Assessing code decay by detecting software architecture violations

By

Ajay Bandi

A Dissertation
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

December 2014

Copyright by

Ajay Bandi

2014

Assessing code decay by detecting software architecture violations

By

Ajay Bandi

Approved:

Edward B. Allen
(Major Professor, Co-Dissertation
Director, and Graduate Coordinator)

Byron J. Williams
(Co-Dissertation Director)

Eric Hansen
(Committee Member)

Tomasz Haupt
(Committee Member)

Jason M. Keith
Interim Dean
Bagley College of Engineering

Name: Ajay Bandi

Date of Degree: December 13, 2014

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Edward B. Allen

Title of Study: Assessing code decay by detecting software architecture violations

Pages of Study: 176

Candidate for Degree of Doctor of Philosophy

Code decay is a gradual process that negatively impacts the quality of a software system. Developers need trusted measurement techniques to evaluate whether their systems have decayed. This dissertation aims to assess code decay by discovering software architectural violations. Our methodology uses Lightweight Sanity Check for Implemented Architectures to derive architectural constraints represented by can-use and cannot-use phrases. Our methodology also uses Lattix to discover architectural violations. We also introduce measures that indicate code decay in a software system. We conducted two case studies of proprietary systems (9 versions of System A and 14 versions of System B) to demonstrate our methodology for assessing code decay. Resulting architectural constraints and architectural violations were validated by the expert of each system. Our results show that the net violations of System A increased from one version to other version even though there were architecture changes. However, in System B, the number of net violations decreased after changes in the architecture.

The proposed code decay metrics can give managers insight into the process of software development, the history of the software product, and the status of unresolved violations. Our results and qualitative analysis showed that the methodology was effective and required a practical level of effort for moderate sized software systems. Code decay values increase because of an increase in the number of violations over multiple versions.

We compared our code decay measures with definitions in the literature, namely coupling metrics. In addition, our empirical results showed that coupling is qualitatively correlated with the size of a system as expected. We note that coupling is an opportunity for architectural violations. We concluded that coupling is not consistently related to violations.

Key words: architectural constraints, architectural violations, code decay, coupling, maintainability, metrics, reverse engineering, software architecture, software evolution

DEDICATION

This dissertation is dedicated to the memory of my peternal uncles and grandparents. Also dedicated to my beloved parents Sri. Subba Rao Bandi, Smt. Padmaja Bandi, my brother Vijay Bandi, my wife Smt. Tharunima Bandi, other family members and friends who supported me.

ACKNOWLEDGEMENTS

I thank my major professor and dissertation co-director, Dr. Edward B. Allen, for all his thoughts and guidance in this dissertation. I thank my dissertation co-director, Dr. Byron J. Williams, for his help in conducting the systematic mapping study, which eventually shaped this work. A sincere thanks to Dr. Tomasz Haupt, for funding me for four years during my doctoral studies. The software development work under his supervision motivated me to perform research on code decay. I thank Dr. Eric A. Hansen for his suggestions and comments during the intermediate reviews of this research. Once again I thank all the committee members. I could not have completed my work without all of your help and constant encouragement.

I thank the Computer Science and Engineering Department for funding me during the final year of my studies. I thank the Empirical Software Engineering group at Mississippi State University for their helpful suggestions. I would like to thank Rakesh Pakalapati for helping me in solving various technical issues with tools we used in this research. My sincere thanks to all the participants, who spent their valuable time in our case studies. I would also like to acknowledge my wife Tharunima Bandi and an undergraduate student Veera Venkata Satayanarayana Reddy Karri for helping me in formatting the figures and tables in this document.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES vii

LIST OF FIGURES ix

DEFINITIONS xii

CHAPTER

1. INTRODUCTION 1

1.1 Research hypothesis 2
1.2 Research questions 2
1.3 Relevance 4
1.4 Overview 6

2. RELATED WORK 7

2.1 Empirical evidence of code decay: A systematic mapping study . 7
2.1.1 Study methodology 8
2.1.2 Results 14
2.1.2.1 Human-based approach 14
2.1.2.2 Metric-based approach 19
2.1.2.3 Metrics 26
2.1.3 Discussion 33
2.1.4 Conclusions 36
2.2 Architecture evaluation techniques 37
2.3 Architectural constraints 39
2.4 Architecture conformance techniques 40

3. TOOLS 44

3.1 Lattix 44
3.2 LiSCIA 44

4. METHODOLOGY FOR PRACTITIONERS 46

4.1 Choose an initial or refactored version 47
4.2 Derive architectural constraints 47
4.3 Discover current architectural violations 49
4.4 Find new, solved, and reoccurred violations 51
4.5 Assess code decay over multiple versions 52

5. CASE STUDY DESIGN 54

5.1 Research goals 54
5.2 Criteria for selecting cases and subjects 55
5.3 Study procedure 55
5.3.1 Categorize architectural violations 57
5.3.2 Compute coupling metrics 57
5.3.3 Calculate the rate of coupling metrics 61
5.4 Data collection procedure 62
5.5 Analysis procedure 63
5.6 Validation procedure 63
5.7 Pilot study 64

6. CASE STUDY OF SYSTEM A 68

6.1 Objective 68
6.2 Case and subjects selection 68
6.3 Results and analysis 69
6.3.1 Derive architectural constraints 69
6.3.2 Discover architectural violations 74
6.3.3 Find new, solved, and reoccurred violations 78
6.3.4 Assess code decay over multiple versions 79
6.3.5 Comparison of results 84

7. CASE STUDY OF SYSTEM B 94

7.1 Objective 94
7.2 Case and subjects 94
7.3 Results and analysis 95
7.3.1 Derive architectural constraints 95
7.3.2 Discover architectural violations 100
7.3.3 Find new, solved, and reoccurred violations 104

7.3.4 Assess code decay over multiple versions 106
7.3.5 Comparision of results 111

8. DISCUSSION 123

8.1 Research questions revisited 123
8.1.1 Deriving architectural constraints 123
8.1.2 Discovering architectural violations 125
8.1.3 Assessing code decay 126
8.1.4 Comparison of results 127
8.2 Implications 129
8.3 Threats to validity 129

9. CONCLUSIONS 131

9.1 Conclusions 131
9.2 Contributions 134
9.3 Publications 135
9.4 Future research directions 135

REFERENCES 138

APPENDIX

A. DETAILS OF MAPPING STUDY 145

A.1 Search strategy 146
A.2 Inclusion and exclusion criteria 147
A.3 Quality assessment criteria 148
A.4 Data extraction 150
A.5 Data mapping 150

B. DETAILS OF LISCIA 157

C. SYSTEM A ARTIFACTS 161

D. SYSTEM B ARTIFACTS 165

E. SOURCE CODE 169

E.1 Source code for CBMCCalculator 170
E.2 Source code for CBMCalculator 174

LIST OF TABLES

2.1 Studies by research method . . . 11

2.2 Distribution of primary studies . . . 13

2.3 Definitions . . . 16

2.4 Code decay detection techniques . . . 18

2.5 Metrics . . . 27

2.6 Architecture evaluation techniques . . . 38

5.1 Interview guide . . . 56

5.2 Pilot system details . . . 65

6.1 Details of System A . . . 69

6.2 Violation count over multiple versions of System A . . . 75

6.3 Values of development time of System A . . . 80

6.4 Code decay values of System A . . . 80

6.5 Coupling values of System A . . . 85

7.1 Details of System B . . . 95

7.2 Violations count over multiple versions of System B . . . 102

7.3 Values of development time of System B . . . 107

7.4 Code decay values of System B . . . 108

7.5 Coupling values of System B . . . 112

9.1 Publication plan 136

A.1 Inclusion and exclusion criteria 149

A.2 Quality assessment criteria 151

A.3 Quality assessment criteria results for case studies/archival studies 154

A.4 Quality assessment criteria results for controlled/quasi-experiments 155

A.5 Quality assessment criteria results for experiences/surveys 155

A.6 Data extraction form 156

LIST OF FIGURES

2.1 Overview of systematic mapping study 9

2.2 Overview of article selection 12

2.3 Different forms of code decay 15

2.4 Violations in layered architecture style 40

2.5 Violations in mediator design pattern 41

4.1 Deriving architectural constraints 48

4.2 Discovering architectural violations 50

5.1 Example architectural design 59

5.2 Intermodule coupling of the example system 59

5.3 Intermodule class references for the example system 60

6.1 The number of classes in System A per version 70

6.2 The number of interfaces in System A per version 70

6.3 Percentage of violations in System A 76

6.4 Number of net violations in System A 77

6.5 New and solved violations in System A 80

6.6 Code decay for each release in System A 82

6.7 Code decay since beginning of the System A 83

6.8 $CBM_{net,i}$ values of System A 86

6.9 $CBMC_{net,i}$ values of System A 86

6.10 $CBM_{net,i}$ vs. size of System A 87

6.11 $CBMC_{net,i}$ vs. size of System A 87

6.12 $Rate\Delta CBM_i$ for each release in System A 89

6.13 $Rate\Delta CBMC_i$ for each release in System A 90

6.14 $RateCBM_{net,i}$ since the beginning of the System A 91

6.15 $RateCBMC_{net,i}$ since the beginning of the System A 91

6.16 $CBM_{net,i}$ vs. net violations in System A 92

6.17 $CBMC_{net,i}$ vs. net violations in System A 93

7.1 The number of classes in System B per version 96

7.2 The number of interfaces in System B per version 96

7.3 Percentage of violations in System B 102

7.4 Number of net violations in System B 103

7.5 New and solved violations in System B 105

7.6 Code decay for each release in System B 109

7.7 Code decay since beginning of the System B 110

7.8 $CBM_{net,i}$ values of System B 114

7.9 $CBMC_{net,i}$ values of System B 114

7.10 $CBM_{net,i}$ vs. size of System B 115

7.11 $CBMC_{net,i}$ vs. size of System B 115

7.12 $Rate\Delta CBM_i$ for each release in System B 117

7.13 $Rate\Delta CBMC_i$ for each release in System B 117

7.14 $RateCBM_{net,i}$ since the beginning of the System B 118

7.15 $RateCBMC_{net,i}$ since the beginning of the System B 119

7.16 $CBM_{net,i}$ vs. net violations in System B 120

7.17 $CBMC_{net,i}$ vs. net violations in System B 120

7.18 Quadrants 121

9.1 Contributions 135

C.1 System A conceptual architecture 162

C.2 System A dependency structure matrix 163

C.3 System A high level architecture diagram by participants 164

D.1 System B conceptual architecture 166

D.2 System B dependency structure matrix 167

D.3 System B high level architecture diagram by participants 168

DEFINITIONS

Reverse engineering tool. A tool which is used to extract the module calls from source code. It also detects the violations of architectural constraints. We used the Lattix reverse engineering tool in this dissertation.

Conceptual architecture. The high level architecture diagram which shows the interactions between modules in the form of boxes and arrows. Interactions are represented by uses relationships.

Dependency structure matrix (DSM). Representation of all the module calls in the form of matrix to visualize the organization of the project.

LiSCIA. Lightweight Sanity Check for Implemented Architectures. LiSCIA is a structured manual evaluation method which identifies software architecture problems.

Architectural constraints. The architectural rules can be represented by can-use/cannot-use phrases.

Architectural violations. An architectural violation is a particular piece of code that does not follow specified architectural constraints or does not conform to the conceptual architecture.

Code decay. Code decay is a gradual process that degrades the maintainability of the software system. For example, code decay includes violations in coding standards, flaws in implementing architecture of the system, violations in the design/code level per unit time etc.

Net violations ($V_{net,i}$). The number of violations discovered by Lattix for a version of interest, i, in the software.

New violations ($V_{new,i}$). The number of unique violations that occurred in a version of interest, i, excluding the violations that occurred in the previous version, $i-1$. For the initial version of a system, the number of new violations is equal to the net violations.

Solved violations ($V_{solved,i}$). The number of violations that are missing from the previous version, $i-1$. For the initial version of a system, the number of solved violations is equal to zero.

Reoccurred violations ($V_{reoccur,i}$). The number of solved violations in the previous versions that reappeared in a version of interest i. For the initial version of a system, the number of solved violations is equal to zero.

time (t_i). The development time of one version of interest i.

Time (T_i). The cumulative development time from beginning of the project until the version of interest i.

Code decay for version i (cd_i). This is a measure of type of code decay for a given version of interest i since the last release.

Net code decay (CD_i). For a version of interest i, the net code decay, which is a measure of a type of code decay, is defined as the net violations divided by the cumulative development time from the beginning of the project when coupling was zero.

Overall code decay (CD_n). The value of overall code decay, which is a measure of a type of code decay, is calculated by considering from the initial version to the final version of interest.

Coupling-Between-Modules (CBM). CBM is the number of non-directional, distinct, intermodule references.

Coupling-Between-Module-Classes (CBMC). $CBMC$ is the number of non-directional, distinct, intermodule class-to-class references.

Net CBM ($CBM_{net,i}$). The value of CBMat a given version of interest i.

Net CBMC ($CBMC_{net,i}$). The value of $CBMC$ at a given version of interest i.

Change in CBM (ΔCBM_i). The difference of CBM values between the version of interest i and its previous version.

Change in CBMC ($\Delta CBMC_i$). The difference of $CBMC$ values between the version of interest i and its previous version.

Rate of change in CBMC ($Rate\Delta CBM_i$). The difference of CBM values between the version of interest i and its previous version divided by time since last release (t_i).

Rate of change in CBMC ($Rate\Delta CBMC_i$). The difference of $CBMC$ values between the version of interest i and its previous version divided by time since last release (t_i).

Net rate of CBM ($RateCBM_{net,i}$). For a version of interest i, the net rate of CBM is defined as the net coupling between modules $CBM_{net,i}$ divided by the cumulative development time from the beginning of the project when coupling was zero.

Net rate of CBM ($RateCBMC_{net,i}$). For a version of interest i, the net rate of $CBMC$ is defined as the net coupling between modules $CBMC_{net,i}$ divided by the cumulative development time from the beginning of the project when coupling was zero.

