of these five metrics was *num-co-changed-files*, which indicates the total number of files a file has co-changed with. Note that NCF measures the *magnitude* of change, as opposed to whether the co-changed files were from different architectural modules or not.

I compared NCF with CMC to see which one is more correlated with defects. Table 4.4 shows the results of Spearman correlation with defects. As we can see, in all of the projects, CMC has higher correlation with defects.

To further evaluate the effect of the NCF metric, I first regressed NCF and LOC against defects, and corroborated the earlier study that NCF has a significant positive impact on defects. However, as shown in Table 4.5, when I added NCF in a regression model including my metrics (i.e., CMC and IMC), I see that the effect of *NCF* is often not statistically significant, and it does not have a positive impact on defects. This is while *CMC* remains

Table 4.5: Regression Results for Bunch View Including Num-Cochanged-Files.

	HBase		Hive		OpenJPA		Camel		Cassandra		System J	
	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)
(Intercept)	-3.20	<2e-16	-4.49	< 2e-16	-4.35	< 2e-16	-3.82	< 2e-16	-2.59	2.24e-13	-2.77	0.00044
log2(IMC)	0.04	0.495515	0.07	0.44580	0.04	0.582	0.20	5.41e-07	0.27	0.0013	-0.06	0.59429
log2(CMC)	0.74	<2e-16	0.38	0.00149	0.67	2.15e-09	0.37	1.33e-15	0.77	1.33e-09	1.00	1.00E-07
log2(NCF)	-0.17	0.000189	0.16	0.01743	-0.07	0.227	-0.15	6.67e-08	-0.40	1.04e-05	-0.27	0.26421
log2(LOC)	0.34	<2e-16	0.33	< 2e-16	0.37	< 2e-16	0.51	< 2e-16	0.22	5.71e-06	-	-

positively correlated with defects, and its effect is consistently significant across the projects.

This result is interesting, as it indicates that the type of change (i.e., cross-module versus intra-module) is more important than the magnitude of change. It also suggests that using a metric that distinguishes cross-module co-changes has the potential to improve bug prediction accuracy. The co-change differences, in particular from an architectural perspective, is a factor that has been largely ignored in the prior research.

Conclusion 3: A co-change metric that considers architectural modules have higher correlation with defects than one that does not distinguish cross-module co-changes.

4.6 Discussion

In this section I summarize the findings and the implications of my study to investigate the first hypothesis.

4.6.1 Role of Architecture in Maintenance

In Section 4.5.1, I showed co-changes that crosscut multiple architectural modules are more correlated with defects than co-changes that are localized in the same module. This could be attributed to the fact that an architectural module supposedly deals with a limited number of concerns, and thus co-changes localized within an architectural module is likely to deal with less complicated issues than those that crosscut the modules. In addition, it is reasonable to assume in a large scale software system, the developers are familiar with only a small subset of the modules, and thus the more architecturally disperse the co-changes, the more difficult it would be for the developer to fully understand the consequences of those changes on the system's behavior, and therefore more likely to make changes that induce defects.

There are also cases where dispersed co-changes, which have introduced defects, have

happened in source files without any apparent architectural dependencies. Further exploration, however, revealed the existence of indirect dependencies among these files. Metrics were shown to be effective in bringing awareness of these hidden coupling and complexities in the system's software architecture.

As an example, the *cross-module co-changes* metric helped us to discover one such case in Hbase project. This system uses ZooKeeper, an external middleware for providing high performance coordination in distributed environments. Furthermore, Hbase implements a *master-slave* architecture, where functionalities are clearly divided into separate roles of Master and Slave servers. But when looking at the recovered architecture of the system, and manually investigated the change logs of this system: I recognized that *HRegionServer.java* and *HRegion.java* files, located in the *slave* module, and *HBase.java* and *HMaster.java*, located in the *master* module exhibit very high *cross-module co-changes*, even though there is no direct dependency between them.

My analysis showed that despite the fact that these files do not have any direct method calls, they are communicating with one another through ZooKeeper. Therefore, architectural decisions impacting one module would often impact the other module, thereby bringing about the observed co-change effect. Such architectural decisions included regular *synchronization* between master and slaves, *leader election* mechanism to handle the failure of a master, and data *locking and unlocking* used to manage access to shared resources. The existence of this indirect dependency has turned this part of the system into a critical spot, exposing inherent complexities that have resulted in various bugs. I observed defects such as deadlock due to problematic implementation or modification of locking decision, performance issues due to excess synchronization between master and slaves, and various other defects as the developers tweaked the code snippets related to the leader election mechanism originally used to handle fail over of master servers. The recurring defects in this part of the system could be attributed to the lack of visible architectural dependency in the code, which my metrics could detect.

This result is useful, as it corroborates the conventional wisdom that the software architectural decisions (e.g., how a software system is decomposed into its elements) have a significant impact on the system's evolution. In addition, it underlines the impact of software architecture on open-source projects, a community that has not been generally at the forefront of adopting software modeling and architecting practices. I hope this study serves as an impetus for the open-source community to document and maintain the architecture of such systems alongside the code.

4.6.2 Building Better Defect Predictors

Co-changes have been used extensively in the past for building defect predictors [97]. My study shows that not all co-changes have the same effect on the system's quality. Moreover, in Section 4.5.3, I showed that the co-change metric (*cross-module co-changes*) has a higher correlation with defects than a co-change metric that has been used previously in bug prediction models (*num-co-changed-files*). This implies that by distinguishing between the types of co-changes, it is possible to develop more accurate defect prediction models.

4.6.3 Architectural Bad Smell Predictors

I experimented with different surrogate representations of the system's architecture in the study. My study shows that the correlation of *cross-module co-changes* and defects is statistically significant at 99% confidence interval in all data points using all of the views (recall Table 4.2). I believe these experiments could inform future research in the discovery of *architectural bad smells*, i.e., architectural choices that have detrimental effects on system lifecycle properties. One approach to identify the architectural bad smells is to leverage the metrics introduced in this paper. For instance, by collecting the number of crosscutting co-changes per architectural module over several releases of a software system, one is able to identify the architectural modules that contribute the most to crosscutting co-changes and thus likely to harbor bad smells.

4.6.4 Empirical Research

Surprisingly few empirical studies have explored the impact of software architecture on its evolution. I believe this is mainly because many open-source software projects commonly used in the empirical software engineering research do not explicitly document and maintain the architecture of the system as it evolves. Thus, an implicit contribution of my work is the research methodology, whereby in the absence of actual models, multiple surrogate models were used as approximation of the system's software architecture. Although these surrogate models inevitably pose a threat to the validity of the results (as discussed in more detail in the next section), they also present a unique opportunity for the research community to investigate and learn from the vast information available in the open-source software repositories.

The potential of applying this methodology to study other relevant questions in light of the system's software architecture are promising. For instance, it is said that *multi-component defects* (i.e., defects requiring changes to multiple components of a software system) tend to expedite architectural degeneration of a system [63]. Similarly, it is said that architectural defects could account for as much as 20 percent of all defects, but compared to other types of defects they could consume twice as much time to fix [62]. However, adequate empirical research on open-source projects has not actually verified these behaviors. I believe the research methodology followed in my work (i.e., using the reverse engineered views of the system's architecture) could pave the way for empirically investigating such hypothesized phenomena.

4.7 Threats to Validity

This section describes the main threats to validity of the findings in chapter 4

4.7.1 Construct Validity

Construct validity issues arise when there are errors in measurement. First threat to validity is in the way I link bugs with the classes in the system. The pattern matching technique that I use to find bug references in commit logs does not guarantee to find all the links. Furthermore, since I am using bug fixes, not reported bugs, I do not consider faults that are reported, but not yet fixed. There may be architectural modules with several reported defects that have not been fixed in the period of analysis, although the chances of that happening are low.

There is also a threat to validity of the results regarding the 3 months interval for data collection. However, as mentioned in Section 4.4, when I repeated the experiments using the 6 months interval, I obtained consistent results as those reported in the proposal. In fact, using equal periods for collecting co-changes and bug fixes is an approach that I have borrowed from prior research [76].

There is also a threat to validity regarding the reverse engineering methods that I used. For example, Bunch uses several heuristics in a hill climbing approach to find the clusters and therefore the clustering results may be slightly different in consecutive runs on the same project. That said, Mitchell et al. [71] have showed that the result of Bunch is mostly stable over individual runs. Moreover, I did some sensitivity analysis and observed that these differences would not have a considerable effect on the results of my study. The other reverse engineering techniques have been previously used by other researchers; I also manually examined their accuracy and usefulness of their output before incorporating them in the proposal.

One could argue our findings are not due to the recovered architectures and basically any clustering of files would have produced the same results. To assess this threat, we repeated the experiments by replacing the surrogate architectural modules with randomly constructed clusters. The results (summarized in Table 4.6) are not consistent across the projects. In HBase and Cassandra, CMC is greater than IMC, while in the other three projects we observe the reverse of that.

Table 4.6: Regression Results Using Random Clusters.

	HBase		Hive		OpenJPA		Camel		Cassandra	
	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)	Est.	Pr(> |z|)
(Intercept)	-3.70	<2e-16	-4.04	< 2e-16	-4.43	< 2e-16	-4.19	< 2e-16	-3.47	< 2e-16
log2(IMC)	0.20	0.00078	0.46	5.75e-07	0.37	1.63e-05	0.22	1.73e-08	-0.07	0.466172
log2(CMC)	0.35	1.35e-08	0.31	1.72e-05	0.13	0.0595	0.11	0.00068	0.69	3.02e-10
log2(LOC)	0.38	<2e-16	0.33	< 2e-16	0.40	< 2e-16	0.52	< 2e-16	0.18	0.000179

4.7.2 External Validity

External threats deal with the generalization of the findings. First, I intentionally chose projects that have bug-fix information in the commit logs, but this information may not be available for other projects.

The second threat is related to the projects that I used in this empirical study, since all of them are developed in Java. An interesting future work could be to replicate this study on software projects implemented in other object oriented languages, like C++.

The third threat is the way I defined Package View, which is based on package structuring of Java language and is only applicable to Java projects, although one may be able to use similar concepts in other programming languages, e.g., namespace in C#.

Chapter 5: Architectural Decay Prediction from Evolutionary History of Software

This chapter discusses the methodology to investigate the second hypothesis (See 2.2). It discusses the techniques I used followed by the results [54].

5.1 Prediction Model Construction

Figure 5.1 overviews my approach for predicting architectural quality. My approach begins with a set of *source files*, a *version control* repository, and *architectural modules* identified by an *Architectural Module Extractor* from the source files. Given those three artifacts, four *Metrics Extractors—Lifted File-Level Extractor*, *Architectural Co-Change Extractor*, *Architectural Smell Extractor*, and *Architectural Dependency Extractor*—compute 19 metrics that are used as *independent variables* for a *stepwise regression analysis*. A user selects a metric among six architectural-quality metrics to be predicted, which serves as the *dependent variable* inputted to the stepwise regression analysis. The result of regression analysis is a prediction model for the selected quality metric. Each prediction model produced by our approach utilizes independent variables of release k of system s and predicts the selected architectural-quality metric for $k + 1$ of system s.

In the remainder of this section, I describe the major parts of my approach: the techniques I leveraged to obtain architectural modules, my selected regression models, the six quality metrics to be predicted, and the metrics extracted and used as independent variables.

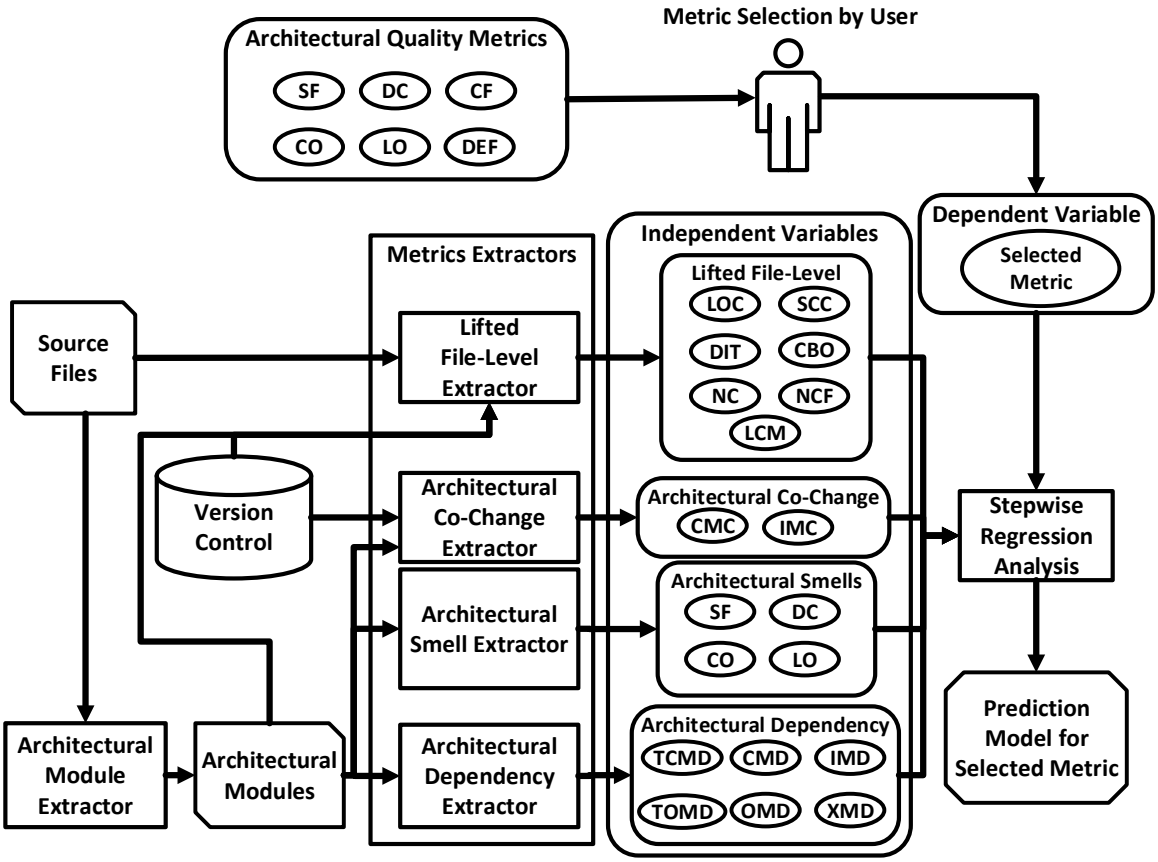

Figure 5.1: Overview of My Approach for Architectural-Quality Metric Prediction.

5.1.1 Obtaining Architectural Modules

I consider two different techniques for recovering architectural modules, which are used by *Architectural Module Extractor*. As a result, I obtain multiple architectural views [59], allowing an engineer to obtain architectural-quality metrics from different perspectives. This maximizes the possibility of identifying architectural-quality problems throughout a software system. Note that an architecture-recovery technique can be substituted for a ground-truth architecture verified as correct by a software system's architects. In such a situation, my prediction models would likely achieve better performance, since they would not need to correct for improperly recovered modules.