CHAPTER 1

INTRODUCTION

Research in software evolution shows that violations of architecture and design rules cause code to decay [18, 22, 39]. These violations are due to new interactions between modules that were originally unintended in the planned design [39, 67]. Such violations may be caused by adding new functionality, or modifying existing functionality to implement changing requirements or to repair defects. Such changes are inconsistent with the planned architecture and design principles. As a result, the system becomes more complex, hard to maintain, and defect prone [18, 51, 72]. Often, redesign or reengineering of the whole system is the only practical solution for this problem [22]. The phenomenon of gradual increase in software complexity due to unintended interactions between modules that are hard to maintain has been termed code decay and architectural degeneration [18, 26]. In this dissertation, code decay refers to the rate of violations of architecture, design rules and coding standards over time that make software more difficult to modify. This research focuses on violation of architectural constraints. The main goal of this dissertation is to find ways to derive architectural constraints, to detect architectural violations, and to assess code decay of software over multiple versions.

1.1 Research hypothesis

This section presents the research hypothesis of this work and the definitions of terms used in our research hypothesis.

> *Given source code, a method can be developed to detect changes in the maintainability of a system by identifying the architectural violations over multiple versions.*

Maintainability is the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers [27]. We assume that architectural violations degrade maintainability of a software system. Code decay is a gradual process that degrades the maintainability of the software system. For example, code decay includes violations in coding standards, flaws in implementing architecture of the system, violations in the design/code level per unit time etc. In this dissertation, we quantify “code decay” as the number of architectural violations per unit time in work weeks. The details of code decay are presented in Section 4.5 of Chapter 4. An architectural violation is a particular piece of code that does not follow specified architectural constraints or does not conform to the conceptual architecture, represented by can-use/cannot-use rules.

1.2 Research questions

The major focus of this dissertation is identifying techniques and metrics to assess code decay. The following are research questions to assess support for the research hypothesis.

1. What is an effective method to derive architectural constraints from the source code?
2. What is an effective method to discover the extent of violations in architectural constraints?
3. What does the existence and repair of architectural violations over time imply about code decay?

4. How does our definition of code decay compare to definitions of code decay in the literature, specifically, is our definition of code decay redundant with coupling metrics?

Brunet et al. [6] used architecture documentation or the uses relationship diagram of the older version of the system for considering the architectural constraints. They didn't use any formal method to derive architectural constraints. Researchers used architecture conformance techniques such as reflexion models [48] and heuristics [43, 44]. The disadvantages of reflexion model requires successive refinements or iterations in the high level mental model to discover the absences and divergences in the source code. The expressiveness of the reflexion models is limited to regular expressions but no other types of rules. In addition, these models focus on the conformance of the design and implementation and do not deal with different architecture styles (e.g., layered architecture). On the other hand, the drawback of the heuristics technique is using many threshold values in heuristics. This architecture conformance process based on the proposed heuristics should follow an iterative approach — running the heuristics several times, starting with rigid thresholds. After each execution, the new warnings should be evaluated by the architect. The selecting of threshold values may takes several iterations. Our methodology overcome these disadvantages and also assess code decay over multiple versions. In the systematic mapping study of code decay, Bandi, Williams, and Allen [1] concluded that the coupling related metrics were used to assess different forms of code decay. This dissertation proposed a complementary and alternative approach to assess code decay that uses architecture violations over development time.

The contributions of this dissertation are the following.

1. We perform a literature review on code decay and conducted a systematic mapping study on code decay that gives the classification of the code decay and its related terms, classification of code decay detection techniques (human-based and metric-based approaches), and the metrics used to measure the code decay.

2. We propose a methodology to derive architectural constraints that uses a reverse engineering tool and LiSCIA. In our case studies we used Lattix as our reverse engineering tool.

3. We propose a methodology that also uses a reverse engineering tool to discover architectural violations and validate them. In our case studies we used Lattix as our reverse engineering tool.

4. We also propose an alternative and complementary method to assess code decay which uses code decay indicator measures (cd_i, CD_i, and CD_n). The empirical evidence and qualitative assessment shows our methodology is practical for deriving architectural constraints, discovering architectural violations, and assessing code decay.

5. We qualitatively compare our code decay results with coupling metrics (CBM and $CBMC$).

1.3 Relevance

Identifying and minimizing code decay is important to software engineering practitioners who are focused on improving software quality during software maintenance. Code decay is an attribute that is evident only in retrospect. It is usually assumed that "decay" is a gradual process that goes unnoticed until a crisis occurs. One can detect decay by comparing measured attributes from the past with current values, and determine that quality has "decayed." A challenge for researchers is to find ways of detecting incipient "decay" well before a crisis develops.

Eick et al. [18] define code decay as code being harder to change than it should be. They assessed code decay in a 15 year old real-time telephone switching software system using change management data. The system consisted of fifty major subsystems and about

five thousand modules in C and C++. They used measures such as the number of changes to a file, the number of files touched to implement a change, sizes of modules, the average age of constituent lines of modules, fault potential, and change effort. Their analysis confirmed that the system decayed due to successive modifications.

As another example, Godfrey and Lee [22] analyzed the open source project of the Mozilla web browser release M9 by extracting architectural models using reverse engineering tools. The Mozilla web browser (M9) consisted of more than 2 million lines of source code in more than seven thousand header and implementation files in C and C++. After a thorough assessment of architecture models, Godfrey and Lee concluded that either Mozilla's architecture has decayed significantly in its relatively short lifetime or it was not carefully architected in the first place [22].

Software metrics characterize attributes of software. Product metrics measure attributes of development artifacts, such as source code and design diagrams. Lines of code and McCabe complexity are two of the best known metrics in this category. Process metrics measure attributes of the development process and events associated with the product, such as effort spent, defects discovered, and number of changes to code. Considerable research has modeled relationships between attributes that can be measured early and those measured later [23, 57]. For example, a statistical model might predict which modules are more likely to have bugs in the future, based on attributes measured early [57].

1.4 Overview

The remainder of this dissertation is organized as follows. Chapter 2 describes the related work. Chapter 3 explains the tools used in this research. Chapter 4 details our proposed methodology for practitioners. Chapter 5 presents the case study design. Chapter 6 reports the System A case study results and its analysis. Chapter 7 presents the System B case study results and its analysis. Chapter 8 discusses answers to our research questions and Chapter 9 presents conclusions of our research.

CHAPTER 2

RELATED WORK

This chapter focuses on background and related work of this research. Section 2.1 presents a systematic mapping study of the literature on empirical evidence of code decay. Section 2.2 discusses architecture evaluation techniques. Section 2.3 presents related work on architecture constraints.

2.1 Empirical evidence of code decay: A systematic mapping study

This systematic mapping study identifies detection techniques and metrics used to measure code decay. This section is based on a paper by Bandi, Williams, and Allen [1]. We followed Kitchenham and Charters [35] approach to perform this study. A systematic mapping study of code decay [1] aims to give a classification and thematic analysis of the literature with respect to code decay detection procedures or methods by aggregating information from empirical evidence. In contrast, the purpose of a systematic literature review is often to "identify, analyze and interpret all available evidence related to a specific research question" [34], [70, p.45]. We chose a mapping study to find the empirical evidence of code decay because of our broad research questions.

The contributions of this review include presentation of various terms used in the literature to describe decay, a categorization of code decay detection techniques, and description

of metrics used to identify code decay. The remainder of this section is organized as follows. Section 2.1.1 describes the details of the research methodology of mapping study. Section 2.1.2 presents the results of the study. Sections 2.1.3 and 2.1.4 discuss the findings and present conclusions of the mapping study respectively.

2.1.1 Study methodology

A systematic mapping study helps to provide a comprehensive overview of the literature and topic categorization in a variety of dimensions (such as architecture violations and design rule violations). Kitchenham et al. [34] provide the key characteristics, the differences between systematic literature reviews and mapping studies, and the benefits of mapping studies in software engineering. The overview of our systematic mapping study is shown in Figure 2.1. The following are the steps in our study.

1. Plan the study
2. Conduct the study
3. Report the study

The mapping study plan includes the following actions.

- Identify the need for the study
- Specify the research questions
- Develop the study protocol

The need for this mapping study is to identify and understand the scope of the empirical research on code decay and its forms. This study clarifies the research and defines future research questions. The major focus of this review is identifying techniques and metrics to

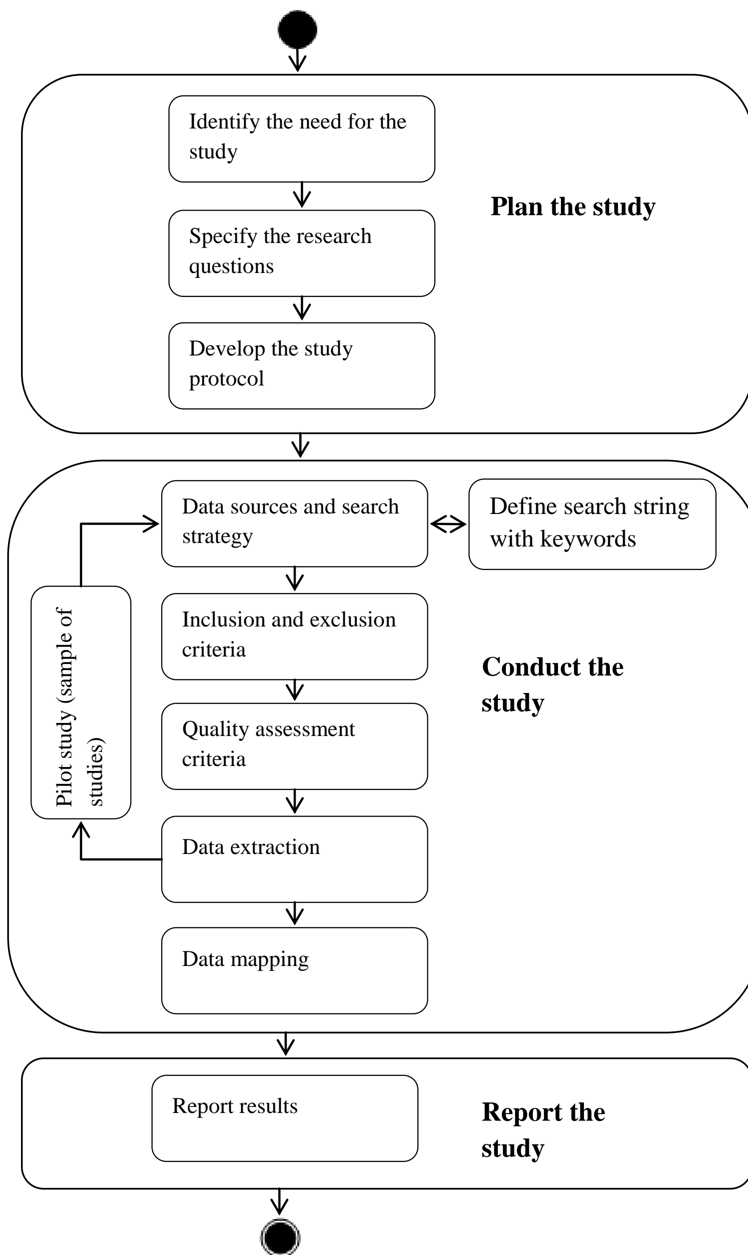

Figure 2.1

Overview of systematic mapping study

assess code decay without including a general literature on fault prediction performance in software engineering [23], or the literature on fault prediction metrics [57]. Our research questions are given below.

> **Research Question 1:** What are the techniques used to detect code decay? (i.e., How is it discovered?)

To answer the above question, we reviewed the literature and categorized code decay detection techniques.

> **Research Question 2:** What metrics are used to evaluate code decay? (i.e., How is it measured?)

To answer this question, a comprehensive tabular overview of code decay metrics is presented that helps software engineering practitioners to assess the severity of code decay.

Following the Kitchenham and Charters [35] guidelines, Dybå and Dingsøyr [15, 16] proposed a review protocol in the systematic review of empirical studies of agile software development. We followed a similar approach to develop our study protocol because our focus is on identifying empirical evidence of code decay. This review protocol includes data sources and search strategy, inclusion and exclusion criteria, quality assessment criteria, a data extraction form, and data mapping. Table A.1, Table A.2 and Table A.6 in Appendix A, shows the inclusion and exclusion criteria, quality assessment criteria, and data extraction forms respectively. Quality evaluation is not essential in mapping studies, but we applied quality criteria assessment when selecting our primary studies.

Conducting the study means performing the study protocol which includes data sources and search strategy. The goal of the search is to identify relevant papers describing code

decay detection and measurement techniques and related concepts. We searched peer-reviewed articles in the following electronic databases.

- ACM Digital Library
- Google Scholar
- IEEE Xplore Electronic Library
- Scopus (includes Science Direct, Elsevier, and Springer)

Figure 2.2 shows the review stages and number of studies selected at each stage. The details of the search strategy are given in Appendix A.

The studies we found cover a range of research topics on architecture violations, design defects and problems with source code. The numbers of publications using each particular research method are listed in Table 2.1. Of these 30 primary studies, 18 were performed on open source projects and 12 on proprietary systems. Table 2.2 presents the publication channels of our primary studies. Most of the studies (75%) were published in conferences, while others appeared in journals. These primary studies are noted as [Primary Study] in the references.

Table 2.1

Studies by research method

Research method	Number	Percent
Case studies and archival studies	26	86
Controlled and quasi experiments	2	7
Experience reports and surveys	2	7
Total	30	100

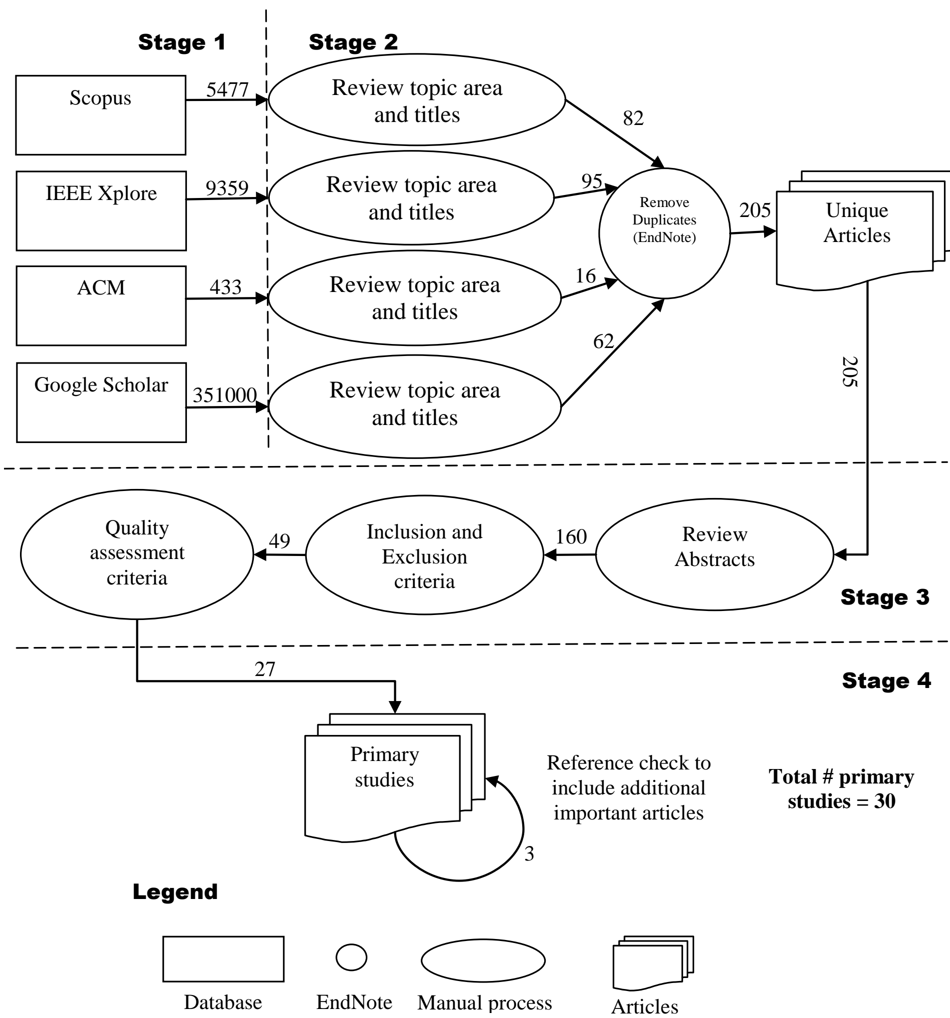

Figure 2.2

Overview of article selection

Table 2.2

Distribution of primary studies

Publication channel	Number
—International Symposium on Empirical Software Engineering and Measurement	4
—Working Conference on Reverse Engineering	4
—International Conference on Software Maintenance	4
—European Conference on Software Maintenance and Reengineering	3
—The Journal of Systems and Software	2
—International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS)	2
—IEEE Transactions on Software Engineering	1
—IEEE Software	1
—Software—Practice and Experience	1
—Communications in Computer and Information Science	1
—International Conference on Software Engineering	1
—IEEE International Software Metric Symposium	1
—Asia Pacific Software Engineering Conference	1
—IEEE Aerospace Conference	1
—International Conference on Software Engineering and Knowledge Engineering	1
—International Workshop on Software Aging and Rejuvenation	1
—Brazilian Symposium on Software Components, Architectures and Reuse	1
Total	30

2.1.2 Results

The results of our review are presented as answers to each research question defined above. During our review of papers we encountered various terms in the literature that relate to code decay. These terms are organized on the basis of architecture, design, and source code. This terminology of code decay is shown in Figure 2.3. The definitions of these terms are given in Table 2.3.

> **Research Question 1:** What are the techniques used to detect code decay?
> (i.e. How is it discovered?)

Table 2.4 gives the summarized view of the different strategies to detect code decay and its categories. Detection of code decay is an important research area. The motivation for the high level classification was the level of potential automation is a natural distinction among the techniques. Research techniques can be broadly categorized as human-based (manual) and metric-based (semi-automated) approaches.

2.1.2.1 Human-based approach

Human-based detection techniques consist of manual visual inspection of source code and architectural artifacts.

Source code inspections are performed subjectively and are guided by questionnaires. In the technique presented by Mäntylä et al. [45], developers manually inspected source code to identify code smells. They identified three different code smells (duplicate code, god class, and long parameter list) by filling out a web-based questionnaire. The assessment was based on subjective evaluation on a seven-point numeric Likert scale. These

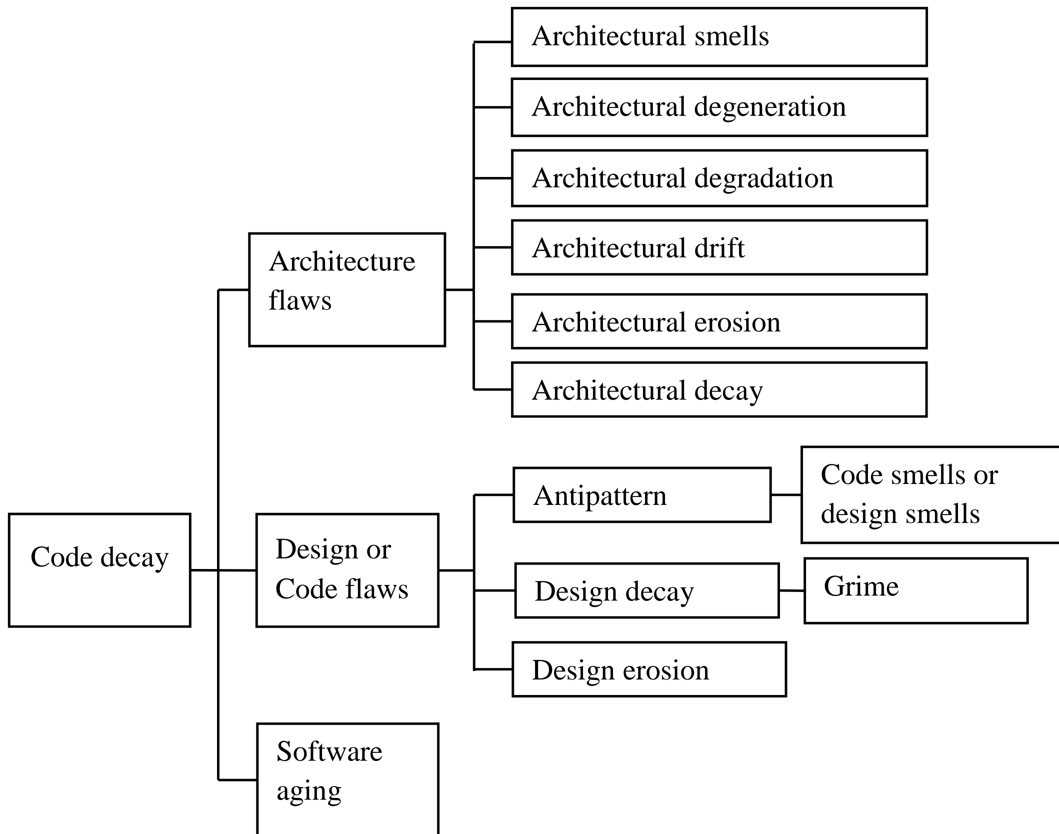

Figure 2.3

Different forms of code decay

Table 2.3

Definitions

Term	Definition	Reference
Code decay	"A unit of code is decayed if it is harder to change than it should be as reflected by COST of the change, INTERVAL to complete the change, and QUALITY of the changed software."	[18]
Architectural smells	"An architectural smell is a commonly (although not always intentionally) used architectural decision that negatively impacts system quality. Architectural smells may be caused by applying a design solution in an inappropriate context, mixing design fragments that have undesirable emergent behaviors, or applying design abstractions at the wrong level of granularity."	[21]
Architecture degeneration	"A system is degenerated when the actual architecture of the system deviates from the planned architecture of the system."	[25]
Architectural drift	"Architectural drift is due to insensitivity about the architecture. This insensitivity leads more to inadaptability than to disasters and results in a lack of coherence and clarity of form, which in turn makes it much easier to violate the architecture."	[55]
Architecture erosion	"Architectural erosion is due to violations in architecture. These violations often lead to an increase in problems in the system and contribute to the increasing brittleness of the system."	[55]
Architecture decay	"Architectural decay is the phenomenon when the concrete (as-built) architecture of a software system deviates from its conceptual (as-planned) architecture where it no longer satisfies the key quality attributes that led to its construction OR when architecture of a software system allows no more changes due to changes introduced in the system over time that render it unmaintainable."	[58]
Design erosion	"It is the phenomenon in which the design of a system becomes less and less suitable to incorporating new features over time."	[25]
Antipattern	"Antipattern describes a recurring situation that has a negative impact on a software project."	[21]

Table 2.3

(continued)

Term	Definition	Reference
Code smells/design smells	Code smells are anomalies in the source code that contribute to the degradation of software design maintainability.	[19]
Design decay	"Design pattern decay is the deterioration of the structural integrity of a design pattern realization."	[28]
Grime	"An increase in code within design pattern participants that does not contribute to the 'mission' of individual design patterns. This added non-pattern code is grime."	[28]
Software aging	The decline in the value of the software over time is known as software aging.	[53]

results do not correlate with code smells found using source code metrics. Similarly, Schumacher et al. [64] detected code smells (god class) manually by inspecting source code. In this study, the subjects were encouraged to "think aloud" as they filled out the questionnaires. They compared the subjective results with the metric values from automated classifiers. In contrast to Mäntylä et al. [45], the results of their study increased the overall confidence in automatic detection of smells. They also found that god classes require more maintenance effort.

Inspection of architecture artifacts is done by subjective evaluations using checklists and by comparing architecture models. Bouwers and van Deursen [5] proposed the Lightweight Sanity Check for Implemented Architectures (LiSCIA) to identify architecture erosion. They provide a checklist of 28 questions based on units of modules, module functionality, module size, and module dependencies. Developers evaluate implemented architec-

Table 2.4

Code decay detection techniques

Detection Technique	Category	Subcategory	Reference
Human-based (manual) approach	Inspection of source code	Filling out questionnaires	[45, 64]
	Inspection of architectural artifacts	Answering checklist questions	[5, 59]
Metric-based (semi-automated) approach	Historical data analysis	Change management data	[18]
		Architecture history	[6, 24]
		Defect-fix history	[38, 50]
		Source code metrics	[58, 67, 68]
	Rule-based interpretations	Heuristics with threshold filter rules	[39, 46, 51, 52, 56, 66]
		Domain specific language rules	[8, 33]
	Model-based techniques	Probabilistic model	[69]
		Graphical model	[30, 62]
		Modularity model	[71]

tures by inspecting the architecture artifacts. The evaluation phase consists of answering a list of questions concerning the architecture elements. LiSCIA provides corresponding actions to the questions to help identify the erosion in an implemented architecture. We used part of LiSCIA in our methodology.

Rosik et al. [59] assessed architectural drift by comparing the implemented architecture with the original architecture of a system using a reflexion model [48]. Developers create and update the code base and associated mappings to the original and implemented architectures. This model displays the architecture in a pictorial representation with nodes and edges. The participants "think aloud" and assess the inconsistencies with implemented architecture to identify violations from the results of the model. Their case study confirmed that architectural drift occurred during the evolution of the system.

Manual detection of code decay and its categories is tedious work. Moreover, this process is time consuming, non-repeatable, and non-scalable. Moreover, manual detection of code smells do not correlate with the results of source code metrics derived from automated classifiers [45, 64].

2.1.2.2 Metric-based approach

In this subsection we review the literature that deals with semi-automated approaches to detect code decay and its categories. The metric-based approach is further divided into four subcategories. They are historical data analysis or mining software repositories, comparison techniques, interpretation of rules, and model based techniques. These are discussed below. The metric details used in these techniques are presented.

The categories of historical data analysis types used for discovery of code decay are change management data, architecture history, and defect-fix history.

In the change management subcategory, Eick et al. [18] dealt with the history of change management data to detect code decay using code decay indices. Change management history includes source code of the feature, modification request, delta, and change to severity levels. Their statistical analysis on this data showed that an increase in the number of files touched per change and decrease in modularity over time yields strong evidence of code decay.

Under the architecture history subcategory, Brunet et al. [6] used the architectural diagrams from several versions of four different open source systems. They found violations in the architectures by applying the reflexion model technique [48]. Using JDepend, Lattix, and design documentation they extracted the high level architecture. They identified more than 3000 architecture violations in the systems they analyzed. We use Lattix in our methodology.

Hassaine et al. [24] proposed a quantitative approach called ADvISE to detect architectural decay in software evolution. They used the architectural histories of three open source systems. The architecture history consisted of architectural diagrams of different versions that were extracted from source code using a tool. The extracted architecture is represented as a set of triples (S, R, T) where S and T are two classes and R is the relationship between two classes. They performed pair-wise matching of the subsequent architectures to identify deviations in the actual architecture from the original architecture by tracking the number of common triples. This procedure was accomplished by matching architectural diagrams

using a bit-vector algorithm. An increase in the number of classes and number of common triples over time was a good indicator of architecture decay.

In the defect-fix history subcategory, Li and Long [38] used defect-fix history to measure architecture degeneration. Defect-fix history consists of information about the release, the component in which the defect occurred, and the number of files changed to fix the defect. To analyze the defect history, they used multiple component defect metrics (e.g., percentage of defects that affected multiple components in a system and the average quantity of files changes to fix a defect). After analyzing the defect-fix history of a compiler system they found that an increase in the value of these metrics between two versions of the system indicates that the architecture has degenerated. Ohlsson et al. [50] performed historical data analysis on the defect-fix reports or source change notices of a large embedded mass storage system to identify code decay. The defect-fix reports consist of description of release and the defect that has to be corrected. Metrics that evaluated the average number of changes, the size, the effort and the coupling were used to identify code decay. The average number of changes and coupling metrics played a major role in identifying code decay in this system. Their results showed that increases in values of these metrics indicate code decay.

Under the source code metrics subcategory, researchers [58, 67, 68] compare the metrics of the source code over different versions of system using the original version to identify code decay. Tvedt et al. [67] compare the interactions between the mediator and the colleagues in the mediator pattern between the two versions of Visual Query Interface (VQI) system (VQI1 and VQI2). They used coupling between modules (CBM) to identify

unintended violations in the mediator design pattern and other misplaced violations. They concluded that the actual design of the system veered from the planned design. Riaz et al. [58] used coupling related metrics to compare two versions of a system. They found that an increase in the value of coupling related metrics indicates architecture decay. Van Gurp and Bosch [68] compared UML design diagrams of an ATM simulator from one version to another version by calculating metrics related to packages, functions and inner classes. Increases in the values of metrics, design decisions, and new requirements during the evolution of the system were associated with design erosion.