The package structure of a system can be treated as a proxy for the decomposition of the

system into architecturally significant elements, as packages are created by the developers of the system. In fact, package structuring has been used as a decomposition reference in prior research [10, 25, 52]. Packages and their sub-packages can be represented in a tree structure corresponding to the packaging hierarchy. Each leaf of the tree is a Java class contained in a package, which itself may belong to a higher level package. The root of the tree is the top-level package.

In section 4, I showed that high-level packages are not suitable for studying the evolution of architecture—due to the coarse granularity—and low-level packages should be used instead. Therefore, I use low-level packages in this part of study. In low-level packages, architectural modules correspond to packages that only contain Java classes and no sub-packages.

In addition to packages, I include a semantic view of modules obtained using an architecture-recovery technique called *Architectural Recovery using Concerns (ARC)* [40, 44, 60], which utilizes hierarchical clustering and information retrieval to produce modules. ARC leverages a statistical language model, Latent Dirichlet Allocation (LDA) [11], to represent each source file of a system as textual documents consisting of concerns, which are extracted from the identifiers and comments of each file. A concern could be a role, concept, or responsibility of a system. The number of modules recovered by ARC is selectable by an engineer, enabling the consideration of recovered modules at a high level and low level, just as in the case of packages.

Once modules have been identified or recovered, I must be able to determine which module m_k in release k is the same module m_{k+1} in release $k+1$. This determination allows us to make predictions for m_{k+1} based on our metrics for m_k. I leverage a technique described in prior work that traces modules across releases based on the degree of overlap among them [60].

I use a metric, $c2c$, that calculates the similarity between modules [40]. $c2c$ is calculated as follows:

$$c2c(m_k, m_{k+1}) = \frac{|m_k \cap m_{k+1}|}{|m_k|} \times 100\%$$

For this study, normalizing over $|m_k|$ ensures that I identify the module in release $k+1$ most similar to module m_k. To use $c2c$ to identify similarity among modules, I must select a threshold for similarity expressed as a percentage. I utilize a 50% similarity threshold, which means that a module m_{k+1} is the same as a module m_k if $c2c(m_k, m_{k+1}) > 50\%$. Consequently, m_{k+1} is similar to module m_k if they have mostly the same files in the module.

5.1.2 Regression Analysis Selection

I constructed the prediction models in this study using the releases of each project. I use three well-known regression models in this study and compare the results: linear regression (LR), negative binomial regression (NBR), and random forest (RF). I used the *MASS* library in R [1] for building LR and NBR and the *randomForest* library for RF [2].

Although LR is popular and more widely used in the literature, some have argued that NBR is a more appropriate regression model for defect prediction [81]. Unlike LR, NBR makes no assumptions about the linearity of the relationship between the variables, or the normality of the variable distributions. NBR is applicable to non-negative integers and, more importantly, can be used for over-dispersed count data (i.e., when the conditional variance of the data exceeds the conditional mean) [23]. I also chose RF since it has been shown to perform best for software defect prediction [61], making RF potentially suitable for predicting architectural quality. For NBR, we use the log_2 transformation of our metrics to reduce the influence of extreme values, similar to prior work [23].

I do not want my prediction metrics to exhibit multicollinearity, a phenomenon where prediction metrics are correlated, since this can cause my prediction models to become unstable [36]. To avoid the multicollinearity problem, I use stepwise regression to build the models. I leverage the *stepAIC* function in the *MASS* library of R for this purpose. *Akaike Information Criteria (AIC)* is a commonly used static measure for goodness of fit. Models

can be built in two ways: forward and backward. Forward stepwise regression begins with no variable in the model. The variable that improves the model the most is identified and added to the model. The process continues until none of the remaining variables can improve the model. Backward stepwise regression starts with the full model, improves the model by deleting variables, and repeats this deletion until no further improvement is possible. To determine the optimal model, I ran both forward and backward stepwise regression. I used stepwise regression when building models with LR and NBR. I utilized all of the metrics when building models using RF because it works well with a large number of independent variables [86], where our model includes only 19 such variables.

5.1.3 Dependent Variables

I selected the following six metrics that serve as representations of architectural decay: the number of defects in a module; four architectural-smell metrics, where each metric indicates whether a module has a specific type of smell; and a metric that indicates a module's quality in terms of coupling and cohesion. Each of these metrics is a dependent variable for a single architectural-quality prediction model.

The number of defects per module is determined by summing up the defects in each file contained within an architectural module.

The coupling and cohesion of a module is a strong indicator of the module's quality. To that end, I select a metric, *Cluster Factor (CF)* [71], used widely in previous architectural studies [40, 71, 85, 107] that represents the coupling and cohesion of a module. We calculate CF for a module m as follows:

$$CF_m = \frac{\mu_i}{\mu_i + 0.5 \times \sum_j \epsilon_{ij} + \epsilon_{ji}}$$

where μ_i is the number of dependencies between entities within a module, and $\epsilon_{ij} + \epsilon_{ji}$ is the number of dependencies between module i and module j.

The presence or absence of architectural bad smells in a module may inform my prediction models as to the future occurrence of architectural decay. To that end, I select four architectural smells for my study that represent structural or semantic maintainability problems of a module. Each smell falls into one of two categories: *concern-based smells* or *dependency-based smells*. Concern-based smells are caused by inappropriate or inadequate separation of concerns; dependency-based smells arise due to module interactions resulting from code relationships among entities within a module.

I identify the following smells that a module may suffer from, which have been studied in previous work [39, 42, 43].

- *Scattered Functionality (SF)* is a concern-based architectural smell that describes a system in which multiple modules are responsible for realizing the same high-level concern, while some of those modules are also responsible for additional, orthogonal concerns.

- *Concern Overload (CO)* is a concern-based architectural smell that occurs for a module when it implements an excessive number of concerns.

- *Dependency Cycle (DC)* is a dependency-based architectural smell that occurs when a set of modules are linked in such a way that they form a cycle, causing changes to one module to possibly affect all other modules involved in the cycle.

- *Link Overload (LO)* is a dependency-based smell that occurs when a module is involved in an excessive number of dependencies to other modules. A module can have an excessive number of incoming links, outgoing links, or both.

To represent each of these smells as an architectural-quality metric to be predicted, I create a binary metric for each smell: s_{sf}, s_{co}, s_{dc}, and s_{lo}. If a module m has a smell s, then $s = 1$. Otherwise, $s = 0$. For example, if a module m_1 has CO, then $s_{co} = 1$ for m_1. As another example, if module m_2 is involved in a DC with other modules, $s_{dc} = 1$ for m_2.

5.1.4 Independent Variables

I use four types of metrics extractors to obtain a combination of *file-level* and *architectural-level* metrics for predicting architectural quality. Many prediction models from existing literature have focused on predicting software defects [19, 28, 67, 81]. I chose a subset of metrics from the prior literature, particularly at the file level, as independent variables for prediction, since they may be indicators of architectural problems.

Lifted File-Level Extractor obtains the following file-level metrics:

- The *lines of code (LOC)* of a file is a measure of the size of a file determined by counting the number of non-empty non-comment lines.
- *Sum cyclomatic complexity (SCC)* of any structured program with only one entry point and one exit point is equal to the number of decision points contained in that program plus one.
- The *depth of inheritance tree (DIT)* is the depth of a class within an inheritance hierarchy calculated as the maximum number of nodes from the class node to the root of the inheritance tree.
- *Coupling between objects (CBO)* for a class C is the number of other classes to which C is coupled. Class A is coupled to class B if class A uses a type, data, or member from class B.
- *Lack of cohesion in methods (LCM)* is calculated as 100% minus average cohesion for class data members. Average cohesion is calculated as the percentage of pairs of methods in a class that have at least one field in common. A lower percentage means higher cohesion between class data and methods.
- *Number of changes (NC)* is the number of times that a file is committed to a repository.
- *Number of co-changed files (NCF)* is the number of other files that a file f is changed with [97].

To represent file-level metrics at the module-level, I *lift them up* to the architectural level by summing up the values of each file-level metric across all files inside each module. The resulting sum is then used as a representation of each file-level metric for a module. For example, in the case of SCC, a module m with four files can have the following SCC values, one for each file: 2, 5, 6, and 9. The SCC for module m is the sum of all SCCs of its constituent files, i.e., 22. It is worth nothing that it is possible to use other approaches for lifting up the metrics, such as considering the maximum value of each file-level metric across all files inside each module or calculating the weighted average. I used the sum of values since it has been used for predicting defects for packages [50, 112].

Among our architectural metrics, I include metrics involving *co-changes* between modules that are extracted by *Architectural Co-Change Extractor*. Co-changes are process metrics that represent modifications that occur simultaneously within or across modules. Section 4 demonstrated that architectural co-changes correlate with defects. Consequently, architectural co-change metrics may potentially improve my prediction models. I select the following architectural co-change metrics:

- *Cross-module co-changes (CMC)* is the number of co-changes for a file, where the co-changes are made across more than one architectural module.
- *Inner-module co-changes (IMC)* is the number of co-changes for a file, where there is at least another co-changed file in the same architectural module.

A number of our selected architectural-quality metrics are based on dependencies between modules, which are code relationships among source-level entities within a module (e.g., method invocations, field accesses, import statements, etc.). To predict architectural quality based on such dependencies, *Architectural Dependency Extractor* obtains module-dependency metrics.

I consider two methods for measuring the dependencies between modules. The first method models the dependencies as a binary variable, meaning that I only measure whether a module has a dependency on another module. The second method is to count all of the

dependencies between the modules, which considers the number of dependencies between the files inside each of the modules. Using these two methods, I select the following dependency-based metrics:

- *Incoming module dependency (CMD)* is a binary metric for a module m_1 with a value of 1 if there is at least one dependency from another module m_2 to m_1, and 0 otherwise.
- *Outgoing module dependency (OMD)* is a binary metric for a module m_1 with a value of 1 if there is at least one dependency from m_1 to another module m_2, and 0 otherwise.
- *Total incoming module dependencies (TCMD)* is the total number of dependencies to a module m_1 and originating from other modules in a software system.
- *Total outgoing module dependencies (TOMD)* is the total number of dependencies from a module m_1 to other modules in a system.
- *Internal module dependencies (IMD)* is the total number of dependencies among all files within a module.
- *External module dependencies (XMD)* is the total number of incoming and outgoing dependencies of a module.

The existence of architectural smells in a module may indicate further architectural decay in the future for that module. For example, a module with CO may be more likely to exhibit LO in the future. As another example, LO may be an indicator of future reductions in a module's CF. To that end, *Architectural Smell Extractor* identifies the four architectural smells described in Section 5.1.3 and computes the corresponding metrics.

5.2 Experimental Setup

To evaluate my prediction models, this section discusses the experimental setup I use to answer our research questions.

5.2.1 Projects Studied and Data Collection

My experimental subjects include five projects, listed in Table 5.1. They are all written in Java and are maintained by Apache Software Foundation (ASF). However, they vary in their sizes and application domains, allowing me to draw broader conclusions.

To enable prediction of architectural quality, I collect data about defect fixes and metrics at both the code and architectural levels. I utilize different tools for that purpose.

I obtain code-level metrics per file and for each release. The first five file-level metrics (*LOC, SCC, DIT, CBO and LCM*) are measured using UNDERSTAND from Scitools[1] for each release.

The change metrics (*NC, NCF, CMC and IMC*) are calculated by processing the developer commits from an SVN repository and extracting the groups of files in the same commit transaction that have been modified together (i.e., co-changes). I use *SVNKit*, a Java toolkit providing APIs to subversion repositories.

To obtain architectural metrics, I leverage *Architecture Recovery, Change, And Decay Evaluator (ARCADE)* [39, 60], a workbench containing tools for addressing architectural decay. Specifically, ARCADE consists of algorithms for detecting architectural smells and computing architectural dependency information, enabling the extraction of four selected architectural smell metrics (*SF*, *CO*, *DC*, and *LO*) and six architectural dependency-based metrics (*CMD*, *OMD*, *TCMD*, *TOMD*, *IMD*, and *XMD*).

In the ASF software repositories and, by extension, the projects studied in this dissertation, the commits that are defect fixes are identifiable since defects are referred to by a project name and defect number in SVN commit logs. For example, all of the defect fixes in HBASE begin with *HBASE-<bug number>* (e.g., *HBASE-3172*). This enabled me to find all defect fixes by just parsing the log of commits in SVN and finding the keyword *HBASE-<bug number>*. To determine the number of defects for each module, I sum up the number of defect fixes in all files within each module.

[1] http://www.scitools.com/

Table 5.1: Studied Projects and Release Information.

Project	Description	Releases	SLOC
HBase	Distributed Scalable Data Store	0.1.0 ,0.1.3 ,0.18.0 ,0.19.0, 0.19.3, 0.20.2, 0.89.20100621, 0.89.20100924, 0.90.2, 0.90.4, 0.92.0	39K-246K
Hive	Data Warehouse System for Hadoop	0.3.0, 0.4.1, 0.5.0, 0.6.0, 0.7.0, 0.8.1	66K-226K
OpenJPA	Java Persistence Framework	1.0.1, 1.0.3, 1.1.0, 1.2.0, 2.0.0-M3, 2.0.1, 2.1.0	153K-407K
Camel	Enterprise Integration Framework	1.6.0 ,2.0.M ,2.2.0 ,2.4.0, 2.5.0, 2.6.0, 2.7.1, 2.8.0, 2.8.3	99K-390K
Cassandra	Distributed Database Management System	0.3.0 ,0.4.1 ,0.5.1 ,0.6.2, 0.6.5, 0.7.0, 0.7.5	50K-90K

5.2.2 Data Splitting and Evaluation Metrics

I first discuss the splitting strategy I select for training my models and testing them. I then cover the two criteria I chose to evaluate the performance of my prediction models: predictive power and ranking.

Data Splitting. In order to evaluate the performance of the models, I use *data splitting*, a commonly used evaluation technique, where a data set is divided into subsets for building and evaluating the model. Specifically, I randomly split the data set as follows: two-thirds of the data are used to train and build the prediction model, and one-third of the data is used to test the performance of the model. For generating a stable analysis result, I conduct the experiments 100 times and use the average results over all those experiments. I train the model on a given release and evaluate its performance on the next release. Specifically, I calculate all the metrics described in Section 5.1.4 on the k-th release and use them as independent variables of our models. I then predict each of the architectural-quality metrics for the k+1-th release. Consequently, each architectural-quality metric is a dependent variable for each prediction model.

Predictive Power. I assess the predictive power of a model by selecting an appropriate performance measure. I considered a variety of measures often utilized to evaluate the performance of predictive models for software-engineering purposes. I will briefly discuss some commonly used measures—*accuracy*, *precision*, and *recall*—and why they are undesirable for my study. I then follow that discussion with an introduction and justification of my chosen measure for predictive performance: *area under the curve (AUC)* of the *receiver operating characteristic (ROC)*.