Rule-based interpretations are divided into two types. They are 1) Metric heuristics with threshold filters and 2) Domain specific language rules.

In the metric heuristics with threshold filters subcategory, several researchers [46, 51, 52, 56] used metric-based heuristics to detect code/design smells. Marinescu [46] proposed a metric-based approach to detect code/design flaws in an object-oriented system. He identified possible code smells (god class and data class) using the values of metrics as heuristics. An example of one metric is given here: 1) The lower the Weight of Class (WOC) value, the more the class is expected to be a data class and 2) The higher the WOC value, the more the class is expected to be god class. Similarly, he used metric heuristics on other metrics to detect both of these classes in an industrial case study. The threshold values of the metrics are based on expert opinion.

Raţiu et al. [56] used heuristics and threshold values for each metric to detect god classes and data classes. These threshold values are based on the experience of the analyst. The results of this detection technique are suspected code smells. They analyzed different

versions of software to obtain class and system history using the Evolution Matrix method. Their results highlight that this method improves accuracy in detecting god classes and data classes.

Olbrich et al. [51, 52] used heuristics with threshold filter rules to detect god classes, brain classes, and shotgun surgery code smells. The threshold values used in the filtering rules are based on the expert opinion. An example of a detection strategy for god class using metrics is shown below. The presence of god class is represented by '1'. A '0' value indicates that there is no god class. The definitions of the metrics Equation (2.1) is given in Table 2.5.

$$GodClass(C) = \begin{cases} 1, & ((WMC(C) \geq 47) \\ & \wedge (TCC(C) < 0.3) \\ & \wedge (ATFD(C) > 5)) \\ 0, & else \end{cases} \tag{2.1}$$

An analysis of different versions of Lucene and Xerces found that there is a large correlation between the size of the system and the number of god classes, the number of shotgun classes, and the number of brain classes. The preceding code smell techniques are indicators of code decay.

Lindvall et al. [39] compares the interactions between the modules in two versions of an Experience Management System (EMS1 and EMS2) to detect architectural degeneration. They measured architecture degeneration using coupling between modules (CBM) and coupling between module classes (CBMC). The values of CBM and CBMC were lower for

the ESM2 version than the ESM1 version, which indicates developers avoided architecture degeneration in the system.

Under the domain specific language rules subcategory, Khomh et al. [33] used the DEtection and CORrection (DECOR) technique to detect code smells (god class). This technique generates automatic detection algorithms to detect code smells or antipatterns using rule cards. Rule cards are designed in a domain specific language with the combination of metrics and threshold values. The threshold values are defined based on in-depth domain analysis and empirical studies. The authors analyzed the relation between code smells and the changes in 9 releases of Azureus and 13 releases of Eclipse and concluded that code smells do have higher change-proneness. Ciupke [8] proposed automatic detection of design problems by specifying queries to the information gathered from the source code. The result of the query is the location of the problem in the system. These design queries can be implemented using logical propositions. The heuristics used to build these queries are based on the experience of the author. The author presented design violations in different versions of industrial and academic systems.

The three model-based techniques are: 1) Probabilistic model 2) Graph model and 3) Modularity model

In the probabilistic model subcategory ,Vaucher et al. [69] used a Bayesian network approach to detect the presence of god classes. They built a Bayesian network model of the design detection rules. This model is based on metrics used to characterize specific classes and compute the probability that these specific classes are god classes. Metrics such as number of methods, number of attributes of a class and other cohesion values are used as

inputs to the model. This probabilistic model predicts all the occurrences of god classes with a few false positives in different versions of Xerces and EclipseJDT.

In the graph model subcategory, Sarkar et al. [62] detected back-call, skip-call and dependency cycle violations in a layered architecture using a module dependency graph. The metrics used to detect these violations are: back-call violation index, skip-call violation index, and dependency violation index. In a dependency graph, back-call violations can be detected if the modules of one layer call the modules in another layer except the top layer. Skip-call violations can be detected by identifying the modules of one layer that call the modules existing in other layers but not the modules in adjacent layers. Dependency cycle violations are detected by the identifying strongly connected components in the module dependency graph. In a strongly connected graph, there exists a path from each vertex to every other vertex in a graph. The authors analyzed MySql 4.1.12 and DSpace and identified these violations using module dependency graphs. Johansson and Host [30] identified an increase in violations of design rules using graph measures of the architecture of software product lines. Increase in the design rule violations from one version to other version of the software is a good indicator of code decay.

In the modularity model subcategory, Wong et al. [71] detected software modularity violations using their CLIO tool. This tool computes the differences between predicted co-change patterns and actual co-change patterns to reveal modularity violations (co-change patterns reflect classes that are often changed simultaneously). They analyzed 10 releases of Eclipse JDT and 15 releases of Hadoop and identified four types of modularity violations

that contribute to code decay. They are: cyclic dependency, code clone, poor inheritance hierarchy, and unnamed coupling.

2.1.2.3 Metrics

This section presents the metrics used to detect code decay and its forms.

> **Research Question 2:** What metrics are used to evaluate code decay? (i.e., How is it measured?)

We found that certain metrics are used to evaluate code decay. The results are summarized in Table 2.5. Eick et al. [18] defined code decay indices: history of frequent changes, span of changes, size, age, and fault potential to analyze historical change management data. An increase in values for history of frequent changes for a class and span of changes for modification record are indicators of code decay. Ohlsson et al. [50] found empirical evidence of code decay using average number of changes in a module, and 'coupling' (how often a module is involved in defects that required corrections extended to other modules). Increase in the value of coupling and average number of changes in a module is a good indicator of code decay. Lindvall et al. [39, 67] uses coupling between modules (CBM) and couple between module classes (CBMC) to avoid architecture degeneration by identifying violations in the mediator pattern. The increase in value of CBM and CBMC from one version of the system to another indicates degeneration in architecture. Li and Long [38] used various metrics related to defects spanning multiple components in a system. The greater the values of these metrics, the more significant is the architecture degeneration.

Hassaine et al. [24] used metrics such as the number of classes and the number of triplets to identify architecture decay by analyzing the architecture history using archi-

Table 2.5

Metrics

Category/Metrics	Relationship
Code decay	
History of frequent changes: Number of changes to a module over time [18].	Increase in number of changes to a module is an indicator of code decay.
Span of changes: Number of files touched by a change [18].	Increase in span of changes is an indicator of code decay.
Coupling: How often a module involved in defects that required corrections extended to other modules [50].	Increase in coupling between modules is an indicator of code decay.
Size: Number of non-commented source code lines from all the files in a module [18] (OR) Sum of added LOC, deleted LOC, added executable LOC and deleted executable LOC [50].	Growth in size of the system over time alone does not tell about the code decay. It represents the complexity of the system.
Fault potential: Number of faults that will have to be fixed in a module over time [18].	Number of faults need to fixed itself does not reveal evidence of code decay. It is the likelihood of changes to induce faults in the system.
Effort: Man hours required to implement a change [18, 50].	This depends on the total number of files touched to implement a change.
Architecture degeneration (modular level)	
Coupling-between-modules (CBM): Number of non-directional, distinct, inter-module references [39, 67]. *Coupling-between-module-classes(CBMC):* Number of non-directional, distinct, inter-module, class-to-class references [39].	Increase in the values of CBM and CBMC from one version to other version of the system indicates architectural degeneration.

Table 2.5

(continued)

Category/Metrics	Relationship
Architecture degeneration (defect perspective)	
The average quantity of strong fix relationships that a component has in a system, the percentage of multiple -component defects (MCD) in a system, the average MCD density of components in a system, the average quantity of components that an MCD spans in a system, the average quantity of code changes (fixes) required to fix an MCD in a system. MCD means defects spanning multiple components in a system. Fixing an MCD requires changes in the associated components. The relationship among these components is a fix relation. [38]	The greater the values of these metrics are, the more serious the architectural degeneration is.
Architecture decay	
Number of classes: Growth in size of the application [24]. *Number of triplets:* Triplet(S,R,T) S and T are two classes. R is the relation between S and T. [24].	Increase in the number of classes and number of common triplets from one version to another version by architectural diagram matching is a good indicator of architectural decay. Matching of architecture diagrams is automated using bit-vector algorithm.
Data Abstraction Coupling(DAC): Number of instantiations of other classes within a given class [58]. *Message Passing Coupling (MPC):* Number of method calls defined in methods of a class to methods in other classes [58]. *Coupling between objects (CBO):* Average number of classes used per class in a package [58].	Increase in the values of DAC, MPC and CBO from old version to latest version of the system becomes harder to maintain and indicates architecture decay.

Table 2.5

(continued)

Category/Metrics	Relationship
Design pattern decay (Modular grime)	
Strength of coupling: Determined by removing the coupling relationship between classes (can be persistent or temporary) [63].	Persistent relationship between classes is more prone to decay compared to temporary association.
Scope of coupling: Demarcates the boundary of a coupling relationship (can be internal or external) [63].	Grime originating from external classes is more prone to decay than internal classes.
Design pattern decay (Modular grime) *Direction of coupling:* Number of inbound and out-bound relationships [63].	Increase in number of in-bound classes is more difficult to remove than out-bound classes.
Design erosion	
Number of packages, Number of inner classes, Number of functions, Non-commented source code statements, New (inner) classes, New functions, Removed (inner) classes. Metrics related to packages, functions, and inner classes. [68].	Increase of these metrics between different versions of system indicates design erosion. However, not all changes are reflected in the metrics. It also depends on how design decisions accumulate and become invalid because of new requirements.
Software aging	
LOC, CountCodeDel, countLineCodeExe, CountLineComment, CountDeclFileCode, CountDeclFileHeader, CountDeclClass, CountDeclFunction, CountLineInactive, CountStmtDecl, CountStmtExe, RatioCommentToCode. Metrics related to program size (amount of lines of code, declarations, statements, and files) [13].	Program size metrics are positively correlated with software aging.
Architecture Violations	
Back-call violation index (BCVI), Skip-call violation index (SCVI), Dependency cycle violation index (DCVI). These metrics are used to detect back-call, skip-call and dependency cycle violations in layered style architecture. [62].	If BCVI/SCVI/DCVI is 1, then no violation. If BCVI/SCVI/DCVI is 0, then there is violation.

Table 2.5

(continued)

Category/Metrics	Relationship
Code smells (god class) *Access to Foreign Data (ATFD):* The number of external classes from which a given class access attributes, directly or via accessor methods. Inner classes and super classes are not counted. [46, 51, 52, 56, 66] *Weighted Method Count (WMC):* WMC is the sum of static complexity of all methods in a class. [46, 51, 52, 56, 66] *Tight Class Cohesion (TCC):* TCC is defined as the relative number of directly connected methods. [46, 51, 52, 56, 66] *Number of Attributes (NOA):* Number of attributes in a class. [56]	Increase in the number of god classes over time is an indicator of code decay. However, there are some harmless god classes also. (Ex: Class that has functionality of parser.)
Code smells (data class) *Weight of Class (WOC):* Number of attributes in a class. The number of non-accessor methods in a class divided by the total number of members of the interface[46, 56, 66] *Number of Public Attributes (NOPA):* The number of non-inherited attributes that belong to interface of a class. [46, 56, 66] *Number of Accessor Methods (NOAM):* The number of non-inherited accessor methods declared in the interface of a class [46, 56, 66]. *Weighted Method Count (WMC):* The sum of the statical complexity of all methods in a class [66]	Increase in the number of data classes over time is an indicator of code decay.

Table 2.5

(continued)

Category/Metrics	Relationship
Code smells (brain class) *WMC and TCC* are same as described under god class detection. [52] *Number of brain methods (NOM):* Number of methods identified as brain methods in class. LOC in a method, cyclomatic complexity of a method, maximum nestinng level of control structures within the method and number of accessed variables in a method.[52]	Increase in the number of brain classes over time is an indicator of code decay.
Code smells (shotgun surgery) *Changing Methods (CM):* The number of distinct methods that call a method of a class. [46, 56] *Changing Class (CC):* The number of classes in which the methods that call the measured method are defined. [46, 56]	Increase in the number of shotgun surgery smells over time is an indicator of code decay.
Code smells (Feature envy) *Access to Foreign Data (ATFD):* The number of external classes from which a given class accesses attributes, directly or via accessor methods. Inner classes and super classes are not counted. [66] *LAA:* The number of attributes from the method's definition class, divided by total number of variables accessed.[66]	Increase in the number of feature envy type code smells over time is an indicator of code decay.
Design smells (Extensive coupling and intensive coupling) *CINT:* The number of distinct operations called from the measured operation. [66] *CDISP:* The number classes in which the operations called from the measured operations are defined in, divided by CINT.[66]	Increase in coupling over time is an indicator of code decay.

Table 2.5

(continued)

Category/Metrics	Relationship
Architecture degradation	
Graph measure: It is a function that denotes the deviation of the architecture structure compared to the wanted structure defined by design rules. [30]	Increase in the number of design rule violations makes architecture degraded and an indicator of code decay.

tectural diagram matching. Riaz et al. [58] used coupling related metrics such as Data Abstraction Coupling (DAC), Message Passing Coupling (MPC), and Coupling between objects (CBO) and by comparing these values between two versions of the system. The increases in the values of these metrics indicate architecture decay of the system.

Grime is the phenomenon of accumulating unnecessary code in a design pattern. It is a form of design pattern decay. The three levels of grime are class grime, modular grime and organizational grime [28]. Schanz and Izurieta [63] use metrics of strength, scope, and direction of coupling to classify modular grime. Van Gurp and Bosch [68] assessed design erosion using metrics related to packages, functions, and inner classes. Increase in the values of these metrics between different versions of the systems indicate design erosion. Design erosion is not fully explained by the metrics. They found that design erosion is also based on the accumulation of design decisions that are not implemented due to new requirements. Sarkar et al. [62] used violation indices (BCVI, SCVI, and DCVI) to detect back-call, skip-call, and dependency cycle violations in a layered architectural style. If the value of BCVI/SCVI/DCVI indices is 1 then, there is no corresponding violation in the

architecture. If BCVI/SCVI/DCVI value is zero, then there is corresponding violation in the architecture.

We found empirical evidence that an increase in the number of code smells from one version to another is an indicator of code decay. The code smells were identified using using well-defined metrics [46, 51, 52, 56]. These metrics are listed in Table 2.5. The threshold values of the metrics is based on expert opinion and empirical analysis. An increase in the number of code smells during the evolution of software is an indicator of code decay. Cotroneo et al. [13] used metrics related to the program size (such as amount of lines of code, declarations, statements and files) to predict the relation between software aging trends and software metrics.

2.1.3 Discussion

In this mapping study we identified 30 primary studies related to our research questions. In this section we address the implications of our results and provide the limitations of our study.

The detection strategies we found in this review are categorized into human-based (manual) and metric-based (semi-automated) approaches. In manual processes, code decay is typically identified by answering questionnaires and using checklists. This approach is time consuming and non repeatable for larger systems. Moreover, it is expensive.

Metric-based approaches involve less human intervention in identifying code decay. Among the metric-based approaches, historical data analysis is useful only if the history of the system is available. In comparison techniques, the architecture of one version is used

as a baseline for comparison to subsequent architecture version. Metrics are compared to one another at the modular level. These metric values help to understand and avoid architecture degeneration. Module metrics are helpful in identifying structural violations in design patterns and architectural styles. Applying heuristics with threshold filtering rules is a prominent technique to identify code/design smells. The disadvantage of this technique is threshold values are determined by expert opinion. Using expert opinion for threshold values does not apply to all the systems uniformly in identifying code decay. A model-based approach uses Bayesian models where the probability is computed using manually validated data. In metric-based approaches there is less human intervention and they are scalable to larger systems.

From our observations, historical data analysis is a predominant technique to identify code decay. From the current state-of-the art of code decay detection techniques, we can infer that there is an opportunity for more research on automated detection techniques of code decay. Automated detection means automatic decision-making in identifying violations in architectural rules, design rules, and source code standards.

Metrics that identify module and class coupling are predominantly used in the literature to detect code decay. Our review did not identify metrics related to complexity of the system itself to detect code decay. Coupling related metrics such as Coupling between modules (CBM), Coupling between module classes (CBMC), Data Abstraction Coupling (DAC), Message Passing Coupling (MPC), Coupling between objects (CBO), number of files coupled for a change, strength of coupling, scope of coupling, and direction of cou-

pling give evidence of code decay. It is important to measure coupling when assessing code decay.

Code decay degrades the quality attributes of the system. Some of the quality attributes include maintainability (hard to change the code) [18, 51, 64, 67, 72], understandability (difficult to understand the code) [39, 51, 64], and extendability (hard to add new functionality) [67].

We also observed different factors, both developer-driven and process-driven lead to code decay. Developer-driven decay involves:

- Inexperienced/novice developers [64]
- Lack of system's architecture knowledge [59]
- Developers focused on pure functionality [64]
- Developers apprehension due to system complexity [59]
- Impure hacking (carelessness of the developers)

Process-driven decay includes difficulties related to:

- Missing functionality [59]
- Violation of object-oriented concepts (data abstraction, encapsulation, modularity and hierarchy) [64]
- Project deadline pressures [51, 64]
- Changing and adding new requirements [18, 39, 67]
- Updating new software and hardware components [18]
- Ad hoc modifications without documentation [62]

Studies that concentrated on the relation between the design/code smells and architecture degradation [30, 41, 42] provide evidence of how design/code smells affect the

architecture degradation. In aspect-oriented programming, modularity anomalies scattered among different classes are usually architecturally-relevant smells. Such architecturally-relevant smells are difficult and expensive to fix in the later stages of software development [42]. Macia et al. [41] suggested that developers should promptly identify and address the code smells up front, otherwise code anomalies increase the modularity violations and cause architecture degradation.

One of the limitations of the review is bias in selection of our primary studies. To ensure that the selection process was unbiased, we developed a research protocol based on our research questions. We selected our data sources and defined a search string to obtain the relevant literature. Since the software engineering terms are not standardized, there is a risk that the search results might omit some of the relevant studies. To reduce this risk, we did a bibliography check of every article we selected as a primary study. The key limitation in this study is that only researchers participated in the selection and analysis of the papers. We mitigate this risk by having discussions on the inconsistencies raised while conducting our study. Another potential limitation is in excluding papers that do not emphasize time or successive versions of a system when evaluating quality at a point of time (e.g., current version).

2.1.4 Conclusions

Our systematic mapping study that targeted empirical studies of detection techniques and metrics is used to find code decay. A total of 30 primary studies were selected using a well-defined review protocol. The three contributions of this study are:

- First, we categorize different terms used in the literature that leads to code decay with respect to the violations in architectural rules, design rules and source code standards.
- Second, we classify the code decay detection techniques into human-based and metric-based approaches. Subcategories of these approaches are also discussed.
- Finally, we present a comprehensive tabular overview of metrics used to identify code decay and their relationship with code decay.

Metrics identified to detect code decay help to assess the severity of code decay and to minimize it. Coupling related metrics are widely used and helpful at identifying code decay.

2.2 Architecture evaluation techniques

Architecture evaluation of software is to analyze the architecture to identify potential architectural risks and to verify that the quality requirements have been addressed in the design [14]. Existing architecture evaluation methodologies are divided into early and late evaluations. Early evaluations focus on designed architectures whereas late evaluations focus on implemented architectures after the source code of the system is available. The different architecture evaluation techniques and their goals are listed in the Table 2.6.

Researchers used architecture evaluation techniques for different goals such as risk identification and suitability analysis, sensitivity and trade-off analysis, validating a design's viability for insights, assessing software architecture for reuse and evolution, change impact analysis, predicting maintenance effort, evaluating the ability of software architecture to achieve quality attributes, predicting context relative flexibility, risk assessment, analyzing flexibility for reusing. We chose LiSCIA with a different goal to derive architecture constraints because LiSCIA pre-defines a notion of quality in terms of maintainability.

Table 2.6

Architecture evaluation techniques

Architecture evaluation technique	Goal
—Scenario based Architecture Analysis Method (SAAM)	Risk identification and suitability analysis [31]
—Architecture Tradeoff Analysis Method (ATAM)	Sensitivity and tradeoff analysis [32]
—Active Reviews for Intermediate Design(ARID)	Validating design's for viability insights [10]
—SAAM for Evolution and Reusability (SAAMER)	Assessing software architecture for reuse and evolution [40]
—Architecture-Level Modifiability Analysis (ALMA)	Change impact analysis and predicting maintenance effort [2]
—Scenario-Based Architecture Reengineering (SBAR)	Evaluate ability of SA to achieve quality attributes [3]
—SAAM for Complex Scenarios (SAAMCS)	Predicting context relative flexibility, risk assessment [37]
—Integrating SAAM in domain-Centric and Reuse-based development	Analysing flexibility for reusability [47]
—Light weight Sanity Check for Implemented Architectures (LiSCIA)	Detection of architecture erosion [5]

2.3 Architectural constraints

Software systems often refer to an architectural model and are organized into several subsystems and modules that follow some design rules. These design rules constitute the constraints on architectural styles and software design patterns. Developers may violate these constraints from one version to another. This is the starting point of architecture degeneration that causes code decay and makes maintenance difficult. Managing architectural violations for each version during software development and maintenance can prevent architectural degeneration. Below are a couple of examples that show the violations of design rules to architectural styles and design patterns.

Figure 2.4 shows a simple layered architecture that consists of layers and modules within those layers. Following are some of the constraints imposed on the layered architecture [9].

- Layer dependencies are not transitive. If layer A is allowed to use layer B and layer B is allowed to use layer C, it does not automatically follow layer A can access layer C.
- Each module within in a layer is allowed to access other modules within the layer.
- If a layer accesses another layer, all modules defined with public visibility in the accessed layer are visible within the accessing layer.

A violation can occur when a developer attempts to allow a module from Layer C to access data from Layer A, which is not defined in our rules (marked as X in Figure 2.4). This small violation represents an initial sign of architecture degeneration.

The Mediator design pattern is often used when interactions among objects are unstructured, complex, and their reuse is difficult [20]. Figure 2.5 shows an example of a Mediator pattern and potential violations (marked as X). If tasked to implement new functionality

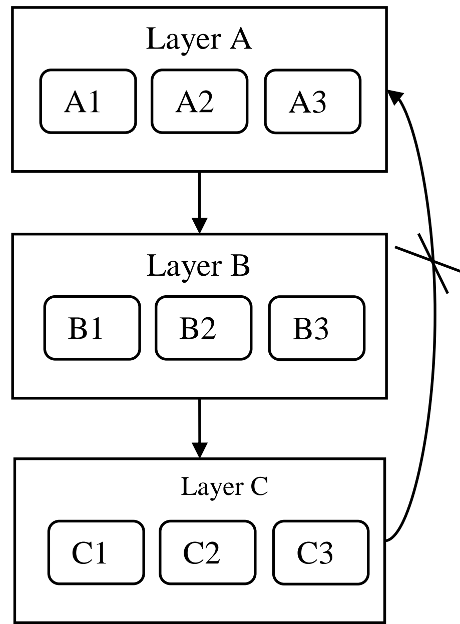

Figure 2.4

Violations in layered architecture style

for aCheckbox, a developer might complete the task using several interactions between the other colleagues (e.g., aListBox, aButton, anEntryField) but violates the rules of the Mediator design pattern. This implementation results in tight coupling between colleagues and makes it more difficult to understand the architecture of the system. A correct implementation is marked as dashed line in Figure 2.5. All these constraints can be represented using can-use/cannot-use phrases.

2.4 Architecture conformance techniques

This section presents the architecture conformance techniques and compares how our methodology is different from existing techniques. There are at least three architecture

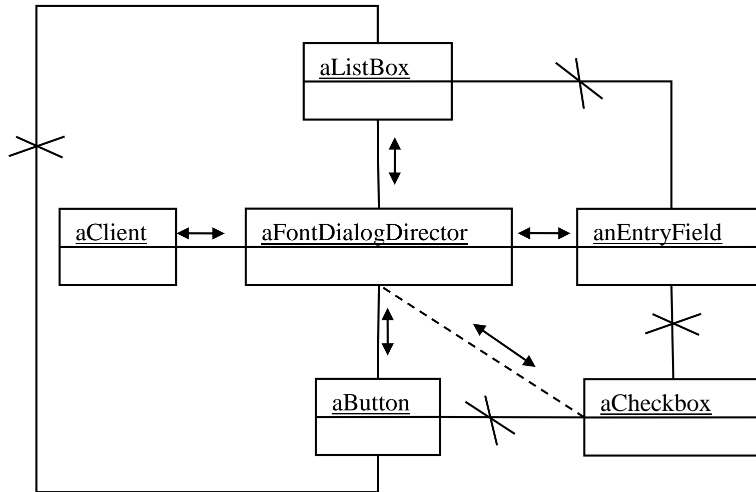

Figure 2.5

Violations in mediator design pattern

conformance techniques. They are reflexion models, using heuristics, and domain specific languages.

Murphy, Notkin, and Sullivan [48] proposed the software reflexion model, a technique to compare the high level model and source model to discover the convergence (relations in high level model is followed by the source model), divergence (relations not in high level model exist in the source model), and absence (relations in high level model does not exist in the source model) in the design. The following are the steps in the reflexion model process.

1. The high level mental model of the system is drawn purely based on the developer's experience.
2. The source model is extracted by the reverse engineering tool from the source code or by collecting information during the system's execution.
3. The developer described a mapping of files between the extracted source model and the high level mental model.

4. Using the above three inputs, the developer computes a software reflexion model that provides a comparison between the two models.
5. The developer investigates/interprets the reflexion model to derive information that helps the engineer to reason about the software engineering task.

Maffort et al. [43, 44] proposed ArchLint to discover architectural violations using heuristics. ArchLint is another method used to discover architecture violations. This method uses heuristics to discover suspicious dependencies in the source code to detect divergences and absences. Heuristics use threshold values of the dependency insertion rate, dependency scattering rate, dependency deletion rate and dependency direction rate. Basically the heuristics are implemented as SQL queries. In addition to these techniques, researchers [17, 65] used domain specific languages to focus on confirming the planned architecture and to express in a customized syntax. These constraints are defined from the planned architecture.

However, the reflexion model requires successive refinements or iterations in the high level mental model to discover the absences and divergences in the source code. The expressiveness of the reflexion models is limited to regular expressions but no other types of rules. In addition, these models focus on the conformance of the design and implementation and do not deal with different architecture styles (e.g., layered architecture). On the other hand, the drawback of the ArchLint methodology is using a large number of threshold values in heuristics. This architecture conformance process based on the proposed heuristics should follow an iterative approach — running the heuristics several times, starting with rigid thresholds. After each execution, the new warnings should be evaluated by the architect. Then, the architect may also decide to perform many iterations of the

conformance process, with more flexible thresholds. On the other hand, domain specific languages need more detailed constraints.

Our methodology uses a conceptual architecture, dependency structure matrix [61], and 'LiSCIA process' [4, 5] to derive the architecture constraints from the implemented software. Architecture violations are detected using the Lattix tool based on the derived rules. In this methodology, the expressiveness of rules is limited to can-use and cannot-use constraints rather than regular expressions. We believe that it is easy for the software engineering practitioners to write the architecture rules of their system in can-use and cannot-use phases. Rules for design patterns (e.g., mediator pattern or adapter pattern) and architecture styles (e.g., layered style or Model-View-Controller) also can be written in can-use and cannot-use phrases. We also examine the evolutionary nature of architectural violations using the number net violations, number of solved violations over different versions of software. Finally, we propose measures to indicate code decay and compare the results with coupling metrics CBM and $CBMC$.

CHAPTER 3

TOOLS

This chapter presents different tools we used for our case studies.

3.1 Lattix

Lattix[1] is a commercial reverse engineering tool used for the following purposes in our case studies.