Precision and recall are pairs of performance measures commonly used together for prediction models. Precision is a measure of a model's ability to predict modules without falsely marking them as having low architectural quality. Recall is a measure of a model's ability to correctly predict all modules with low architectural quality. Precision is defined as $\frac{tp}{tp+fp}$. tp is the number of true positives, where a true positive occurs for a module when it is correctly predicted as having low architectural quality. fp is the number of

false positives, where a false positive occurs for a module when it is predicted having low architectural quality when it, in fact, does not. Recall is defined as $\frac{tp}{tp+fn}$. fn is the number of false negatives, where a false negative occurs for a module when it is predicted as not having low architectural quality, even though it, in fact, does. A prediction model should have a high precision and recall; however, increasing one often decreases the other.

Accuracy is the proportion of correct predictions, and is defined as $\frac{tp+tn}{tp+tn+fp+fn}$. tn is the number of true negatives, where a true negative for a module occurs when a model correctly predicts the module as not having low architectural quality. However, accuracy can be a bad performance measure for imbalanced data [99]. For example, if we only have a few defective modules in our data set, a model that considers all modules as clean would have a high accuracy.

Precision, recall, and accuracy all require the arbitrary setting of discrimination thresholds to declare a module as having low architectural quality. To avoid arbitrary setting of thresholds in our experiments, I utilize AUC of ROC as the performance measure for comparing prediction models, as suggested by [61], and further described below.

Receiver operating characteristic (ROC) is a curve that plots true-positive rates (y-axis) against false-positive rates (x-axis) for all possible thresholds between 0 and 1—precluding the need to arbitrarily set thresholds. AUC is a scalar performance measure derived from ROC and is the area enclosed by the curve and the x-axis. AUC separates predictive performance from class and cost distributions, which are based on characteristics of projects. The best possible model is a curve close to $y = 1$ with AUC of 1.0; a random classifier would obtain AUC of 0.5. In code-level defect prediction literature, an AUC of 0.7 or above is considered a high level of performance for a prediction model [61, 67]. Given the similarity of architectural decay and defects, I also consider AUC of 0.7 and above as a high level of performance for architectural-quality prediction.

For illustration, Figure 5.2 shows an ROC curve corresponding to one of our models for predicting defects in architectural modules of OpenJPA project. By choosing a different discrimination threshold for declaring a module defective, the prediction model would produce

a different performance, as shown in this curve. Rather than reporting the results using an arbitrary threshold, I use AUC to holistically compare the classification performance of different prediction models under all possible thresholds.

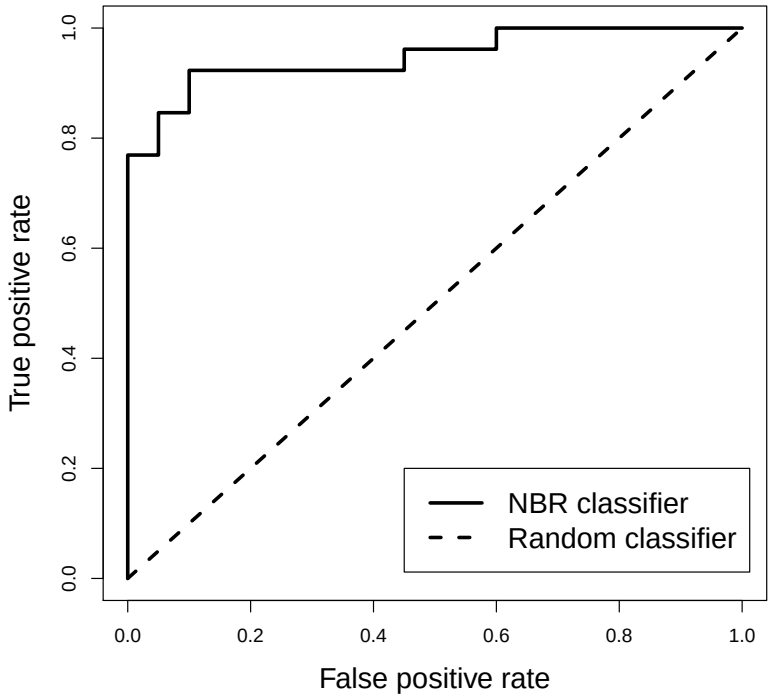

Figure 5.2: ROC Curve for Defect Prediction.

My approach for evaluating the prediction models is orthogonal to how the engineers would use the models in software projects. In practice, the engineer can choose a discrimination threshold that achieves the desired balance of precision and recall based on the characteristics of a project. For instance, if a project is understaffed and there are insufficient resources to thoroughly review the system's architecture/code, the engineer may choose a threshold that achieves a higher precision and a lower recall, meaning less wasted effort investigating false positives, at the expense of not fixing all architectural issues in time. On the other hand, if a project has the necessary staff and resources to thoroughly review

the system's architecture and code, the engineer may choose a threshold that achieves a lower precision and a higher recall, meaning more wasted effort of investigating false positives, but increased likelihood of fixing all architectural concerns. As another example, in a safety-critical software project, the engineers may choose to use thresholds that maximize the recall to reduce architectural decay factors, and thereby improve the quality of software, as much as possible.

Ranking. Determining the modules with the lowest architectural quality allows engineers to prioritize their efforts to those modules first. To that end, I assess if a model can correctly predict the order of modules according to their architectural-quality metrics. Ranking is not applicable to architectural smells since they are binary variables. However, we can obtain ranking results for defects and CF. In defect ranking, we build the prediction models using data splitting, predict the number of faults for each module, and compare the ordering of the predicted defect numbers with actual defect numbers using Spearman correlation. Similarly, we predict CF values for each module and compare the ranking of predicted CF values with the ranking of actual CF values.

We consider a Spearman correlation greater than 0.4 that is statistically significant at the 0.01 level to be a reliable ranking of modules. A correlation of 1.0 denotes a perfect ranking. Previous work on code-level defect prediction has considered Spearman correlation values greater than 0.4 to be noteworthy [111, 112]. Given the similarity of predicting code-level defects and architectural decay, this consideration is sensible for our prediction models. Note that all the Spearman correlations that we report are significant at the 0.01 level.

5.3 Experimental Results

Given my approach and the experimental design described in the previous sections, I now discuss the results obtained for each of the research questions. I begin by presenting some information about the modules. I continue by assessing the overall performance of my prediction models for each architectural-quality metric. I follow that study by assessing the degree of change for each architectural-smell metric. Afterwards, I focus on prediction

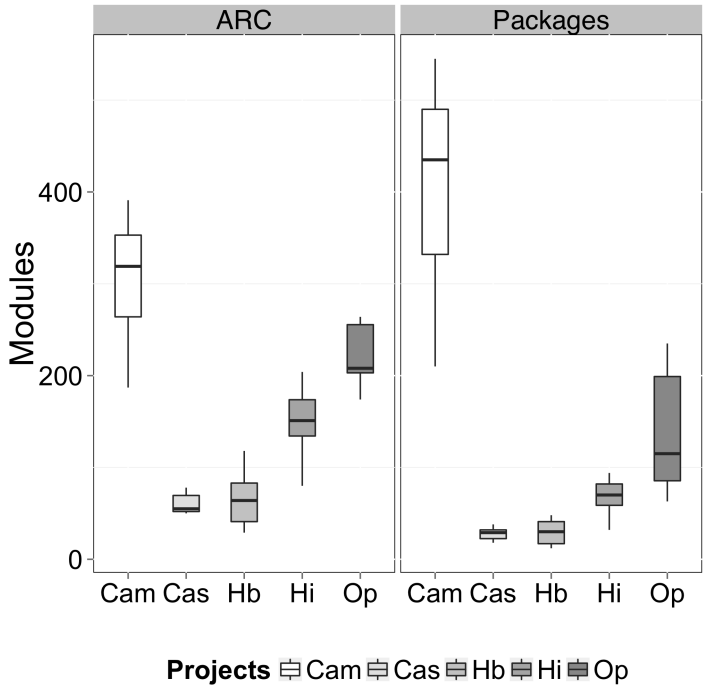

Figure 5.3: Number of Architectural Modules.

results for smell emergence. Lastly, I determine the metrics that best predict architectural quality.

I start by presenting some information about the data in my study. Figure 5.3 shows the number of modules in different projects. The projects are marked in x-axis as 1: Camel, 2: Cassandra, 3: HBase, 4: Hive, 5: OpenJPA using both ARC (A) and packages (P). As it is shown the number of modules varies across projects and projects with more LOC has more clusters. Also in some projects (e.g. Cassandra) the number of modules don't change much while in some others (e.g. Camel), we see different number of modules across releases.

Figure 5.4 shows the percentages of existence of each of architectural-quality metrics across releases (D stands for defects). As it is shown (33-58)% of ARC's modules have defects while (47-80)% of packages have defects across different releases. Also among the architectural smells, most of the modules shows to have DC.

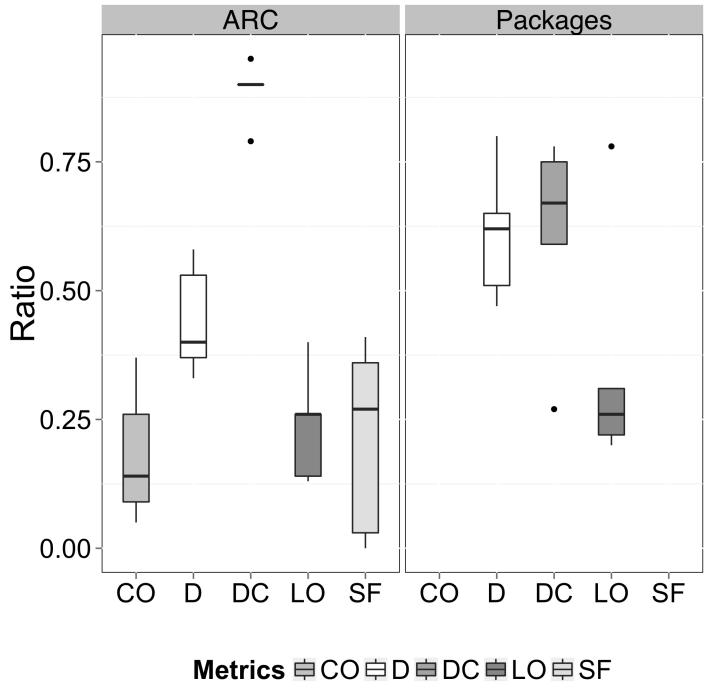

Figure 5.4: Percentages of Existence of Architectural-Quality Metrics.

5.3.1 Results for RQ2.a

I now discuss the results obtained for each of the research questions in chapter 2.2.

> ***RQ2.a:*** *What is the performance of each prediction model for the different architectural quality metrics?*

I first assess my model's ability to predict whether a module has at least one defect, which I refer to as *defect existence prediction*. Figure 5.5 shows AUC results for defect existence prediction for RF (F), LR (L), and NBR (N), using both ARC and packages. The results show that the prediction performance of NBR is higher than LR and RF. Particularly in the case of NBR, our models predict module defectiveness with AUC of at least 0.76.

I further observe that AUC results for module-level defect prediction are higher for packages than ARC. This higher performance for packages may result from the fact that

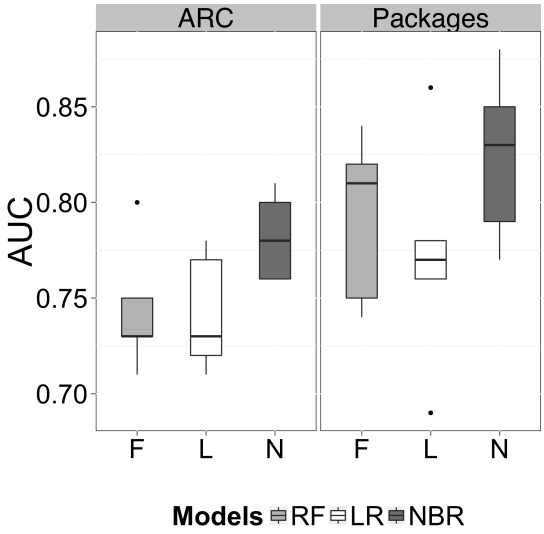

Figure 5.5: AUC Performance Defects.

no special technique is needed to obtain packages and are, thus, less susceptible to error. However, given that my models can obtain at least 0.76 AUC, they exhibit resilience to errors that may exist in ARC.

Only predicting which modules have defects in future releases does not help in prioritizing modules for defect analysis and removal. Particularly, roughly 50% of modules in our study tend to have defects, which provides engineers with little information as to which modules should be allocated more maintenance resources. To address this issue, my models can predict the amount of defects a module may have, rather than simply whether a module has a defect. Predicting the magnitude of a module's defectiveness allows an engineer to prioritize modules for defect analysis and removal.

I assess my model's ability to predict the extent of a module's defectiveness by using Spearman correlation to compare the actual ranking of defective modules with our model's predicted rankings. Figure 5.6 shows these results. As in the case of defect existence prediction of modules, NBR outperforms LR and RF: Prediction for ARC modules obtains Spearman correlation of 0.48-0.69; for packages, my models obtain a spearman correlation of 0.62-0.73. Similar to defect existence prediction for modules, ranking results are higher for

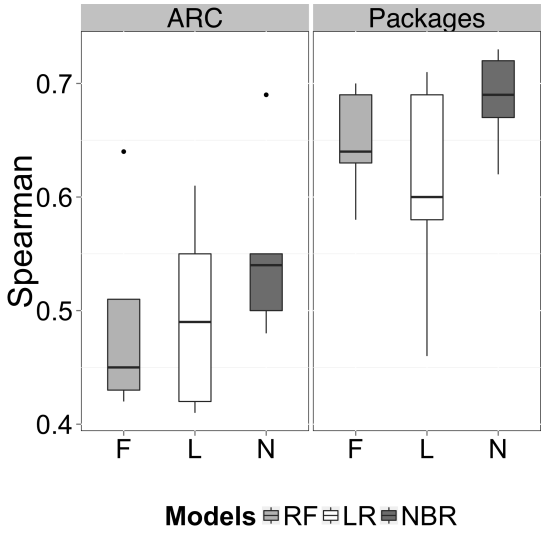

Figure 5.6: Spearman Correlation for Ranking Defective Modules.

packages than ARC. Again, this is likely due to error introduced by ARC when recovering modules.

For smell prediction, I determine whether my models can predict the occurrence of different types of smells. To that end, I utilize AUC as our performance measure. Figure 5.7 shows the AUC results for predicting smells in ARC. I have the results of all four smells from ARC; however, two of the smells are concern-based and only applicable to ARC. Thus, for packages, I have results for DC and LO only. As shown in Figure 5.7, I can predict the occurrences of smells in modules with a high AUC of 0.84 or above. Furthermore, LR, NBR, and RF obtain similar prediction results, in terms of AUC, for smells. Overall, prediction results are better for packages than ARC modules, which is consistent with the prediction results for defects.

The overwhelming majority of modules in projects have low architectural quality as measured by CF. We consider a module m as having a low CF when CF < 0.3 for m. This CF value indicates that the vast majority of m's dependencies are with entities outside of m, as opposed to within m, indicating high coupling and low cohesion. Given that modules mostly have low CF values, it is particularly important that engineers identify the modules

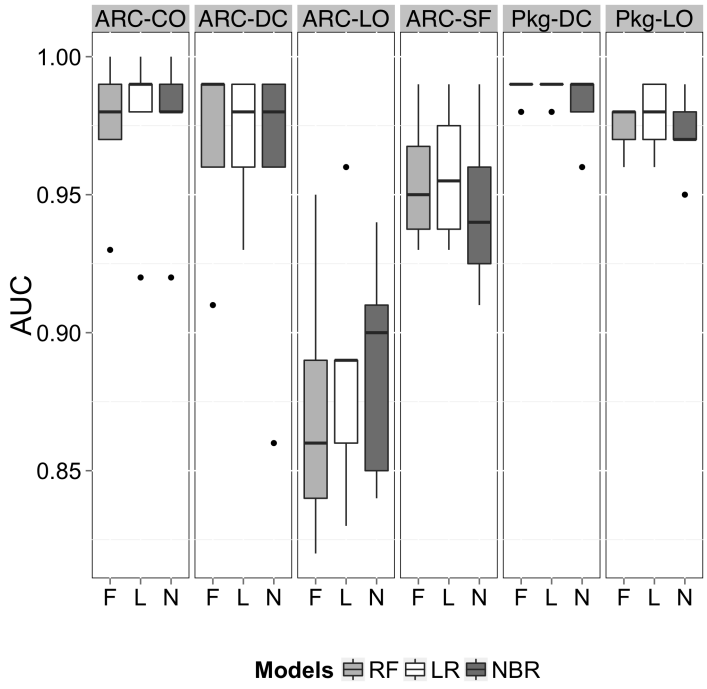

Figure 5.7: AUC Performance Architectural Smells.

with the worst CF. With such information, engineers can allocate maintenance resources to those modules first. To that end, we focus on the ranking results of CF, as opposed to AUC results.

Figure 5.8 depicts the ranking results for CF values compared using Spearman correlation. For both ARC and packages, NBR and RF perform similarly, achieving more than 0.7 correlation, with RF performing slightly better than NBR. Both models outperform LR. The superior performance of NBR and RF is significantly more pronounced for ARC. This difference in CF may be due to ARC ignoring dependency-based coupling and cohesion, which is what CF is based on.

To illustrate how the results of this research might be used by the engineers, I describe one of the prediction models from Figure 5.8 in more detail. I show the CF prediction results for a subset of packages in HBase version 0.92. Table 5.2 shows the actual values of CF for packages, the predicted value of CF, and also the corresponding ranking. As shown,

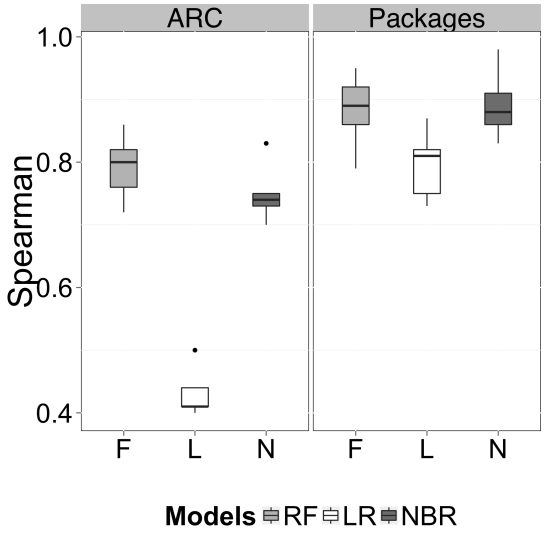

Figure 5.8: Spearman Correlation Cluster Factor.

the predicted values of CF is very close to the actual values of CF. Out of 15 modules, 12 modules are ranked correctly by the prediction model, while for the 3 remaining modules (i.e., handler, executor, and replication) the actual and predicted rankings are quite close. Engineers could use such information to identify architectural problems (e.g., identify the modules with low CF) and prioritize their effort (e.g., refactor the modules with lowest CF).

In summary, the results show that my models can effectively predict the different architectural-quality metrics. For most cases, NBR provides superior results and is the best overall model for predicting architectural quality.

5.3.2 Results for RQ2.b

Next I report the results of the amount of change for each architectural-smell metric.

> ***RQ2.b:*** *What is the amount of architectural change across releases for each architectural-smell metric?*

Figure 5.9 shows the percentages of changes across all releases and systems for each

Table 5.2: Prediction of CF for Packages in HBase (Version 0.92).

Package Name	CF	Predicted CF	Rank of CF	Rank of Predicted CF
org.apache.hadoop.hbase.mapreduce	0.11	0.11	10	10
org.apache.hadoop.hbase.filter	0.27	0.31	15	15
org.apache.hadoop.hbase.io.hfile	0.25	0.28	14	14
org.apache.hadoop.hbase.client.coprocessor	0.01	0.02	2	2
org.apache.hadoop.hbase.mapred	0.13	0.14	11	11
org.apache.hadoop.hbase.io	0.03	0.03	3	3
org.apache.hadoop.hbase.master.handler	0.05	0.05	5	6
org.apache.hadoop.hbase.regionserver	0.18	0.20	12	12
org.apache.hadoop.hbase.executor	0.04	0.04	4	5
org.apache.hadoop.hbase.rest.client	0.11	0.10	9	9
org.apache.hadoop.hbase.thrift.generated	0.00	0.01	1	1
org.apache.hadoop.hbase.replication.regionserver	0.09	0.08	8	8
org.apache.hadoop.hbase.replication	0.05	0.04	6	4
org.apache.hadoop.hbase.rest	0.22	0.24	13	13
org.apache.hadoop.hbase.util.hbck	0.06	0.06	7	7

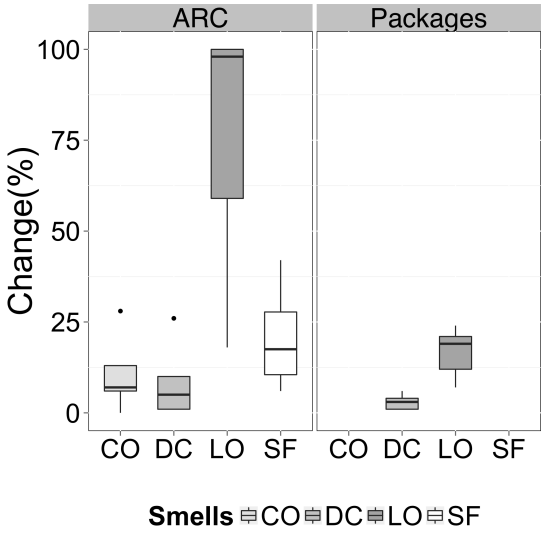

Figure 5.9: Percentages of Changes of Architectural Smells.

architectural smell. Although all types of architectural smells change across releases, the amount of change varies: SF, DC, and CO exhibit relatively little change; LO changes drastically across all releases of our systems.

The results for smell changes indicate that, for the selected systems, modules that suffer from concern-based smells (CO and SF) tend to retain those smells across releases—with little addition or removal of such smells afterwards. At the same time, change for SF is significantly higher than CO.

For each dependency-based architectural smell (DC and LO), change across releases varies significantly. The amount of change represented by LO varies drastically between ARC and packages. This difference is likely due to the fact that ARC does not take dependencies into account, which are used to compute LO.

DC exhibits a similar amount of architectural change, across releases and systems, for both ARC and packages. Furthermore, the amount of change for DC is quite low (largely between 1%-26%). Consequently, across releases, the same modules tend to be involved in a DC, for our selected systems.

Overall, I find that architectural smells do exhibit significant change worth predicting.

However, we would like to determine if my prediction models can forecast a particular type of architectural-quality change, i.e., smell emergence, so that engineers can possibly take action before a smell occurs—resulting in possible savings of future time and effort. To that end, I examine the results for my next research question:

5.3.3 Results for RQ2.c

> ***RQ2.c:*** *Can we effectively predict architectural-smell emergence between two consecutive releases?*

As part of answering this research question, I first assess the frequency of smell emergence. Figure 5.10 shows the percentages of smell emergence in architectural modules across all systems and releases. LO is the most frequent type of smell emergence with a median of 9% occurring for modules. SF and DC smell emergence occurs less than 5% in ARC; DC smell emergence does not occur in most projects. Although smell emergence occurs infrequently, this phenomenon is intuitively difficult to predict and preventing its occurrence may reduce future maintenance issues.

To build a model for predicting smell emergence cases, I created new binary variables for each smell: se_{co}, se_{dc}, se_{lo}, se_{sf}. se variables are equal to one whenever the value of the corresponding smell is 0 in the current release and 1 in the next release—meaning that the smell does not exist in the previous release, but it emerges in the next release. I created models for predicting smell emergence using these new dependent variables.

Figure 5.11 shows the AUC prediction results for smell emergence for all systems and releases. Although the number of smell-emergence instances are low, we predict those instances with AUC of 0.75-0.97 using NBR.

The performance of RF drops considerably for smell-emergence prediction compared to LR and NBR. This occurs because RF can lose significant performance when a dataset is extremely imbalanced [20]; however, stepwise regression with LR and NBR are less susceptible to imbalanced data.

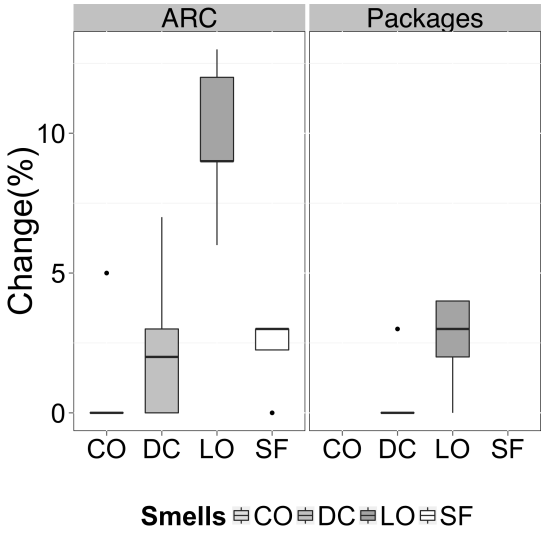

Figure 5.10: Percentages of Extreme Changes of Architectural Smells

In summary, my models can predict smell emergence—and architectural-quality metrics in general—with high performance. To obtain such prediction models, it is important to identify the metrics that best improve our prediction models. I make that determination as I answer the following research question:

5.3.4 Results for RQ2.d

> ***RQ2.d:*** *What are the important metrics for predicting each architectural-quality metric?*

My previous results show that prediction models using NBR significantly outperform LR and RF in the majority of cases. Consequently, to answer RQ2.d I focus on identifying the best metrics, obtained through stepwise regression, for NBR. I produced 50 prediction models for architectural quality using NBR. These were obtained from the combination of five systems, two architectural views (ARC and packages), and six dependent variables (defects, SF, CO, DC, LO and CF), where SF and CO are only applicable for ARC. Similarly, I constructed several prediction models for smell emergence. Due to the number of prediction models and

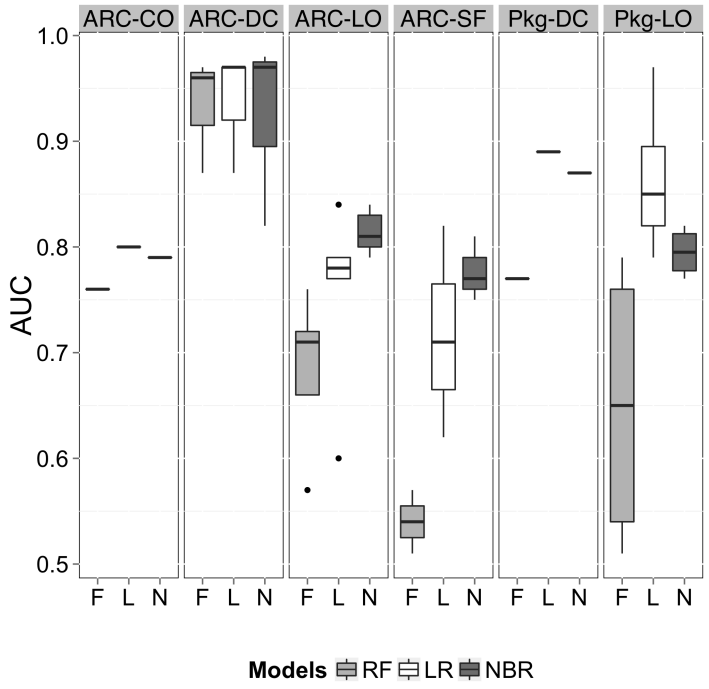

Figure 5.11: AUC Performance for Architectural Smell Emergence.

space constraints, I do not report the coefficient values and significance level of all of the independent variables in each model.[2]

Table 5.3 showcases the factors, i.e., independent variables, that contribute to prediction models for each quality metric: Each column represents an independent variable; each row represents a dependent variable. Factors for smell-emergence models are denoted by "-SE." Values in the table depict the number of times each independent variable contributes to a prediction model. The maximum value in each cell is 10 (the combination of two architectural views and five systems). However, for concern-based architectural smells (SF, CO, SF-SE and CO-SE), 5 is the maximum value, because the package view does not include such smells. For example, LOC contributes to all models for predicting defects and, thus, is included in all 10 models.

[2]Readers may find the study artifacts, including the prediction models and results, at: `https://seal.ics.uci.edu/projects/decayprediction`

Table 5.3: Factors Contributing to Each Model

	LOC	SCC	DIT	CBO	LCM	NC	NCF	CMC	IMC	SF	CO	DC	LO	CMD	OMD	TCMD	TOMD	IMD	XMD
Defects	10	2	4	7	1	8	4	6	7	0	3	5	1	1	6	1	4	4	6
SF									1	4			1						
CO								1			5				1	1			
DC												10							
LO	3		1	4	1	3	3		1	1	2	2	9	2	1	4	3	4	4
CF	1			1												2	1	9	5
SF-SE	1	2	1	1			2		1				2	1	1	1			1
CO-SE	1							1		1		1				1			
DC-SE	1	3		1	1		1	3			1			2	3	3	1	2	2
LO-SE	4		1	4	1	3	1	2	2		1	2	1	4	3	3	2	3	8

A wide variety of metric types, from all categories, are important factors–with values of at least 5—for predicting defects: lifted file-level metrics (LOC, CBO, and NC), architectural co-changes (CMC and IMC), architectural smells (DC), and architectural-dependency metrics (OMD and XMD).