- To extract the conceptual architecture from the given source code
- To identify dependencies at the class level using a dependency structure matrix
- To construct can-use and cannot-use architecture rules in XML
- To discover architectural violations in the source code
- To generate a 'uses' report at the class level and package level

Our case studies uses Lattix Architect version 9.0.3 and Lattix Web version 9.0.3. Researchers may use any reverse engineering tool other than Lattix, that has the above mentioned capabilities.

3.2 LiSCIA

Lightweight Sanity Check for Implemented Architectures (LiSCIA) is a structured manual evaluation method for implemented architectures [4, 5]. It focuses on the maintainability quality attribute of a system which results in discussing issues in the architecture and

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

suggests refactoring. LiSCIA uses a questionnaire to evaluate the architecture. It also constitutes a list of actions to make possible adjustments to the architecture. We use LiSCIA to evaluate the architecture of a system (extracted from source code by Lattix) and derive can-use/cannot-use rules. Bouwers et al. [4, 5] used LiSCIA to identify architecture erosion in the implemented systems and suggests possible actions and guidelines to improve the architecture of the system. In this research, we used LiSCIA for deriving architectural constraints and we are not making any modifications or corrections to the architecture. Details of LiSCIA are given in Appendix B.

CHAPTER 4

METHODOLOGY FOR PRACTITIONERS

This chapter presents our approach to assess code decay by finding software architectural violations. To apply our methodology, the practitioner needs the following tools.

- A tool to extract the architecture of the system.
- A tool to calculate the architecture dependencies.
- LiSCIA questionnaire [4, 5].
- A tool to identify architectural violations for a given set of architectural constraints represented by can-use or cannot-use phrases.

In this research, we used Lattix for extracting the architecture dependencies, and for identifying architectural violations. Given a software system that has multiple versions of interest, the following procedure presents the major steps in our methodology.

1. While there is an unanalyzed refactored version of interest
    (a) Choose an initial or refactored version that is representative of the architecture. (Section 4.1)
    (b) Choose the subsequent versions of interest and note their dates.
    (c) Derive the architectural constraints based on initial or refactored version. (Section 4.2)
    (d) While not done with the versions of interest
        i. Discover current architectural violations in a version of interest. (Section 4.3)
    (e) End while
    (f) Identify new violations and solved violations in each version. (Section 4.4)

(g) Assess code decay over multiple versions. (Section 4.5)

2. End while

Section 4.1 gives the required context for our methodology. Section 4.2 describes how to derive architectural constraints. Section 4.3 details how to discover architectural violations. Section 4.4 shows how to find the new, solved, and reoccurred violations. Section 4.5 explains how to assess code decay over multiple versions of software.

4.1 Choose an initial or refactored version

The following are the system characteristics required to apply our methodology to assess code decay.

- The system must consist of at least two versions of software.
- The source code of all the versions of interest must be available through a revision control system.
- The version history of system must include the dates of versions of interest.
- An expert of the system must be available.

4.2 Derive architectural constraints

This section presents the step by step procedure for an analyst to derive the architectural constraints for a given version. Figure 4.1 shows the methodology for deriving architectural constraints.

The roles of the participants when applying our methodology are the following.

- Evaluator, the person who reviews the system (perhaps from outside of the project team)
- Expert, a person with in-depth knowledge about the system (such as lead developer, software architect or project manager)

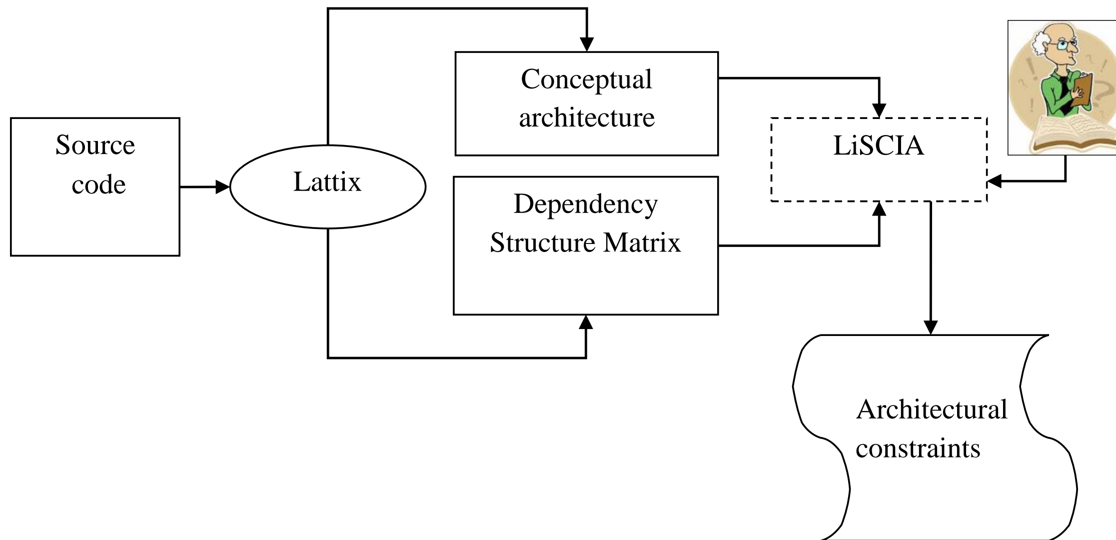

Figure 4.1

Deriving architectural constraints

- Analyst, a person who facilitates the evaluator and expert by providing required artifacts and taking notes during discussions and who analyzes the data to assess code decay.

The same person can fulfill multiple roles and multiple people can fulfill the same roles. In order to get the most out of the evaluation, at least two persons should be involved in order to create discussion. One important element of our methodology is that an expert of the system must participate in deriving architectural constraints.

In our methodology we use LiSCIA [4, 5] to derive architectural constraints. LiSCIA has two major phases. 1) Start-up phase and 2) Evaluation phase. The following are the steps to derive architectural constraints.

1. Analyst prepares the following software artifacts before the start-up phase of LiSCIA.

    - Source code of a system in an IDE (e.g., Eclipse) from the repository using the version control system.

- System's conceptual architecture and the dependency structure matrix derived from the source code using the Lattix tool.

2. Start-up phase: The following are the steps during the start-up phase of LiSCIA.

   (a) Analyst explains the roles to the participants in the study.

   (b) Analyst asks the participants to skim the LiSCIA questionnaire, which is given in Appendix B.

   (c) Analyst provides the artifacts (source code, conceptual architecture, and dependency structure matrix) to the participants.

   (d) The participants review the organization of the source code and draw the high level architecture diagram. During this step, they may use the artifacts provided by the analyst.

   (e) The participants define the components (logical groups of functionality) of the system.

   (f) The participants define the name patterns for each component defined in the above step.

   (g) The participants list the technologies used in the system.

3. Evaluation phase: Given the overview report from the start-up phase of LiSCIA, the following steps are performed by the participants to derive architectural constraints.

   (a) Answer the LiSCIA questionnaire and note the architectural constraints by evaluating the component dependencies, namely draft constraints. These are represented in can-use/cannot-use phrases.

   (b) The draft constraints are verified by the expert resulting in the final architectural constraints.

4.3 Discover current architectural violations

Given the architectural constraints of a system, the following are the steps to discover the architectural violations from the derived constraints. Figure 4.2 shows the methodology for discovering architectural violations.

1. Analyst manually inputs the architecture rules to Lattix and generates an XML file.

2. Analyst imports the XML file to Lattix.

3. Repeat the following.

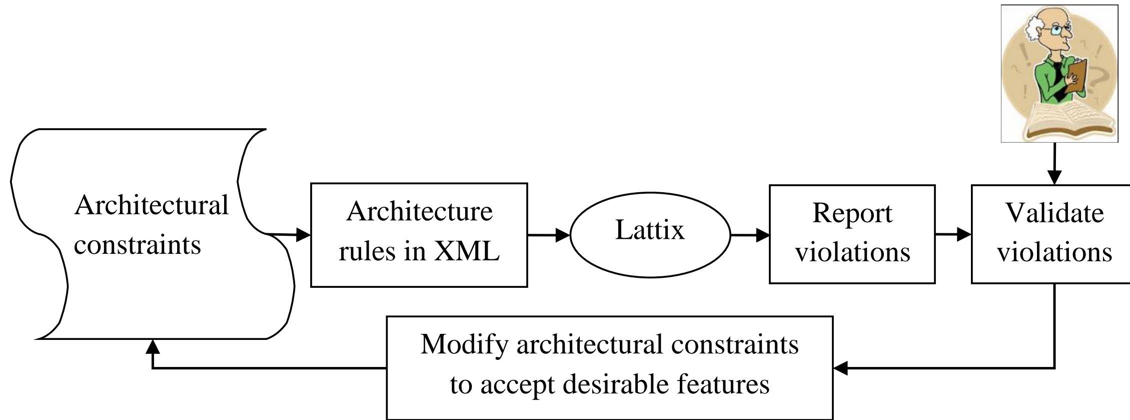

Figure 4.2

Discovering architectural violations

(a) Analyst runs Lattix to identify architectural violations on all versions of interest.

(b) The expert validates the architectural violations identified by Lattix in the above step to determine whether violations are in fact desirable features of the system, rather than violations.

(c) If any architectural constraints are not correct

  i. Modify the constraints so that desirable features are no longer flagged as violations by Lattix.

4. Until the expert identifies no more desirable features among the violations.

The discovered violations in each version, i, are called net violations ($V_{net,i}$). Net violations is defined as the number of violations discovered by Lattix in a version of interest i, in the software.

4.4 Find new, solved, and reoccurred violations

This section defines new violations, solved violations, and reoccurred violations of the architecture and list the steps how to find these violations. The following are the quantities that are calculated from the list of validated violations.

- *New violations ($V_{new,i}$):* Number of unique violations that occurred in a version of interest i, excluding the violations that occurred in the previous version, $i-1$. For the initial version of a system, the number of new violations is equal to the net violations.
- *Solved violations ($V_{solved,i}$):* Number of violations that are missing from the previous version, $i-1$. For the initial version of a system, the number of solved violations is equal to zero.
- *Reoccurred violations ($V_{reoccur,i}$):* Number of solved violations in the previous versions that reappeared in the version i. For the initial version of a system, the number of solved violations is equal to zero.

The steps to find the above defined terms are given below.

1. Analyst finds the solved violations by comparing the net violations from one version to next version of a system.
2. Analyst finds the new violations by comparing the net violations from one version to next version of a system.
3. Analyst finds the reoccurred violations by determining the life cycle of the violations.
   (a) Analyst assigns the unique violation ID for each violation.
   (b) Given the violation is detected in version n, it is easy to find the existence of the violation in later versions by checking the list of net violations.

The analyst may write parsers to analyze the net violations and find $V_{new,i}$, $V_{solved,i}$, and $V_{reoccur,i}$ with less effort.

4.5 Assess code decay over multiple versions

This section defines the terms, formulas used in code decay assessment and details the steps to assess code decay over multiple versions of a software. For a given version, the time is calculated as, the number of working days divided by 5 (because one week has five working days). The calculation[1] of work weeks excludes weekends and public holidays in the United States. The following are the terms defined for a version of interest i.

- $date_i$: Date of a version of interest i.
- $date_0$: Start date of the project or date of previous refactoring.
- *time* (t_i)*:* The development time of one version of interest i.

$$t_i = date_i - date_{i-1} \tag{4.1}$$

  where t_i is measured in work weeks.

- *Time* (T_i)*:* The cumulative development time from the beginning of the project until the version of interest i.

$$T_i = date_i - date_0 \tag{4.2}$$

  where Time (T_i) is also measured in work weeks.

- *Code decay for version* i (cd_i)*:* This is a measure of type of code decay for a given version of interest i since the last release. The value of code decay for one version of interest is calculated by Equation (4.3).

$$cd_i = \frac{V_{new,i} - V_{solved,i}}{t_i} \tag{4.3}$$

  where cd_i is measured in violations/week.

- *Net code decay* (CD_i)*:* For a version of interest i, the net code decay, which is a measure of a type of code decay, is defined as the net violations divided by the cumulative development time from the beginning of the project when coupling was zero. The net code decay value is calculated by Equation (4.4)

[1] http://www.timeanddate.com/date/duration.html?m1=06

where

$$CD_i = \frac{V_{net,i}}{T_i} \quad (4.4)$$

where CD_i is measured in violations/week.

- *Overall code decay (CD_n):* The value of overall code decay, which is a measure of a type of code decay, is calculated by considering from the initial version to the final version of interest. The value of overall code decay is calculated by considering from the initial version to the final version of interest. Suppose a system has a total of n versions of interest, the overall code decay is calculated as:

$$CD_n = \frac{\sum_{i=1}^{n} V_{new,i} - \sum_{i=1}^{n} V_{solved,i}}{T_n} = \frac{V_{net,n}}{T_n} \quad (4.5)$$

where CD_n is measured in violations/week.

The following are the steps to assess code decay over multiple versions.

1. Analyst collects the data for net violations ($V_{net,i}$) for all the versions of a system from the Lattix.
2. Analyst calculates the values of t_i and T_i for the versions of interest i using Equations (4.1), and (4.2)
3. Analyst calculates the value of code decay cd_i for version by version using Equation (4.3).
4. Analyst calculates the value net value of code decay CD_i for a version of interest i, using Equation (4.4).
5. Analyst computes the overall code decay CD_n of the system using Equation (4.5).

CHAPTER 5

CASE STUDY DESIGN

A case study is defined as an empirical method aimed at investigating a phenomenon within a specific time and space [60, 70]. A case study is usually an observation without manipulating any variables. The difference between case studies and experiments is that experiments sample over the variables that are being manipulated to discover cause effect relationships. In this research, our research questions did not explore any cause and effect relationships.

A case study design or protocol is important for its success. Case study protocol includes specific objectives, selection criteria for case and subjects, study procedure, data collection procedure, analysis procedure, and validation procedure. A pilot study is conducted on a small scale to identify the pitfalls in the methodology and evaluation of the research [35, 60, 70]. This chapter presents our case study design protocol and the lessons learned from our pilot study.

5.1 Research goals

The goal of our case studies is to collect empirical evidence to address the following research questions.

1. What is an effective method to derive architectural constraints from the source code?

2. What is an effective method to discover the extent of violations in architectural constraints?
3. What does the existence and repair of architectural violations over time imply about code decay?
4. How does our definition of code decay compare to definitions of code decay in the literature, specifically, is our definition of code decay redundant with coupling metrics?

5.2 Criteria for selecting cases and subjects

The systems we selected for our case studies are proprietary systems. The roles of the participants in our case studies are explained in Chapter 4. Our case studies are limited to proprietary systems because accessibility to the expert (architect or team lead) of the system is an important element of our methodology. Other requirements are: 1) systems with frequent short-term commits and 2) multiple versions. Compilable source code is necessary to get all the runtime dependencies in that particular version of software. The source code of the system should be available in repositories to extract the conceptual architecture and dependency structure matrix using Lattix. The versions of the target system must be released and in production.

5.3 Study procedure

The case study procedure is interleaved with the methodology procedure explained in the Chapter 4. The researcher fulfilled the role of analyst. The following procedure presents the major steps in the design of our case studies.

1. Researcher derives the architectural constraints by following the steps 1.a, 1.b, and 1.c given in Chapter 4.
2. Researcher interviews the participants for validating the procedure for deriving architectural constraints after the session. The interview guide is shown in Table 5.1.

3. Researcher identifies the architectural violations by following the steps 1.d and 1.e given in Chapter 4.
4. Researcher categorizes the architectural violations in each version. (Section 5.3.1)
5. Researcher assesses code decay by following the steps 1.f and 1.g given in Chapter 4.
6. Researcher computes the values for following metrics (Section 5.3.2)
   - Coupling-Between-Modules (CBM)
   - Coupling-Between-Module-Classes ($CBMC$)
7. Researcher calculates rate of coupling metrics (CBM and $CBMC$) for multiple versions. (Section 5.3.3)

Table 5.1

Interview guide

Demographic Information
–What is your role in the project (not your job title)? –How many years of experience do you have in software industry? –What is the size of your project team? –What kind of programming languages were used in your projects? –Have you dealt with constraints in the projects that you implemented? If so, which programming language did you use?
Session questions
–Do you think that you missed evaluating something from the architecture? If so what? –What is the level of your confidence in deriving the architectural constraints? 1. Very Low 2. Low 3. Medium 4. High and 5. Very High
Process feedback
–Do you have any recommendations to change/include in our survey? –Do you have any other suggestions in the whole process of evaluating our technique?

5.3.1 Categorize architectural violations

As stated in Chapter 4, an architectural constraint is represented as a can-use and cannot-use rules. A violation of a constraint is when a module uses a module that should not. The architectural violations we discovered fall into one of the following five types of uses relationships.

- Class Reference — Reference to a class name
- Method call — The three subcategories in the method call are:
  - Virtual (regular Java method)
  - Static
  - Interface (calling a method on a Java interface)
- Inherits — The two subcategories in the inheritance category are:
  - Inherits
  - Implements
- Data Member Reference: Reference to a field in class or interface
- Constructs — The two subcategories of the constructor call category are:
  - constructor without arguments
  - constructor with arguments

5.3.2 Compute coupling metrics

This section defines the coupling metrics that we analyzed and the steps to compute those metrics. The following are the definitions of the metrics from Lindvall, Tesoriero, and Costa [39].

- *Coupling-Between-Modules (CBM):* CBM is the number of non-directional, distinct, intermodule references.
- *Coupling-Between-Module-Classes (CBMC):* $CBMC$ is the number of non-directional, distinct, intermodule class-to-class references.

To evaluate our results, we considered using the Coupling Between Objects (CBO) [7] metric which can be calculated by several tools (for example Understand tool[1]). However, Lindvall, Tesoriero, and Costa [39] explain the disadvantages of using CBO. CBO includes the coupling among the objects inside the modules which is intramodule coupling. Since intramodule coupling is an element of cohesion, we are not using CBO for our evaluation purposes. Therefore, we are interested in intermodule coupling and adopted the CBM and $CBMC$ metrics [39].

Figure 5.1 illustrates an example of an architecture of a hypothetical system. An arrow indicates a 'uses' relationship between two classes. In measuring CBM, the intramodule couplings are ignored (arrows from A1 to A2, A2 to A1, C1 to C4, C3 to C4, B1 to B3, B2 to B4, and D3 to D2). In addition, only distinct coupling between modules are considered (only one arrow is considered among D2 to B5 and B5 to D2). Therefore Figure 5.1 is reduced to Figure 5.2. To calculate CBM, the directions of the arrows are ignored. The value of CBM for the modules A, B, C, and D in Figure 5.1 is 2, 3, 3 and 2 respectively.

When calculating the $CBMC$ of modules, the number of intermodule class references are considered. In addition to the class references, distinct couplings are considered and directions of the arrows are ignored. The $CBMC$ values for modules A, B, C and D are 3, 4, 4, and 3 respectively as shown in Figure 5.3.

In our case studies, we excluded all the external library modules because it is expected that several modules use external libraries. We considered a package to be a module. To drill down and conduct deeper analyses we considered both CBM and $CBMC$ metrics.

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

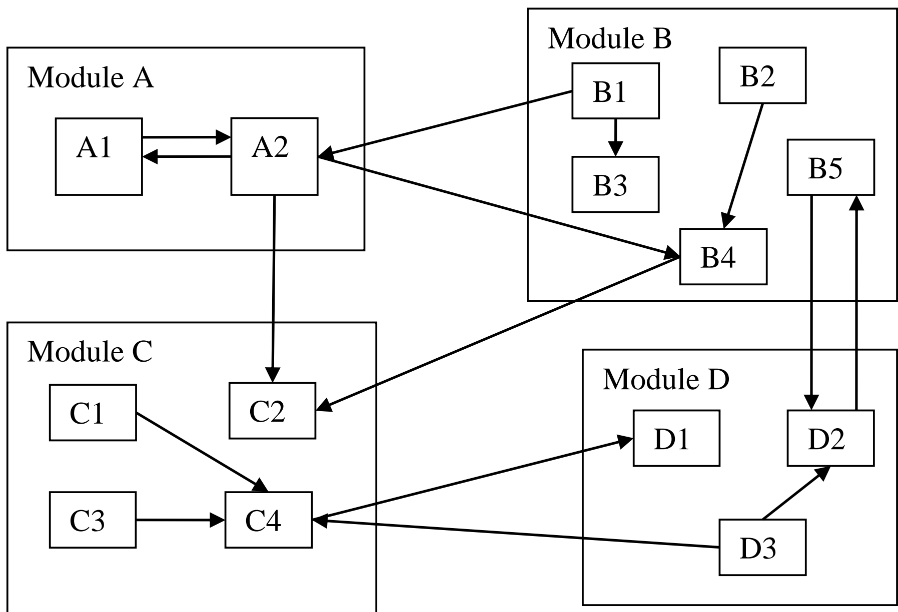

Figure 5.1

Example architectural design

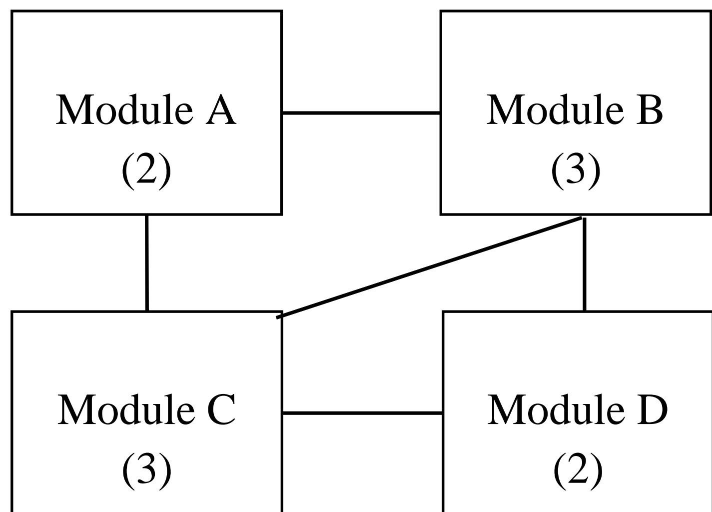

Figure 5.2

Intermodule coupling of the example system

When calculating the CBM and $CBMC$ values for nested packages, we considered all the packages to be at the same level. We also considered interfaces while calculating the $CBMC$ value. The CBM and $CBMC$ of the whole system is the sum of the values of all modules divided by 2, avoiding double counting.

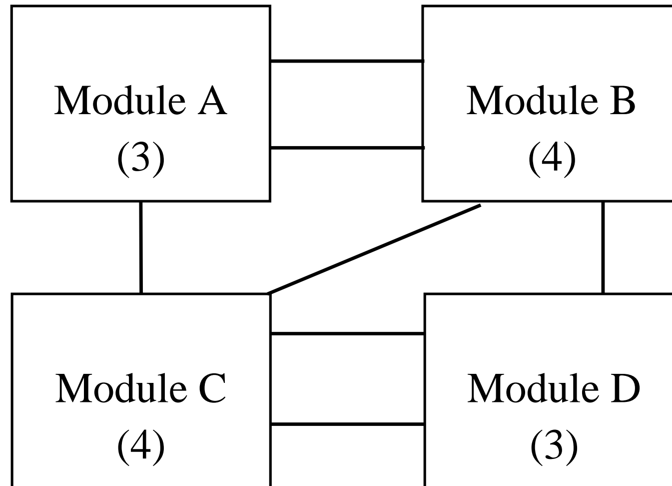

Figure 5.3

Intermodule class references for the example system

The following are the steps we performed to calculate the CBM and $CBMC$ values for larger systems.

1. Researcher runs Lattix tool and generate the 'uses' report at the class level or package level for a given version of source code. The generated report is saved as comma separated values (`*.csv`) file format. This report has a source and target columns. The source and target columns has class names.

2. Then the researcher converts the (`*.csv`) file in into `*.txt` format.

3. Researcher inputs that `*.txt` file to the two Java programs (CBMCalculator and CBMCCalculator) which computes the CBM and $CBMC$ values.

The source code of the CBMCCalculator and CBMCalculator is given in Appendix E.1 and E.2 respectively. To calculate $CBMC$, the CBMCCalculator parses the class level 'uses report' to extract source and target using comma as a delimiter. This program deletes duplicate relationships of the source and target classes, which eliminates the bidirectional 'uses' relationships. Then, the tool deletes the within module relationships if the the source and target have the same package names. The final $CBMC$ value for the whole system is calculated by adding the $CBMC$ all the modules and dividing by 2. In a similar way, CBMCalculator calculates CBM values are calculated using a 'uses' report for the package level.

5.3.3 Calculate the rate of coupling metrics

This section defines the terms and formulas used in calculating the coupling metrics. The following are the terms defined for a source code version of interest i.

- ***Net CBM** ($CBM_{net,i}$):* The value of CBM at a given version of interest i.
- ***Net CBMC** ($CBMC_{net,i}$):* The value of $CBMC$ at a given version of interest i.
- ***Change in CBM** (ΔCBM_i) :* The difference of CBM values between the version of interest i and its previous version.

$$\Delta CBM_i = CBM_{net,i} - CBM_{net,i-1} \tag{5.1}$$

- ***Change in CBMC** ($\Delta CBMC_i$) :* The difference of $CBMC$ values between the version of interest i and its previous version.

$$\Delta CBMC_i = CBMC_{net,i} - CBMC_{net,i-1} \tag{5.2}$$

- ***Rate of change in CBM** ($Rate\Delta CBM_i$):* The difference of CBM values between the version of interest i and its previous version divided by time (t_i). The value of rate of change in CBM is calculated by Equation (5.3) using Equations (4.1), and (5.1).

$$Rate\Delta CBM_i = \frac{\Delta CBM_i}{t_i} \tag{5.3}$$

- ***Rate of change in CBMC** ($Rate\Delta CBMC_i$):* The difference of $CBMC$ values between the version of interest i and its previous version divided by time (t_i). The value of rate of change in $CBMC$ is calculated by Equation (5.4) using Equations (4.1), and (5.2).

$$Rate\Delta CBMC_i = \frac{\Delta CBMC_i}{t_i} \tag{5.4}$$

- ***Net rate of CBM** ($RateCBM_{net,i}$):* For a version of interest i, the net rate of CBM is defined as the net CBM divided by the cumulative development time from the begining of the project when coupling was zero. The net rate of CBM is calculated by Equations (4.2), and (5.5)

$$RateCBM_{net,i} = \frac{CBM_{net,i}}{T_i} \tag{5.5}$$

- ***Net rate of CBM** ($RateCBMC_{net,i}$):* For a version of interest i, the net rate of CBMC is defined as the net CBMC divided by the cumulative development time from the begining of the project when coupling was zero. The net rate of CBMC is calculated by Equations (4.2), and (5.6)

$$RateCBMC_{net,i} = \frac{CBMC_{net,i}}{T_i} \tag{5.6}$$

5.4 Data collection procedure

In this research, we used one of the data collection techniques called the “direct method” where the researcher is in direct contact with the participants of the case study and collects data in the real time [60, 70]. Researcher asked participants to “think aloud” and to draw the high level architecture and to define the components of the system. The researcher noted the discussions between the participants. Noting these discussions helped gather the insights into the system in the case study. After deriving the architectural constraints, the

researcher used the interview guide shown in Table 5.1 to collect the demographic information, session questions, and feedback on the whole process. Researcher in the role of analyst collected architectural violations of different versions using Lattix.

5.5 Analysis procedure

The results of the architectural constraints and architectural violations were qualitatively analyzed by interviewing the participants to extract insights into the system. The participants also validated the violations to identify any desirable features (i.e. mistakes in the constraints). The results of code decay were analyzed by using the following plots.

- Number of classes vs. version of the system
- Number of interfaces vs. version of the system
- Number of net violations ($V_{net,i}$) vs. version of the system
- Number of solved violations ($V_{solved,i}$) and new violations ($V_{new,i}$) vs. version of the system
- cd_i vs. time
- CD_i vs. time

5.6 Validation procedure

Our code decay results were validated by using coupling metrics (CBM and $CBMC$). Since code decay is violations over time, we consider rate of coupling to evaluate our results qualitatively by using the following plots.