In general, for three of the four types of architectural smells (SF, CO, and DC), the important factor for predicting those smells is if the smell exists for a module in the current release. For example, if a module has CO, it is likely to continue having CO in the next release. However, a wider variety of metrics are important factors for predicting LO.

Overall, these smell results indicate that architectural smells are rarely restructured, meaning that smell-oriented decay tends to remain in a system once it emerges. This result further motivates the need to predict smell emergence and prevent smell occurrence.

The factors for predicting CF are mainly from the architectural-dependency metrics. Given that CF is a measure of coupling and cohesion based on architectural dependencies, this result is intuitive and expected.

The important factors for predicting smell emergence are starkly different from predicting the general case of architectural quality: A wide variety of metrics predicted each type of smell emergence. This result indicates that smell emergence originates from a complex set of factors that warrants further research.

Overall, my results indicate that all categories of independent variables are important for predicting architectural quality. Unlike previous work for predicting defects in packages [77, 94], which only used lifted file-level metrics, I show that both lifted file-level metrics and architectural metrics are important for predicting architectural quality. Futhermore, stepwise regression using NBR provides the best results for such prediction.

5.4 Discussion

In the previous section, I relied on statistical criteria to empirically assess the performance of our prediction models. To determine the usefulness of these predictions from a practical perspective, I also manually studied some of the results produced by my models. Without

being exhaustive, here I describe some of our findings in the case of the Camel project, providing concrete evidence as to how the prediction models can be useful in practice for identifying the architectural problems.

I manually investigated whether architectural quality metrics, such as architectural smells, used in the construction of my prediction models, are indeed architectural problems the developers care about and aim to resolve. I found many cases corroborating the validity of my quality metrics through the developers' commit logs and changes that involved restructuring of the system's architecture. As a case in point, my metrics identified the following four packages to have DC on 2/17/09:

- */org/apache/camel/component/cxf*
- */org/apache/camel/component/cxf/util*
- */org/apache/camel/converter/stream*
- */org/apache/camel/converter*

But those packages did not have a DC in a version that was released two months later. To confirm our DC metric is indeed properly capturing an issue in the architecture of the system, I looked at the log commits of Camel, filtered the changes that include those packages, and found the following messages:

- `revision: 749227`
  `author: davsclaus`
  `date: Mon Mar 02 03:20:07 EST 2009`
  `log message: CAMEL-588: LoggingLevel moved from model to root pacakge to improve API package structuring.`
  `changed paths:`
  `/org/apache/camel/util/MessageHelperTest.java`
  `/org/apache/camel/converter/stream/StreamCacheConverterTest.java`
  `/org/apache/camel/converter/stream/StreamCache.java`

/org/apache/camel/converter/stream/StreamCacheConverter.java

/org/apache/camel/util/MessageHelper.java

- revision: 749236

  author: davsclaus

  date: Mon Mar 02 03:48:10 EST 2009

  log message: CAMEL-588: Fixed bad package tangle.

  changed paths:

  /org/apache/camel/util/SystemHelper.java

  /org/apache/camel/util/IOHelper.java

  /org/apache/camel/converter/IOConverter.java

  /org/apache/camel/converter/IOConverterTest.java

- revision: 749561

  author: davsclaus

  date: Tue Mar 03 03:15:15 EST 2009

  log message: CAMEL-588: Removed package dependency and using the type converter API to find the right converter instead of direct usage.

  changed paths:

  /org/apache/camel/converter/stream/StreamCacheConverterTest.java

  /org/apache/camel/converter/stream/StreamCacheConverter.java

I also looked at CAMEL-588 in Jira; the description of the issue starts as follows: ‘‘Currently there is a bad dependency cycle between camel, spi and model...’’. These comments clearly describe the same phenomenon intended to be measured by DC metric (recall Section 5.1.3). Experience such as this provide concrete evidence that architectural smell metrics can be effective in practice with helping the practitioners identify architectural problems and decaying elements.

I also found many cases in which our smell emergence predictions were found to be issues

that the developers had acknowledged in their commit logs and had attempted to resolve. A concrete example of this situation occurred with the */org/apache/camel/language/simple* package, which did not have DC for multiple releases, but our model predicted that it will start to have DC from version 2.5 (10/31/2010). When I manually investigated the commit logs, excerpts of which are shown below, not only did I find evidence of DC emergence, but also attempts by the developers to fix the problem afterwards:

- revision: 1150991

  author: davsclaus

  date: Tue Jul 26 01:49:04 EDT 2011

  log message: CAMEL-3961: Polished and reduced some package tangling.

  changed paths:

  /org/apache/camel/language/simple/SimpleLanguageSupport.java

  /org/apache/camel/language/simple/SimpleLanguage.java

- revision: 1171490

  author: cschneider

  date: Fri Sep 16 06:25:25 EDT 2011

  log message: CAMEl-4457 Move types of the simple language to a new package simple.types to avoid dependency cycle

  changed paths:

  /org/apache/camel/language/simple/SimpleParserException.java

  /org/apache/camel/language/simple/TokenType.java

  /org/apache/camel/language/simple/BinaryOperatorType.java

  /org/apache/camel/language/simple/SimplePredicateParser.java

  /org/apache/camel/language/simple/SimpleTest.java

  /org/apache/camel/language/simple/SimpleTokenType.java

  /org/apache/camel/language/simple/SimpleIllegalSyntaxException.java

  /org/apache/camel/language/simple/SimpleParserPredicateInvalidTest.java

  /org/apache/camel/language/simple/SimpleTokenizer.java

/org/apache/camel/language/simple/SimpleToken.java

/org/apache/camel/language/simple/SimpleOperatorTest.java

/org/apache/camel/language/simple/BaseSimpleParser.java

/org/apache/camel/language/simple/UnaryOperatorType.java

/org/apache/camel/language/simple/SimpleExpressionParser.java

/org/apache/camel/language/simple/SimpleParserExpressionInvalidTest.java

/org/apache/camel/language/simple/LogicalOperatorType.java

/org/apache/camel/language/simple/SimpleBackwardsCompatibleParser.java

The description of CAMEl-4457 in Jira summarizes the issue: ‘‘Currently we have a big dependency cycle between language.simple and language.simple.ast’’.

I believe using my smell emergence prediction models, Camel developers could have identified and refactored the decaying architectural modules earlier.

My experience were not limited to DC. As another case in point, we were able to predict */org/apache/camel/component/log* will not have the LO smell in a future release, even though it had that smell in preceding releases. When I investigated the commit logs, I found evidence that the architecture of the system had been refactored in between the releases:

- revision: 749193

  author: davsclaus

  date: Mon Mar 02 00:30:35 EST 2009

  log message: CAMEL-588: Package tangle fixes. Tokenizer in spring renamed to Tokenize. And fixed a CamelCase.

  changed paths:

  /org/apache/camel/component/log/LogFormatter.java

  /org/apache/camel/model/language/TokenizerExpression.java

- revision: 749212

  author: davsclaus

```
date:  Mon Mar 02 02:04:20 EST 2009
log message:  CAMEL-588:  Moved LoggingLevel from model to core
package, to fix bad tangle.
changed paths:
/org/apache/camel/component/log/LogComponent.java
```

In summary, my analysis suggests that not only can I accurately predict many architectural quality concerns, but that such concerns are indeed taken seriously by the developers of open-source software, as evidenced by commit logs showcasing their attempts to fix degraded architectural modules. I believe my prediction models could help developers detect software architectural decay in a systematic fashion, possibly prior to its full manifestation in code.

5.5 Threats to Validity

I now describe the main threats to validity of my findings.

Construct validity is concerned with whether we are actually or accurately measuring the constructs we are interested in studying. One such threat involves the correctness of my linking of modules and their constituent files with defects. However, recall from Section 5.2.1 that the process used by engineers in ASF to link bug-fixing commits and issues significantly mitigates this threat.

Another threat to construct validity has to do with the accuracy of the architectural modules I obtain. I address this threat in several ways: I selected a technique, ARC, that has exhibited higher accuracy when compared to other techniques in previous work [40]. I further complement the semantic view provided by ARC with a structural view obtained through packages. Additionally, any inaccuracies in our identification of architectural modules would only degrade the results of our predictions. However, my models still achieve high performance. Nevertheless, to ensure that the architectural modules I obtained are meaningful, I attempted to use my prediction models on randomly generated modules for

each of our five subject systems. Given that the resulting modules have random files in them, no traceability can be achieved between modules—i.e., there is no longer a module m_{k+1} in release $k+1$ that is similar to a module m_k in release k. This result further validates that I obtain meaningful architectural modules.

The final threat to construct validity involves whether my selected metrics actually represent architectural decay or the factors that predict architectural quality. To ensure that I have a comprehensive set of metrics that represent architectural decay, I included three types of architectural-quality metrics: architectural defects, architectural smells, and CF. For the factors that may indicate architectural decay, i.e., the independent variables of my models, I selected a wide variety of metrics that do not overlap, in order to avoid the multicollinearity problem.

Threats to *external validity* involve the generalizability of my findings. One such threat is that all our projects are from ASF and are implemented in Java. To mitigate this threat, I selected projects from different application domains that vary in their sizes. Furthermore, Java is a widely used language, making our results more generalizable.

Another threat involves the fact that I only include open-source projects. However, ASF projects are widely used, even in industrial settings, which allows my projects to generalize further.

5.6 Tool

Now I discuss the tool that I implemented for predicting architectural quality. It consists of two parts:

- Data collection: It is implemented in Java and is responsible for collecting the required data for building prediction models from various sources.
- Model construction: This part is implemented in R and constructs the prediction models and evaluates the results on the data.

5.6.1 Data Collection

This part of the tool is a Jar file that takes a number of arguments and generates the data file for building the prediction models. Here I discuss the input parameters to the Jar file:

- Release date: This is the release date of the version of the project that I want to do the analysis on and it is in the format "mm/dd/yyyy" e.g. "3/5/2012".

- Architectural modules: The is the path to the file that shows the architectural modules in the system and the files inside each module. Each line of the file contains a file and the module that it belongs to in this format:

  `contain module file`

- Repository address: This is the address to the repository of the project. For example Camel repository address is:

  `http://svn.apache.org/repos/asf/camel/trunk/`

  The address is used to obtain the change metrics (*NC, NCF, CMC and IMC*) that are calculated by processing the developer commits from an SVN repository and extracting the groups of files in the same commit transaction that have been modified together (i.e., co-changes). I use *SVNKit*, a Java toolkit providing APIs to subversion repositories.

- Prefix for file names: While I was running the experiments, I found out that sometimes a project contains files that belongs to other projects. Since I wanted to make sure that I only consider the files that are specific to the project under study, I use this input parameter to filter outer projects files. For example the prefix of all files that belongs to Camel is `/org/apache/camel/`

- Regular expression for finding defect fixes: In section 5.2.1 I described that in the ASF software repositories and, by extension, the projects studied in this dissertation, the commits that are defect fixes are identifiable since defects are referred to by a project

name and defect number in SVN commit logs. This input is used to find defect fixes. To make the code more general purpose and to be able to use it with more projects, this parameter is in the format of Java regular expression and can be tuned based on the characteristics of a project. For example for Camel I put it as: ".*CAMEL.*".

- Lifted file-level metrics : This is the input file that contains the first five file-level metrics (*LOC, SCC, DIT, CBO and LCM*) that are measured using UNDERSTAND from Scitools[3] for each release. The input to the UNDERSTAND tool is the source code of a project.

- Architectural smell metrics: This is the input file to the four architectural smell metrics of (*SF*, *CO*, *DC*, and *LO*). I used *Architecture Recovery, Change, And Decay Evaluator (ARCADE)* [39, 60], a workbench containing tools for addressing architectural decay, to extract these metrics. Each line of the input file shows a module and its related smell.

  ```
  module smell
  ```

- Architectural dependency-based metrics: This is the input file to the six architectural dependency-based metrics (*CMD*, *OMD*, *TCMD*, *TOMD*, *IMD*, and *XMD*). I used *ARCADE* to extract these metrics.

- Architectural smell metrics for the next version of project: This is the input file that contains the smell metrics for the next version of the project. This information is needed for evaluating the performance of my prediction models.

- Architectural dependency-based metrics for the next version of project: This is the input file that contains the dependency-based metrics for the next version of the project. I use this information for evaluating the performance of my prediction models.

- Mapping between architectural modules: In section 5.1.1 I explained that to use the data of current release to make prediction for next release, I must be able to determine

[3] http://www.scitools.com/

which module m_k in release k is the same module m_{k+1} in release $k+1$. This parameter includes that information. Each line of the file shows m_k m_{k+1}.

The Jar file generates a file which includes architectural modules and corresponding metrics values for each module. that can be used to do the analysis.

5.6.2 Model Construction

The model construction part of the tool is implemented in R. In this section I discuss the code that I used to build my prediction models. I explained the details of approach in sections 5.1 and 5.2. The code starts by loading the required libraries:

```
require(MASS)
require(ROCR)
require(randomForest)
```

Next I load the input data that was created using the Jar file in the previous section. (e.g. data for packages in Hive)

```
change <- read.csv("/Users/Ehsan/Workspace/RData/Data.csv", header = T)
```

Here I explain the code for building defect prediction models for modules (to build prediction models for other architectural quality metrics, I only need to change the dependent variable in the code and replace that with one of the dependent variables (SF, DC, CF, CO and LO) in my study). To build defect prediction models, I choose defect (number of defects per module) as the dependent variable in the code. For building the model using NBR, first I need to create the model and then use stepwise regression to select the best subset of metrics:

```
summary(m1 <- glm.nb(defects ~ log2(LOC+1) + log2(NC+1) + log2(CBO+1)
+ log2(DIT+1) + log2(LCM+1) + log2(SCC+1) + log2(CMC+1) + log2(IMC+1)
+ log2(NCF+1) + CO + SF + DC + LO + log2(CMD+1) + log2(OMD+1) +
log2(IMD+1) + log2(XMD+1) + log2(TCMD+1) + log2(TOMD+1), data = change))
```

I increase the values of independent variables by one before log_2 transformation to avoid encountering $log_2(0)$ in case some of the variables are 0. Next I use stepwise regression

to avoid the multicollinearity problem and find the best subset of metrics for building prediction models (for NBR and LR).

```
step <- stepAIC(m1, direction = "both")
step$anova #display results
```

For example the results of stepwise regression for Hive packages is:

```
Stepwise Model Path
Analysis of Deviance Table
Initial Model:
glm.nb(defects ~ log2(LOC+1) + log2(NC+1) + log2(CBO+1) + log2(DIT+1)
+ log2(LCM+1) + log2(SCC+1) + log2(CMC+1) + log2(IMC+1) + log2(NCF+1)
+ CO + SF + DC + LO + log2(CMD+1) + log2(OMD+1) + log2(IMD+1) +
log2(XMD+1) + log2(TCMD+1) + log2(TOMD+1)
Final Model:
defects ~ log2(LOC+1) + log2(CBO+1) + log2(LCM+1) + log2(CMC+1) +
log2(OMD+1) + log2(IMD+1) + log2(TOMD+1)
```

Which shows that LOC, CBO, LCM, CMC, OMD, IMD and TOMD are the optimal subset of metrics for predicting defects for packages in Hive.

Next I use the optimal set of metrics and populate them in the function that I wrote to assess the predictive power of models (see 5.2.2).

```
predictivePowerNBR <- function(train, test)
{
   model.glm.nb <- glm.nb(defects ~ log2(LOC + 1) + log2(CBO + 1) +
   log2(LCM + 1) + log2(CMC + 1) + log2(OMD + 1) + log2(IMD + 1) +
   log2(TOMD + 1), data = train)
   test.prob <- predict(model.glm.nb, test, type = "response")
   pred <- prediction(test.prob, test$defects > 0)
   auc <- performance(pred, "auc")@y.values[[1]]
   return(list(auc = auc))
```

```
}
```

predictivePowerNBR function builds a prediction model using train input data and returns the AUC prediction performance of model by using the test input data.

In the next step, I use the optimal subset of metrics and evaluate the ranking performance of prediction:

```
rankingNBR <- function(train, test)
{
   model.glm.nb <- glm.nb(defects ~ log2(LOC+1) + log2(CBO+1) +
   log2(LCM+1) + log2(CMC+1) + log2(OMD+1) + log2(IMD+1) + log2(TOMD+1),
   data = train)
   test.pred <- predict(model.glm.nb, test, type = "response")
   spearman <- cor(test$defects, test.pred, method = "spearman")
   spearman.p <- cor.test(test$defects, test.pred, method = "spearman",
   exact = FALSE)$p.value
   return(list(spearman = spearman, spearman.p = spearman.p))
}
```

rankingNBR function builds a prediction model using train input data and returns the ranking performance of model using the test input data.

The procedure for building LR prediction models is similar to NBR except using *lm* instead of *glm.nb* in the above code.

I use all of the metrics for RF without using stepwise regression. The following is the code for predictive power and ranking functions for RF:

```
predictivePowerRF <- function (train, test)
{
   randomForest <- randomForest(defects ~ log2(LOC+1) + log2(NC+1) +
   log2(CBO+1) + log2(DIT+1) + log2(LCM+1) + log2(SCC+1) + log2(CMC+1)
   + log2(IMC+1) + log2(NCF+1) + CO + SF + DC + LO + log2(CMD+1) +
   log2(OMD+1) + log2(IMD+1) + log2(XMD+1) + log2(TCMD+1) + log2(TOMD+1),
```

```
   data=train)
   test.prob <- predict(randomForest, test, type="response")
   pred <- prediction(test.prob, test$defects>0)
   auc <- performance(pred,"auc")@y.values[[1]]
   return(list(auc=auc))
}
rankingRF <- function(train, test)
{
   randomForest <- randomForest(defects ~ log2(LOC+1) + log2(NC+1) +
   log2(CBO+1) + log2(DIT+1) + log2(LCM+1) + log2(SCC+1) + log2(CMC+1)
   + log2(IMC+1) + log2(NCF+1) + CO + SF + DC + LO + log2(CMD+1) +
   log2(OMD+1) + log2(IMD+1) + log2(XMD+1) + log2(TCMD+1) + log2(TOMD+1),
   data = train)
   test.pred <- predict(randomForest, test, type = "response")
   spearman <- cor(test$defects, test.pred, method = "spearman")
   spearman.p <- cor.test(test$defects, test.pred, method = "spearman",
   exact = FALSE)$p.value
   return(list(spearman = spearman, spearman.p = spearman.p))
}
```

Recall from 5.2.2 that I use data splitting for evaluating the performance of models. The code below is for k splitting that I invoke it with $k = 3$ for my experiments. It runs the experiment for 100 times and returns the average.

```
dataSplittingPredictivePower <- function(inputData, k, method)
{
   counter <- 100
   auc <- 0
   for (j in 1:  counter)
   {
```

```
      change2 <- inputData[sample(nrow(inputData)), ]
      folds <- cut(seq(1, nrow(change2)), breaks = k, labels = FALSE)
      #Segement the data by fold using the which() function
      testIndexes <- which(folds == 1, arr.ind = TRUE)
      testData <- change2[testIndexes, ]
      trainData <- change2[-testIndexes, ]
      if (method == "LR")
         results <- predictivePowerLR(trainData, testData)
      if (method == "NBR")
         results <- predictivePowerNBR(trainData, testData)
      if (method == "RF")
         results <- predictivePowerRF(trainData, testData)
      auc <- results$auc + auc
   }
   print(paste0(" AUC:", auc/counter))
}
```

This is the code for evaluating the performance of prediction model using data splitting. It split the *inputData* to training and test data and then invokes the appropriate *predictivePower* function based on the method (NBR, LR or RF).

Similarly I implemented a method that invokes the ranking functions:

```
dataSplittingRanking <- function(inputData, k, method)
{
   counter <- 100
   spearman <- 0
   spearman.p <- 0
   for (j in 1:  counter)
   {
      change2 <- inputData[sample(nrow(inputData)), ]
```

```
    folds <- cut(seq(1,nrow(change2)), breaks = k, labels = FALSE)
    #Segement the data by fold using the which() function
    testIndexes <- which(folds == 1, arr.ind = TRUE)
    testData <- change2[testIndexes, ]
    trainData <- change2[-testIndexes, ]
    if (method == "LR")
       results <- rankingLR(trainData, testData)
    if (method == "NBR")
       results <- rankingNBR(trainData, testData)
    if (method == "RF")
       results <- rankingRF(trainData, testData)
    spearman <- results$spearman + spearman
    spearman.p <- results$spearman.p + spearman.p
  }
  print(paste0(method, " spearman:  ",    spearman/counter, "
  spearman.p:  ", spearman.p/counter))
}
```

dataSplittingRanking function splits the input data and invokes the appropriate *ranking* method and runs the experiment for 100 times and returns the average result.

These are the commands for invoking dataSplittingPredictivePower and dataSplittingRanking functions:

```
dataSplittingPredictivePower(change, 3, "LR")
dataSplittingPredictivePower(change, 3, "NBR")
dataSplittingPredictivePower(change, 3, "RF")
dataSplittingRanking(change, 3,"LR")
dataSplittingRanking(change, 3,"NBR")
dataSplittingRanking(change, 3,"RF")
```

This is the result of the defect prediction models for packages in Hive:

```
> dataSplittingPredictivePower(change, 3, "LR")
[1] " AUC:0.763423051239963"
> dataSplittingPredictivePower(change, 3, "NBR")
[1] " AUC:0.835438933833255"
> dataSplittingPredictivePower(change, 3, "RF")
[1] " AUC:0.82699183813707"
> dataSplittingRanking(change, 3,"LR")
[1] "lm spearman:  0.567862954885859 spearman.p:  0.000641306444207375"
> dataSplittingRanking(change, 3,"NBR")
[1] "glm.nb spearman:  0.678524030390626 spearman.p:
8.12849987046284e-07"
> dataSplittingRanking(change, 3,"RF")
[1] "randomForest spearman:  0.637010274169935 spearman.p:
1.95707472862749e-06"
```

These results show the performance of the prediction models. We can use the models to show the prediction values instead of the performance of models. For example table 5.2 shows the actual and predicted values of CF for packages in HBase. Now I explain how to use the models to show the prediction values. Suppose that version 0.8.1 of Hive project is just released and we want to predict the number of defects for packages for that version. First I build the prediction model using the available data of previous releases (Here I use the data for versions 0.3.0, 0.4.1, 0.5.0, 0.6.0, and 0.7.0. The train data set includes the data for all of the dependent and independent variables for above releases:

```
trainData <- read.csv("/Users/Ehsan/Workspace/RData/trainData.csv",
header = T)
```

Next I extract that data for independent variables for version 0.8.1 (since I want to predict the values of dependent variables) and put them in the test data set:

```
testData <- read.csv("/Users/Ehsan/Workspace/RData/testData.csv",
header = T)
```

After using the step wise regression on the train data set, I build the model using the best subset of metrics:

```
modelNBR <- glm.nb(defects ~ log2(LOC+1) + log2(CBO+1) + log2(LCM+1)
+ log2(CMC+1) + log2(OMD+1) + log2(IMD+1) + log2(TOMD+1), data =
trainData)
```

Now I use the model to predict the number of defects for modules in test data:

```
prediction <- predict(modelNBR, testData, type = "response")
testData$predictedDefects <- round(prediction)
testData[ , c("Name", "predictedDefects")]
```

The results are shown in table 5.4

Likewise if I want to predict other architectural quality metrics, I just need to build the model using the dependent variable for that architectural quality metric. For example for predicting to see which of the packages in version 0.8.1 of Hive would have architectural smell of LO, I change the dependent variable of defects with LO. After running the step wise regression on the train data set, I build the model for predicting LO using best subset of metrics:

```
modelNBR <- glm.nb(LOnextRelease ~ log2(NC+1) + log2(IMC+1) +
log2(NCF+1) + DC + LO + log2(XMD+1) + log2(TOMD+1), data = trainData)
```

Now I can use the model to predict which packages have LO:

```
prediction <- predict(modelNBR, testData, type = "response")
testData$predictedLO <- round(prediction)
testData[ , c("Name", "predictedLO")]
```

Table 5.4 shows the packages in version 0.8.1 of Hive and the predicted values of defects and LO for each package.

Table 5.4: Prediction of Defects and LO for Packages in Hive (Version 0.8.1)

Name	predicted defects	predicted LO
org.apache.hadoop.hive.service	0	0
org.apache.hadoop.hive.serde2.objectinspector	0	1
org.apache.hadoop.hive.serde2	0	1
org.apache.hadoop.hive.serde2.lazybinary	0	0
org.apache.hadoop.hive.ql.optimizer.unionproc	0	0
org.apache.hadoop.hive.ql.optimizer.physical	1	0
org.apache.hadoop.hive.thrift.client	0	0
org.apache.hadoop.hive.metastore.api	1	1
org.apache.hadoop.hive.ql.processors	0	0
org.apache.hadoop.hive.conf	6	1
org.apache.hadoop.hive.ql.udf.generic	2	0
org.apache.hadoop.hive.serde2.thrift	0	0
org.apache.hadoop.hive.ql.index.compact	1	0
org.apache.hadoop.hive.ql.index	0	0
org.apache.hadoop.hive.metastore	10	0
org.apache.hadoop.hive.ql.optimizer	6	0
org.apache.hadoop.hive.shims	4	0
org.apache.hadoop.hive.serde2.objectinspector.primitive	0	1
org.apache.hadoop.hive.ql.io.rcfile.merge	0	0
org.apache.hadoop.hive.ql.stats.jdbc	1	0
org.apache.hadoop.hive.jdbc	1	0
org.apache.hadoop.hive.ql.exec	27	1
org.apache.hadoop.hive.serde2.lazy	0	0
org.apache.hadoop.hive.ql.plan	4	1
org.apache.hadoop.hive.ql	1	0
org.apache.hadoop.hive.ql.lockmgr.zookeeper	1	0
org.apache.hadoop.hive.hbase	1	0
org.apache.hadoop.hive.ql.parse	6	1
org.apache.hadoop.hive.serde2.typeinfo	0	0
org.apache.hadoop.hive.ql.metadata	3	1
org.apache.hadoop.hive.serde2.dynamic_type	0	0
org.apache.hadoop.hive.ql.session	1	0
org.apache.hadoop.hive.ql.io	4	0
org.apache.hadoop.hive.thrift	6	0

Chapter 6: Challenges and Suggestions for the Community

In this section, I describe the challenges and limitations I faced throughout my research followed by some suggestions for the community.

6.1 Challenges and Limitations

I briefly discuss some of the challenges of software architecture-based empirical studies in this section.

6.1.1 The Lack of the Availability of Software Architecture Information

The first and most important problem to conduct software architecture-based empirical research is the lack of availability of software architecture information for software systems. Buse and Zimmermann conducted an study in which they surveyed 110 developers and managers at Microsoft about the data and analysis needs of professional software engineers [17]. In one part of their study, they asked the participants about a number of popular artifacts and metrics (e.g. bug reports, code clones, dependencies, architecture, change type, test coverage, ownership, etc.) and whether they currently use it or if they would use it if it was made available. More than 95% of developers said that they would use software architecture information (highest ranked) if it is made available to them while only 30% of them said they currently use software architecture information. This clearly shows that the reason that most of the developers do not use architecture information is due to the lack of availability of it, although most of them want to benefit from it.

One reason for this lack of information about software architecture could be based on the fact that developers and other stakeholders in a project do not see the benefits of spending time and resources for up-front architectural design. This is more valid in the software

development environments that use agile processes, which has become more common in recent years. To maximize agility, agile developers often avoid or minimize architectural planning since architectural planning often seen as delivering little values to customers [3]. In many companies, e.g. start-up companies, the top priority is to develop the business idea and deliver the product as quickly as possible. Instead of putting so much effort for up-front designing of architecture that is scalable to millions of users, they rather have the software ready for a smaller crowd as quickly as possible to make sure not to lose the business [104]. Later when they have millions of users, they would deal with the scalability issues and may have to rewrite most of the code and replace the architecture completely. My research can be very beneficial to agile developers as they can use the decay prediction models for identifying the architectural problems and refactoring the code.

6.1.2 Tracing Defects to Architecture

One of the key factors for doing research in defect prediction and change impact analysis is the ability to identify the files/modules that contain defects. When facing a problem and failure in a software system, usually a developer or a user opens an issue in an issue tracking system e.g. Bugzilla or Jira. It is recommended that when a developer makes some changes to one or more files to fix an issue, he/she should include the related issue number that is trying to fix along with the commit in the source version repository. That enables us to trace the failure to its source and identify which parts of the code contains defects and are responsible for that failure. This practice is forced in some open source projects (e.g. Apache Foundation). Another approach is to search in the commit logs for specific tokens like bugs, fixes and defects followed by a number.

For conducting such a study at architectural level one needs to know which architectural modules contain defects. One way to do this is to find the files involved in defects as explained above and then consider the architectural modules that include those files as faulty. This requires having the information that for each file in the system, what architectural module contains it. The problem is that often architectural documentation does not

exist (e.g. in open source projects) or even if there are some documents regarding software architecture, they show the high level structure of the system and do not explicitly contain the information regarding the exact location of each file in architectural modules in the system. That is why we need to have an approach for extracting the architectural module information from the source code which include the location of each file in those modules as well.

6.1.3 Obtaining Architectural Modules

Architecture recovery is the process of extracting architecture from source code and as explained before, is essential for doing architectural related empirical studies. Architectural recovery is also needed for architectural decay investigation, since we need to recover the architecture of a system throughout its evolution. Although there have been many approaches for architectural recovery [29], not all of them are applicable to use when conducting empirical studies. In empirical studies one needs to evaluate hypotheses on multiple software systems in which he/she probably is not familiar with the domain and the code. Therefore manual architectural recovery techniques are not suitable here.

My experience with automatic recovery techniques is that I could not easily access some of the available tools and for some cases I had to contact the authors of the tools to have access to them. Also I observed that different tools often generate different architectural modules for the same system. This makes it harder for the researcher or practitioner to choose which tools to use. One recent study showed that although there have been a lot of research in architectural recovery, there is still a dire need to have more accurate recovery techniques [65].

For addressing this problem I used different architectural recovery techniques: *Bunch* [71], *ArchDRH* [18], *ACDC* [101] and *ARC* [44] in my research. This is in line with the fact that comprehending the architecture and architecturally significant issues of any complex systems requires looking at the architecture from different perspectives [7, 59]. These perspectives are known as *architectural views*, each dealing with a separate concern

and different recovery techniques represents different architectural views. According to Clements et al. [22] three view types are commonly used to represent the architecture of a system: *Module View*, *Component-and-Connector View*, and *Allocation View*. Module View shows units of implementation, Component-and-Connector View represents a set of elements that have runtime behavior and interactions, and Allocation View shows the relationship between software and non-software resources of development (e.g., team of developers) and execution environment (e.g., hardware elements).

I also used package structuring of a system since The package structure of a system can be treated as a proxy for the decomposition of the system into architecturally significant elements, as packages are created by the developers of the system.

6.2 Opportunities and Suggestions

In this section, I provide some suggestions about research opportunities that would benefit the community.

6.2.1 Creating a Repository for Software Architecture

As I discussed before, the biggest challenge for conducting software architectural empirical studies is the lack of availability of software architecture for software systems especially open source systems. This means that every researcher and practitioner has to use different recovery techniques to recover the architecture of the systems he wants to study. A designated repository that holds the architecture information of different systems could have numerous benefits for the community. First, researchers do not need to recover the architecture of the same system from scratch and repeating the same thing over and over again. They can refer to the repository and use the architecture of the systems available there. This also results in having different architectural views for a system and researchers and practitioners could use the view that is most appropriate for their research. Architects and developers of open source software systems can put the ground-truth architecture of the systems in the repository which would be a valuable source of information for the community.

6.2.2 A Comprehensive Tool Suite for Architecture Recovery

A tool suite that contains all the existing well-known recovery techniques can tremendously benefit the community. Researchers and practitioners can use the tool without wasting time trying to find and educate themselves about different techniques. They can recover multiple architectural views of the system under the study and use the ones that are suitable for their needs. Another important factor is the ability to add new recovery techniques to the tool. The tool should provide some facility that makes it easy for the researchers to be able to add their recovery techniques to the tool.

6.2.3 Bringing Software Architecture to Software Engineers' Every Day Life

One of the most important contributions in this field is to make sure that developers and other stakeholders realize the importance of having software architecture information and how they can benefit from it. The important challenge here is that how we can incorporate software architecture in the tools and other resources that developers and other stakeholders use daily. For example by having architectural information in IDEs, developers can monitor the impact of their changes on the architecture while committing their code. We should facilitate the infrastructure in version control systems and issue tracking systems that it makes it easy for the developers to log the architectural changes and architectural problems in the system. We should embed architectural information in defect prediction models so we can avoid making the changes that would lead to architectural problems in software systems.

Software architecture information and architectural-level prediction models would enable the architect to make informed decisions for designing the architecture of a system or selecting the appropriate architecture from different architectural choices. They can also be used in architecture-based adaptation, which is the process of reasoning about and adapting a systems software at the architectural level [57, 80]. Architecture-based adaptation is used to satisfy quality objectives such as reliability of a system [24] based on different

architectural configurations or in presence of uncertainty in a system [35].

Having architectural information is vital in addressing technical debt. Technical debt metaphor is widely used to encapsulate numerous software quality problems. A recent study showed that architectural decisions are the most important source of technical debts [34] and the authors suggested that the research in technical debt tooling should focus on monitoring the gap between development and architecture. Architectural decay prediction models can be used to address technical debt as we can use them avoid making changes that would result in architectural decay.

Another place that can benefit from software architecture is in code review. Baccehelli and Bird survey developers and managers about motivations, challenges and outcomes of tool-based code reviews and found that while finding defects remain the main motivation for code review, reviews are less about defects than expected and instead provide additional benefits such as knowledge transfer, increased team awareness and creation of alternative solutions to problems [5] By incorporating architecture in code review tools, we can have architects review the code when it would result in a major architectural change. Having the architectural models and documents can help developers with knowledge transfer and understanding the code and the system especially when the original developers are not involved in the project anymore. It is worth nothing that architectural models can include other design related information too e.g. knowledge about design patters used in the projects [56, 96] or using techniques and tools that can identify architectural tactics in the system [69, 70].

Chapter 7: Conclusion

Conventional wisdom suggests that a software system's architecture plays an important role in its evolution and maintenance, in particular the ease with which changes can be made to that system. In this dissertation, I have tried to collect empirical evidence as to the role of software architecture in the evolution of a software system. In this chapter I conclude my dissertation by summarizing the findings and the contributions of my research and avenues for future work.

Chapter 4 reports on an empirical study that aims to provide concrete evidence of the impact of architecture on the quality of software during its evolution. Although several studies have used the co-change history to build defect prediction models, no prior study investigated the impact of co-changes involving several architectural modules versus co-changes localized within a single module. In the absence of explicit architectural models in open-source projects, to conduct this study I used surrogate models that approximate the architecture of a system. My findings show that co-changes that crosscut multiple architectural modules are more correlated with defects than co-changes that are localized in the same module. I also arrived at the same conclusion when I performed the study using a commercial project, as well as an open-source project with a documented architecture. My study corroborates the importance of considering software architecture as one of the key factors affecting the quality of a changing software system. I am formulating a research agenda that aims to correlate the revision/defect history of software with its architecture to shed light on the root cause of problems. The insight in my research is that predicting where defects are likely to occur, which has received much attention in the past decade, is not as useful to the developers as helping them understand why they occur. To that end, I believe architecture provides an appropriate level of granularity for understanding the root cause of a large class of defects that are due to bad architectural choices.

Chapter 5 discusses my approach for addressing architectural decay. Architectural decay is a phenomenon of software systems that leads to defects and increases maintenance time and effort. To address this issue, I constructed models for predicting three types of architectural decay: architectural defects, architectural smells, and modularization quality. For 40 versions of five software systems, I can predict architectural decay with high performance across two architectural views—one semantic view and another structural view. Even when architectural smells suddenly emerge in a module, I can predict these rare cases with high performance (AUC of 0.79-0.96). I further discovered that architectural smells tend to remain in modules once they emerge. Lastly, I discovered that a wide variety of metrics—of which file-level metrics are only a subset—are needed to predict architectural decay.

7.1 Contributions

The following is a concrete list of contributions of this research:

- **Fundamental contribution to software architecture-based empirical research:** Due to some limitations that I described in section 6.1, there is a limited number of empirical studies with the focus of software architecture. My research includes a methodology on how to benefit from software architecture information in empirical studies and how to explore the impact of software architecture on evolution of software systems especially open source projects where mostly there is no documented architecture.

- **Presenting new architectural-level metrics:** As part of my thesis, I proposed new metrics that distinguishes between the types of co-changes by considering software architecture and I empirically showed that using those metrics would enable us to identify architectural problems and develop more accurate defect prediction models.

- **Architectural quality prediction:**

I proposed and empirically evaluated an approach for constructing models to predict architectural quality and decay in a software systems. My approach utilizes multiple file-level and architectural-level metrics (including architectural bad smell metrics) and can be used predict different architectural-quality metrics.

- **Tool:** I implemented a tool that can be used to collect and aggregate multiple file-level and architectural-level metrics and use the data to build architectural quality prediction models that can effectively predict architectural decay in software systems. This tool is implemented in Java and R.

7.2 Future Work

In the future, I intend to move beyond prediction of architectural decay by determining the specific actions that can be taken to prevent decay once it occurs. One possible direction is the utilization of the important factors of our prediction models to identify specific preventative measures. As an example, a promising possibility is applying architectural restructurings that removes architectural bad smells e.g. dependency cycle. Once such preventive measures are applied, I can perform further assessment. For example, I can conduct a study to examine whether engineers can more quickly or easily perform maintenance tasks, after restructurings are applied.

Another promising future work is to embed the architectural quality prediction technique in development tools such as IDEs, code review and technical debt tools to help software engineers with maintaining software systems.

Bibliography

Bibliography

[1] CRAN - Package MASS. http://cran.r-project.org/web/packages/MASS/index.html.

[2] CRAN - Package randomForest. http://cran.r-project.org/web/packages/randomForest/index.html.

[3] Abrahamsson, P., Babar, M. A., and Kruchten, P. Agility and architecture: Can they coexist? *IEEE Softw. 27*, 2 (Mar. 2010), 16–22.

[4] Allen, E., and Khoshgoftaar, T. Measuring coupling and cohesion: an information-theory approach. In *Software Metrics Symposium, 1999. Proceedings. Sixth International* (1999), pp. 119–127.

[5] Bacchelli, A., and Bird, C. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA, 2013), ICSE '13, pp. 712–721.

[6] Bachmann, A., Bird, C., Rahman, F., Devanbu, P., and Bernstein, A. The missing links: bugs and bug-fix commits. In *18th ACM SIGSOFT international symposium on Foundations of software engineering* (Santa Fe, New Mexico, Nov. 2010), FSE '10, ACM, pp. 97–106.

[7] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, 2 ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[8] Bavota, G., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., and De Lucia, A. An Empirical Study on the Developers' Perception of Software Coupling. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA, May 2013), ICSE '13, IEEE Press, pp. 692–701.

[9] Bavota, G., Gethers, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies. *Accepted to appear in ACM Transactions on Software Engineering and Methodology (TOSEM)*.

[10] Beck, F., and Diehl, S. Evaluating the Impact of Software Evolution on Software Clustering. In *17th Working Conference on Reverse Engineering* (Beverly, Massachusetts, Oct. 2010), pp. 99–108.

[11] Blei, D. M., Ng, A. Y., and Jordan, M. I. Latent Dirichlet Allocation. *J. Mach. Learn. Res. 3* (Mar. 2003), 993–1022.

[12] Bouwers, E., van Deursen, A., and Visser, J. Quantifying the Encapsulation of Implemented Software Architectures. In *30th IEEE International Conference on Software Maintenance and Evolution (ICSME)* (Victoria, BC, Canada, Oct. 2014), pp. 211–220.

[13] Breu, S., and Zimmermann, T. Mining Aspects from Version History. In *21st IEEE/ACM International Conference on Automated Software Engineering* (Tokyo, Japan, Sept. 2006), pp. 221–230.

[14] Briand, L., Morasca, S., and Basili, V. Measuring and assessing maintainability at the end of high level design. In *Proceedings of the Conference on Software Maintenance* (Montreal, Canada, Sept. 1993).

[15] Briand, L. C., Wust, J., Daly, J. W., and Victor Porter, D. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of Systems and Software 51*, 3 (May 2000), 245–273.

[16] Brunet, J., Bittencourt, R. A., Serey, D., and Figueiredo, J. On the evolutionary nature of architectural violations. In *Reverse Engineering (WCRE), 2012 19th Working Conference on* (2012), IEEE.

[17] Buse, R. P. L., and Zimmermann, T. Information needs for software development analytics. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland, 2012), ICSE '12, pp. 987–996.

[18] Cai, Y., Wang, H., Wong, S., and Wang, L. Leveraging design rules to improve software architecture recovery. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures* (Vancouver, Canada, June 2013), QoSA '13, ACM, pp. 133–142.

[19] Cataldo, M., Mockus, A., Roberts, J., and Herbsleb, J. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering 35*, 6 (2009), 864–878.

[20] Chen, C., Liaw, A., and Breiman, L. Using random forest to learn imbalanced data. Tech. rep., University of California, Berkeley, Statistics Department, 2004.

[21] Chidamber, S. R., and Kemerer, C. F. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng. 20*, 6 (June 1994), 476–493.

[22] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J. *Documenting Software Architectures: Views and Beyond*. Pearson Education, Oct. 2010.

[23] Cohen, J., and Cohen, J. *Applied multiple regression/correlation analysis for the behavioral sciences*. L. Erlbaum Associates, Mahwah, N.J., 2003.

[24] Cooray, D., Kouroshfar, E., Malek, S., and Roshandel, R. Proactive self-adaptation for improving the reliability of mission-critical, embedded, and mobile software. *IEEE Transactions on Software Engineering 39*, 12 (Dec 2013), 1714–1735.

[25] Corazza, A., Di Martino, S., Maggio, V., and Scanniello, G. Investigating the use of lexical information for software system clustering. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on* (March 2011), pp. 35–44.

[26] D'Ambros, M., Gall, H., Lanza, M., and Pinzger, M. *Analysing software repositories to understand software evolution.* Springer, 2008.

[27] D'Ambros, M., Lanza, M., and Robbes, R. On the Relationship Between Change Coupling and Software Defects. In *16th Working Conference on Reverse Engineering* (Lille, France, Oct. 2009), pp. 135–144.

[28] D'Ambros, M., Lanza, M., and Robbes, R. An extensive comparison of bug prediction approaches. In *7th IEEE Working Conference on Mining Software Repositories* (Cape Town, South Africa, May 2010), pp. 31–41.

[29] Ducasse, S., and Pollet, D. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. Softw. Eng. 35*, 4 (July 2009), 573–591.

[30] Eaddy, M., Zimmermann, T., Sherwood, K., Garg, V., Murphy, G., Nagappan, N., and Aho, A. Do Crosscutting Concerns Cause Defects? *IEEE Transactions on Software Engineering 34*, 4 (2008), 497–515.

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

[32] Eick, S., Graves, T., Karr, A., Mockus, A., and Schuster, P. Visualizing software changes. *IEEE Transactions on Software Engineering 28*, 4 (2002), 396–412.

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

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

[35] Esfahani, N., Kouroshfar, E., and Malek, S. Taming uncertainty in self-adaptive software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary, 2011), ESEC/FSE '11, ACM, pp. 234–244.

[36] Farrar, D. E., and Glauber, R. R. Multicollinearity in regression analysis: The problem revisited. *The Review of Economics and Statistics 49*, 1 (1967), pp. 92–107.

[37] Figueiredo, E., Silva, B., Sant'Anna, C., Garcia, A., Whittle, J., and Nunes, D. Crosscutting patterns and design stability: An exploratory analysis. In *IEEE 17th International Conference on Program Comprehension, 2009. ICPC '09* (Vancouver, Canada, May 2009), pp. 138–147.

[38] Gall, H., Hajek, K., and Jazayeri, M. Detection of logical coupling based on product release history. In *International Conference on Software Maintenance* (Bethesda, Maryland, Nov. 1998), pp. 190–198.

[39] Garcia, J. *A unified framework for studying architectural decay of software systems.* PhD thesis, University of Southern California, 2014.

[40] Garcia, J., Ivkovic, I., and Medvidovic, N. A comparative analysis of software architecture recovery techniques. In *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)* (Palo Alto, CA, USA, Nov. 2013), pp. 486–496.

[41] Garcia, J., Krka, I., Mattmann, C., and Medvidovic, N. Obtaining ground-truth software architectures. In *Proceedings of the International Conference on Software Engineering* (San Francisco, CA, USA, May 2013), ICSE '13, IEEE Press, pp. 901–910.

[42] Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. Identifying Architectural Bad Smells. In *13th European Conference on Software Maintenance and Reengineering* (Kaiserslautern, Germany, Mar. 2009), pp. 255–258.

[43] Garcia, J., Popescu, D., Edwards, G., and Medvidovic, N. Toward a Catalogue of Architectural Bad Smells. In *Proceedings of the 5th International Conference on the Quality of Software Architectures: Architectures for Adaptive Software Systems* (East Stroudsburg, PA, USA, June 2009), QoSA '09, Springer-Verlag, pp. 146–162.

[44] Garcia, J., Popescu, D., Mattmann, C., Medvidovic, N., and Cai, Y. Enhancing Architectural Recovery Using Concerns. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering* (Lawrence, KS, USA, Nov. 2011), ASE '11, IEEE Computer Society, pp. 552–555.

[45] Gethers, M., and Poshyvanyk, D. Using Relational Topic Models to capture coupling among classes in object-oriented software systems. In *IEEE International Conference on Software Maintenance (ICSM)* (Timisoara, Romania, Sept. 2010), pp. 1–10.

[46] Graves, T., Karr, A., Marron, J., and Siy, H. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering 26*, 7 (2000), 653–661.

[47] Hassaine, S., Guéhéneuc, Y., Hamel, S., and Antoniol, G. Advise: Architectural decay in software evolution. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on* (2012), IEEE.

[48] Hassan, A. E. Predicting faults using the complexity of code changes. In *31st International Conference on Software Engineering* (Vancouver, Canada, May 2009), ICSE '09, IEEE Computer Society, pp. 78–88.

[49] Hochstein, L., and Lindvall, M. Combating architectural degeneration: a survey. *Inf. Softw. Technol. 47*, 10 (July 2005), 643–656.

[50] Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K.-i., Adams, B., and Hassan, A. E. Revisiting Common Bug Prediction Findings Using Effort-aware Models. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance* (Timisoara, Romania, Sept. 2010), ICSM '10, IEEE Computer Society, pp. 1–10.

[51] Kim, S., Whitehead, E., and Zhang, Y. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering 34*, 2 (2008), 181–196.

[52] Kobayashi, K., Kamimura, M., Kato, K., Yano, K., and Matsuo, A. Feature-gathering dependency-based software clustering using dedication and modularity. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on* (Sept 2012), pp. 462–471.

[53] Kouroshfar, E. Studying the effect of co-change dispersion on software quality. In *Proceedings of the International Conference on Software Engineering, ACM Student Research Competition Track,* (San Francisco, CA, USA, 2013), ICSE '13, IEEE Press, pp. 1450–1452.

[54] Kouroshfar, E., Garcia, J., and Malek, S. Architectural decay prediction from evolutionary history of software. *Submitted to IEEE Transactions on Software Engineering* (2016).

[55] Kouroshfar, E., Mirakhorli, M., Bagheri, H., Xiao, L., Malek, S., and Cai, Y. A study on the role of software architecture in the evolution and quality of software. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (Florence, Italy, 2015), MSR '15, IEEE Press, pp. 246–257.

[56] Kouroshfar, E., Yaghoubi Shahir, H., and Ramsin, R. Process patterns for component-based software development. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering* (East Stroudsburg, PA, USA, 2009), CBSE '09, Springer-Verlag, pp. 54–68.

[57] Kramer, J., and Magee, J. Self-managed systems: an architectural challenge. In *Int'l Conf. on Software Engineering* (Minneapolis, Minnesota, May 2007), pp. 259–268.

[58] Kruchten, P. The 4+1 view model of architecture. *Software, IEEE 12*, 6 (Nov 1995), 42–50.

[59] Kruchten, P. Architecture blueprints–the '4+1' view model of software architecture. In *Tutorial Proceedings on Ada's Role in Global Markets: solutions for a changing complex world* (Anaheim, CA, USA, Nov. 1995), TRI-Ada '95, ACM, pp. 540–555.

[60] Le, D., Behnamghader, P., Garcia, J., Link, D., Shahbazian, A., and Medvidovic, N. An empirical study of architectural change in open-source software systems. *To appear in the 12th Working Conference on Mining Software Repositories* (2015).

[61] Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering 34*, 4 (July 2008), 485–496.

[62] Leszak, M., Perry, D., and Stoll, D. A case study in root cause defect analysis. In *International Conference on Software Engineering* (Limerick, Ireland, June 2000), pp. 428–437.

[63] Li, Z., Gittens, M., Murtaza, S., Madhavji, N., Miranskyy, A., Godwin, D., and Cialini, E. Analysis of pervasive multiple-component defects in a large software system. In *IEEE International Conference on Software Maintenance* (Edmonton, Alberta, Sept. 2009), pp. 265–273.

[64] Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidovic, N., and Kroeger, R. Comparing software architecture recovery techniques using accurate dependencies. In *Proceedings of the 37th International Conference on Software Engineering* (2015).

[65] Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidović, N., and Kroeger, R. Comparing software architecture recovery techniques using accurate dependencies. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2* (Florence, Italy, 2015), ICSE '15, pp. 69–78.

[66] Martin, R. C., and Martin, M. *Agile principles, patterns, and practices in C#*. Prentice Hall, Upper Saddle River, NJ, 2007.

[67] Menzies, T., Greenwald, J., and Frank, A. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering 33*, 1 (2007), 2–13.

[68] Mirakhorli, M., Carvalho, J., Cleland-Huang, J., and Mader, P. A Domain-Centric Approach for Recommending Architectural Tactics to Satisfy Quality Concerns. In *Third International Workshop on the Twin Peaks of Requirements and Architecture* (Rio de Janeiro, Brazil, July 2013), pp. 1–8.

[69] Mirakhorli, M., Fakhry, A., Grechko, A., Wieloch, M., and Cleland-Huang, J. Archie: A tool for detecting, monitoring, and preserving architecturally significant code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China, 2014), FSE 2014, pp. 739–742.

[70] Mirakhorli, M., Shin, Y., Cleland-Huang, J., and Cinar, M. A tactic-centric approach for automating traceability of quality concerns. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland, 2012), ICSE '12, pp. 639–649.

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

[72] Mo, R., Cai, Y., Kazman, R., and Xiao, L. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. In *2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)* (May 2015), pp. 51–60.

[73] Mockus, A., and Weiss, D. M. Predicting risk of software changes. *Bell Labs Technical Journal 5*, 2 (2000), 169–180.

[74] Murphy, G., Notkin, D., and Sullivan, K. Software reflexion models: Bridging the gap between design and implementation. *IEEE TSE 27*, 4 (2001), 364–380.

[75] Nagappan, N., and Ball, T. Use of relative code churn measures to predict system defect density. In *27th International Conference on Software Engineering* (St. Louis, Missouri, May 2005), pp. 284–292.

[76] Nagappan, N., and Ball, T. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *First International Symposium on Empirical Software Engineering and Measurement* (Madrid, Spain, Sept. 2007), pp. 364–373.

[77] Nagappan, N., Ball, T., and Zeller, A. Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China, May 2006), ICSE '06, ACM, pp. 452–461.

[78] Nagappan, N., Murphy, B., and Basili, V. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany, May 2008), ICSE '08, ACM, pp. 521–530.

[79] Offutt, J., Abdurazik, A., and Schach, S. R. Quantitatively measuring object-oriented couplings. *Software Quality Journal 16*, 4 (Dec. 2008), 489–512.

[80] Oreizy, P., Medvidovic, N., and Taylor, R. N. Architecture-based runtime software evolution. In *Int'l Conf. on Software Engineering* (Kyoto, Japan, Apr. 1998), pp. 177–186.

[81] Ostrand, T., Weyuker, E., and Bell, R. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering 31*, 4 (2005), 340–355.

[82] Perry, D. E., and Wolf, A. L. Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes 17*, 4 (Oct. 1992), 40–52.

[83] Poshyvanyk, D., Marcus, A., Ferenc, R., and Gyimthy, T. Using Information Retrieval Based Coupling Measures for Impact Analysis. *Empirical Softw. Engg. 14*, 1 (Feb. 2009), 5–32.

[84] Posnett, D., D'Souza, R., Devanbu, P., and Filkov, V. Dual Ecological Measures of Focus in Software Development. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA, May 2013), ICSE '13, IEEE Press, pp. 452–461.

[85] Praditwong, K., Harman, M., and Yao, X. Software module clustering as a multi-objective search problem. *Software Engineering, IEEE Transactions on 37*, 2 (March 2011), 264–282.

[86] Prinzie, A., and Van den Poel, D. Random multiclass classification: Generalizing random forests to random mnl and random nb. In *Database and Expert Systems Applications*, R. Wagner, N. Revell, and G. Pernul, Eds., vol. 4653 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 349–358.

[87] Rahman, F., and Devanbu, P. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering* (Honolulu, Hawaii, May 2011), pp. 491–500.

[88] Rahman, F., and Devanbu, P. How, and Why, Process Metrics Are Better. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA, May 2013), ICSE '13, IEEE Press, pp. 432–441.

[89] Rosik, J., Le Gear, A., Buckley, J., Babar, M. A., and Connolly, D. Assessing architectural drift in commercial software development: a case study. *Software: Practice and Experience* (2011).

[90] Sangwan, R. S., Vercellone-Smith, P., and Neill, C. J. Use of a multidimensional approach to study the evolution of software complexity. *Innovations in Systems and Software Engineering* (2010).

[91] Sant'Anna, C., Figueiredo, E., Garcia, A., and Lucena, C. On the modularity of software architectures: A concern-driven measurement framework. In *Software Architecture*, F. Oquendo, Ed., vol. 4758 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 207–224.

[92] Sarkar, S., Kak, A., and Rama, G. Metrics for measuring the quality of modularization of large-scale object-oriented software. *Software Engineering, IEEE Transactions on 34*, 5 (Sept 2008), 700–720.

[93] Sarkar, S., Rama, G., and Kak, A. Api-based and information-theoretic metrics for measuring the quality of software modularization. *Software Engineering, IEEE Transactions on 33*, 1 (Jan 2007), 14–32.

[94] Schroter, A., Zimmermann, T., and Zeller, A. Predicting Component Failures at Design Time. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering* (Rio de Janeiro, Brazil, Sept. 2006), ISESE '06, ACM, pp. 18–27.

[95] Schwanke, R., Xiao, L., and Cai, Y. Measuring architecture quality by structure plus history analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA, May 2013), ICSE '13, IEEE Press, pp. 891–900.

[96] Shahir, H., Kouroshfar, E., and Ramsin, R. Using design patterns for refactoring real-world models. In *35th Euromicro Conference on Software Engineering and Advanced Applications, 2009. SEAA '09.* (Aug 2009), pp. 436–441.

[97] Shihab, E., Mockus, A., Kamei, Y., Adams, B., and Hassan, A. E. High-impact defects: a study of breakage and surprise defects. In *19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (Szeged, Hungary, Sept. 2011), ESEC/FSE '11, ACM, pp. 300–310.

[98] Sliwerski, J., Zimmermann, T., and Zeller, A. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes 30*, 4 (May 2005), 1–5.

[99] Tan, P.-N., Steinbach, M., and Kumar, V. *Introduction to Data Mining.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[100] Turhan, B., Menzies, T., Bener, A. B., and Di Stefano, J. On the Relative Value of Cross-company and Within-company Data for Defect Prediction. *Empirical Softw. Engg. 14*, 5 (Oct. 2009), 540–578.

[101] Tzerpos, V., and Holt, R. C. ACDC: An Algorithm for Comprehension-Driven Clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering* (Washington, DC, USA, Nov. 2000), WCRE '00, IEEE Computer Society, p. 258.

[102] van Gurp, J., Brinkkemper, S., and Bosch, J. Design preservation over subsequent releases of a software product: A case study of Baan ERP: Practice articles. *J. Softw. Maint. Evol. 17*, 4 (July 2005), 277–306.

[103] Walker, R. J., Rawal, S., and Sillito, J. Do crosscutting concerns cause modularity problems? In *20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina, Nov. 2012), FSE '12, ACM, pp. 49:1–49:11.

[104] Waterman, M., Noble, J., and Allan, G. How much up-front?: A grounded theory of agile architecture. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy, 2015), ICSE '15, pp. 347–357.

[105] Wermelinger, M., Yu, Y., Lozano, A., and Capiluppi, A. Assessing architectural evolution: a case study. *Empirical Software Engineering* (2011).

[106] Wong, S., Cai, Y., Kim, M., and Dalton, M. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering* (Honolulu, Hawaii, May 2011), ICSE '11, ACM, pp. 411–420.

[107] Wu, J., Hassan, A., and Holt, R. Comparison of clustering algorithms in the context of software evolution. In *21st IEEE International Conference on Software Maintenance* (Budapest, Hungary, Sept. 2005), pp. 525–535.

[108] Zhou, Y., Xu, B., Leung, H., and Chen, L. An In-depth Study of the Potentially Confounding Effect of Class Size in Fault Prediction. *ACM Trans. Softw. Eng. Methodol. 23*, 1 (Feb. 2014), 10:1–10:51.

[109] Zimmermann, T., Diehl, S., and Zeller, A. How history justifies system architecture (or not). In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of* (2003), IEEE.

[110] Zimmermann, T., and Nagappan, N. Predicting Subsystem Failures using Dependency Graph Complexities. In *The 18th IEEE International Symposium on Software Reliability* (Trollhattan, Sweden, Nov. 2007), pp. 227–236.

[111] Zimmermann, T., and Nagappan, N. Predicting Defects Using Network Analysis on Dependency Graphs. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany, May 2008), ICSE '08, ACM, pp. 531–540.

[112] Zimmermann, T., Premraj, R., and Zeller, A. Predicting Defects for Eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering* (Minneapolis, MN, USA, May 2007), PROMISE '07, IEEE Computer Society, pp. 9–.

Curriculum Vitae

Ehsan Kouroshfar started his PhD with the Department of Computer Science at George Mason University (GMU) in 2009. His current research mainly focuses on empirical software engineering, mining software repositories, and software architecture. Ehsan received his MS degree in Computer Engineering with an emphasis on Software Engineering from Sharif University of Technology (SUT) in 2009 and his BS degree in Computer Engineering with an emphasis on Software Engineering from Amirkabir University of Technology (AUT) in 2006.