- $CBM_{net,i}$ vs. version of the system
- $CBMC_{net,i}$ values vs. version of the system
- $Rate\Delta CBM_i$ vs. time
- $Rate\Delta CBMC_i$ vs. time

- $RateCBM_{net,i}$ vs. time
- $RateCBMC_{net,i}$ vs. time
- CBM vs. size
- $CBMC$ vs. size
- CBM vs. violations
- $CBMC$ vs. violations

5.7 Pilot study

This section explains the pilot study and the lessons learned (strengths and weaknesses). The goals of the pilot study were the following.

- To make any enhancements in the procedure for deriving architectural constraints using Lattix and LiSCIA questionnaire. Also any enhancements to the questionnaire.
- To make any improvements in selecting the target systems and human subjects.

The system under pilot study was Verifier, which is a small scale Java based proprietary tool to detect AutoCAD errors in the geometrical structure of a building or a ship. Our requirements for a system under pilot study were the following. The system must be a proprietary system to get access to the developers/team leads/architects of the system. A system with frequent commits allows study of different revisions of the system. The source code of the system should be available in the repositories to extract the conceptual architecture and dependency structure matrix using the Lattix tool. We selected six revisions of Verifier to assess code decay. The details of the system is shown in Table 5.2.

In this case study, researcher, analyst, and evaluator roles were fulfilled by the same person. An expert role was fulfilled by the team lead of the project. The expert had 25 years of experience in developing several software systems in various programming languages

including Fortran, C, Java, C#, and Python. The evaluator, who had 5 years of programing experience in Java.

Table 5.2

Pilot system details

Attribute	Description
System	Verifier
Time period	Sept. 2009–Sept. 2011
Total number of commits	60
Revisions under study	6
LOC	6k LOC

The following procedure presents the major steps in the pilot study.

1. Researcher prepares artifacts (conceptual architecture and dependency structure matrix) using Lattix.
2. Researcher and the expert derives the architectural constraints of the older version of the pilot system using LiSCIA questionnaire.
3. Interview the participants using the interview guide given in Table 5.1.

The rest of this section describes the strengths and weaknesses of the methodology based on the experience of pilot study. The scope of our methodology is:

- The architectural constraints are derived by the expert of the software system.
- The expressiveness of the constraints is limited to the can-use and cannot-use rules.
- Lattix which is used in our methodology, gives only the syntactic dependencies by using `*.class` files of the software developed in Java.

A strength of our methodology is to derive the architectural constraints at the package level and the class level. Having the architecture rules at the package and class levels is

helpful to analyze the architecture of the project. The expert and the evaluator of the project derived the constraints of the software by evaluating the questionnaire given in the LiSCIA process. The start-up phase of the LiSCIA process was helpful to the participants to recap the system organization and the way software is implemented. In the review phase, the participants evaluated the circular dependencies, expected dependencies and unexpected dependencies and listed their constraints. The time taken by the participants to derive architectural constraints using the LiSCIA process was from 1 to 1.5 hours for a small scale system. Another good aspect of the methodology was discovering the architectural violations from the derived constraints using the uses relationship between the classes from the project. The results of this pilot study (constraints report from LiSCIA and the interview session with the expert) provided evidence that deriving architectural constraints and discovering violations can extend to larger scale projects. We confirmed that architectural violations are an indicator of code decay rather than an architecture metric based subjective agreement which we considered earlier.

We identified the following weaknesses of our draft methodology. Validating the discovered architectural violations with the expert was missing from the pilot study. We found that this step is important to distinguish the features of the system mistakenly tagged as discovered violations. Our draft methodology did not categorize architectural violations. We concluded that categorization of architectural violations will be helpful, but is deferred to future work. In the pilot study, we analyzed the system as a whole. However, we found that analyzing the system by considering major changes in the architecture during development gives more insight into the software. We also found that compilable source code

is required in order to find architectural dependencies in the system. In order to assess code decay that is most interesting to software engineers, the target systems must be in production, whereas our pilot system was never released for production.

CHAPTER 6

CASE STUDY OF SYSTEM A

This chapter reports the System A case study. Section 6.1 gives the goals of the case study. Section 6.2 describes the case and subjects who participated in the study. Section 6.3 presents the results and analysis of our research questions.

6.1 Objective

The goal of conducting this case study was to apply our methodology to a moderate size system, collecting empirical evidence to address the research questions given in Section 1.2.

6.2 Case and subjects selection

The system under study was System A, which is a proprietary Web based system developed using the Java Spring framework by a research center at Mississippi State University. The team size had varied over the life of the project, but average size was 2 developers with 1 database administrator. The software development methodology was most similar to agile. Our selection of the participants and different versions under study was based on the selection criteria given in (Section 5.3.1) in Chapter 5. We selected nine released versions of System A to assess code decay. The details of the System A are shown in Table 6.1.

The evolution of classes and interfaces over different versions of the System A is shown in Figure 6.1 and Figure 6.2 respectively.

Table 6.1

Details of System A

Attribute	Description
System	System A
Time period	June 2011– April 2014
Versions under study	9
LOC (version 9)	55k

In this case study, the roles of evaluator, expert, and analyst were fulfilled by different individuals. An expert role was fulfilled by the team lead of the project. The expert of System A had 8 years of experience in developing several software systems in Java and JavaScript. The evaluator, who participated in deriving the architectural constraints, had 3.5 years of programing experience in Java.

6.3 Results and analysis

This section presents the results of the case study. We executed the methodology given in Chapter 4. The researcher fulfilled the role of analyst.

6.3.1 Derive architectural constraints

This section presents architectural constraints derived with the help of participants.

At the beginning of the case study, the analyst provided the Java source code of the first release version (version 1) of System A in an Eclipse IDE, a conceptual architecture

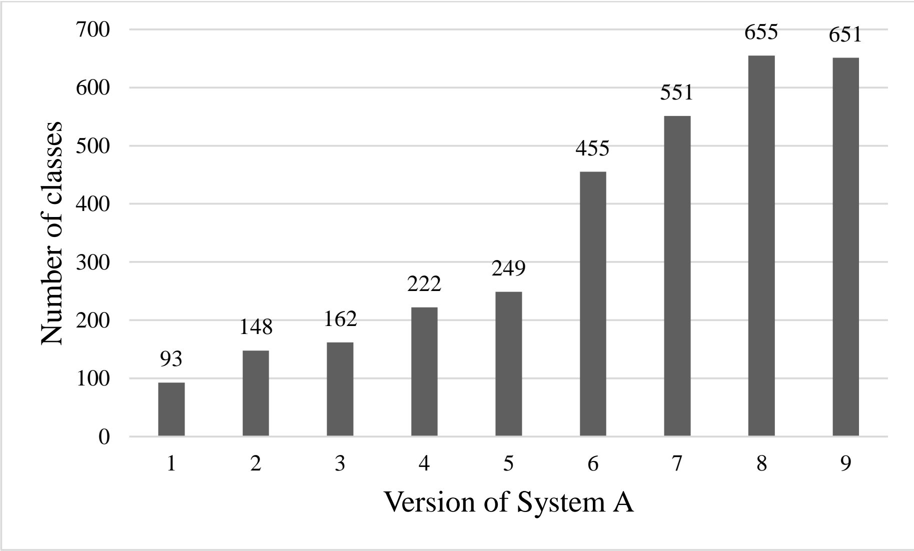

Figure 6.1

The number of classes in System A per version

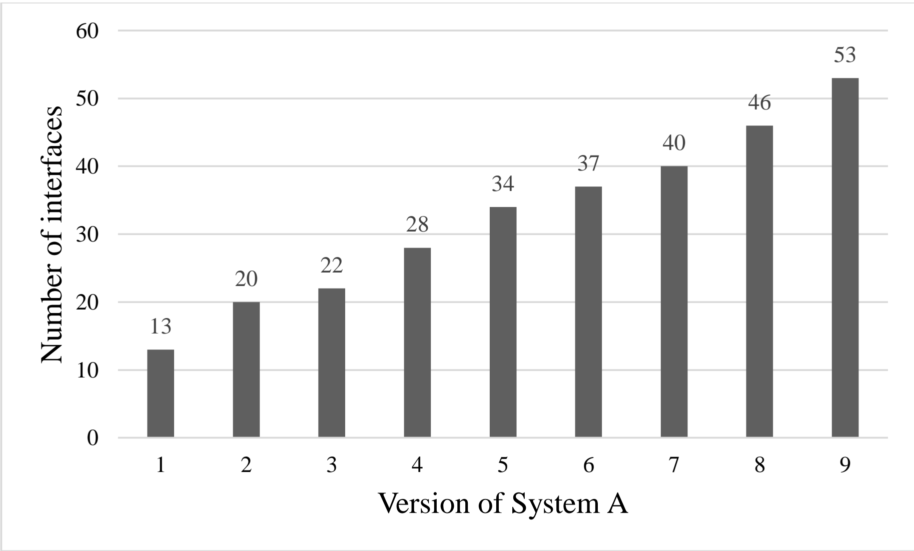

Figure 6.2

The number of interfaces in System A per version

and a dependency structure matrix to the participants. The conceptual architecture and the dependency structure matrix were derived from source code by Lattix. These artifacts are shown in Figure C.1 and Figure C.2 in Appendix C.

During the start-up phase of LiSCIA, the expert of the system drew the high level architecture of the given version of the System A discussing with the evaluator with the help of the given conceptual diagram and the source code in the Eclispe IDE. The high level architecture drawn by the participants is shown in Figure C.3 in Appendix C. Then, the participants defined the components of the system.

In the start-up phase of LiSCIA, the participants define the components of the system. The source code of the software was divided into the logical groups of functionality. In this way, different views on the architecture can be explored, which can lead to more insight and a better understanding of the implemented architecture. Participants divided the system into five components. They are the following.

- Graphical User Interfaces (GUI)
- Processing of Input
- Persistence
- Security
- Utilities

During the start-up phase of LiSCIA, the participants also define the naming-patterns. For each component, participants determined source files belong to it by defining a pattern on the directory-/file names of the source-files. In general a single file should only be matched to a single component, but in this case, the architecture was not well defined

for the project at the time of this version, the same name-pattern matches three different components. The name-patterns for the components are the following.

- Graphical User Interfaces (GUI) – `*.jsp`
- Processing of Input – `*controller`
- Persistence
  1. Model – `*trpdd.entities.*`
  2. DAO – `*JpaRepository`
  3. Lucene – `*trpdd.entities.*`
- Security – `*trpdd.security.*`
- Utilities – `*trpdd.util.*`

At the end of the start-up phase of LiSCIA, participants listed the inventory of technologies. Participants listed the technologies used within the system. They used MySql, Hibernate, Maven, Spring framework, Compass (Lucene), Jackson JSON, Apache Tiles, Liquibase, Apache PDFBox, Apache Taglibs, Joda time, C3PO, Comet, and Apache DS.

In the review phase of LiSCIA, the participants used the start-up phase information to evaluate the implemented architecture with a goal of deriving the architectural constraints. The participants mostly concentrated on the evaluation of component dependencies. To get the details of the dependencies, they used the dependency structure matrix given by the analyst. The participants discussed circular dependencies, unexpected dependencies, and which component depends on most of the other components. The participants listed the following software architectural constraints represented by can-use or cannot-use phrases.

- `Controllers can use persistence`
- `Persistence cannot use controllers`

- `Controllers can use entities`
- `Entities cannot use controllers`
- `Lucene cannot use entities`
- `Entities cannot use lucene`
- `Utilities can be used statically anywhere`
- `Security cannot use persistence`
- `Utilities can use entities`
- `GenericJpaRepository cannot use entities`

There are three major changes in the architecture due to adding new functionality or changing features due to change in requirements in three different versions (3, 7, and 8) of the system.

Major architecture enhancements:

- Added validation package (version 3)
- Added jobs package (version 7)
- Added exception package (version 8)

The participants evaluated the architecture of those versions and listed the following architecture rules. There were no constraints removed by later versions, but new constraints were added.

For System A version 3

- `Controllers can use security`
- `Security cannot use controllers`
- `Controllers can use validation`
- `Validation cannot use controllers`

- `Validation cannot use persistence`

For System A version 7

- `Jobs can use entities`
- `Jobs can use persistence`
- `Entities cannot use jobs`
- `Persistence cannot use jobs`

For System A version 8

- `Controllers can use exceptions`
- `Persistence can use exceptions`
- `Exceptions cannot use controllers`
- `Exceptions cannot use persistence`

The above software architectural constraints were derived by the expert and evaluator. These constraints were validated by asking session questions of the participants. The questions are listed in Table 5.1 in Chapter 5. The participants were very confident when deriving the architectural constraints. The results of the interview session is discussed in Chapter 8. The time taken to derive the constraints was 1.5 to 2.0 hours including the interview sessions. Once again the constraints were reviewed by the expert to finalize them before discovering violations.

6.3.2 Discover architectural violations

Using the derived architectural constraints, an XML file was generated by Lattix by manually entering the constraints, as input to Lattix. Lattix discovered the architectural violations by comparing the the rules with the usage relationships between the modules.

The architectural violations we discovered fall into one of the following five types:

- Class Reference: Reference to class name
- Invokes: Method call — there are three subkinds: virtual (regular Java method), static and interface (calling a method on a Java interface)
- Inheritance — there are two subkinds: inherits, implements
- Data Member Reference: Reference to a field in class or interface
- Constructs: Constructor call — there are two subkinds: constructor without arguments, constructor with arguments

The net violations ($V_{net,i}$) for all the versions of System A were discovered by Lattix and the values of $V_{net,i}$ are given in Table 6.2. Figure 6.3 shows that class reference type violations are the most numerous of violation in various versions of System A. Virtual invocation is second numerous type of violation in System A. Interface invocation and data member reference types of violations are in order of highest to lowest percentage of violations.

Table 6.2

Violation count over multiple versions of System A

Version i	1	2	3	4	5	6	7	8	9
$V_{net,i}$	3	11	19	21	27	28	30	37	60
$V_{solved,i}$	0	0	0	0	0	3	0	0	10
$V_{new,i}$	3	8	8	2	6	4	2	7	33
$V_{reoccur,i}$	0	0	0	0	0	0	0	0	0

From Figure 6.1 and Figure 6.2, the growth of the system from versions 1 through 6 were due to the initial development phase being in progress. The growth in versions 7

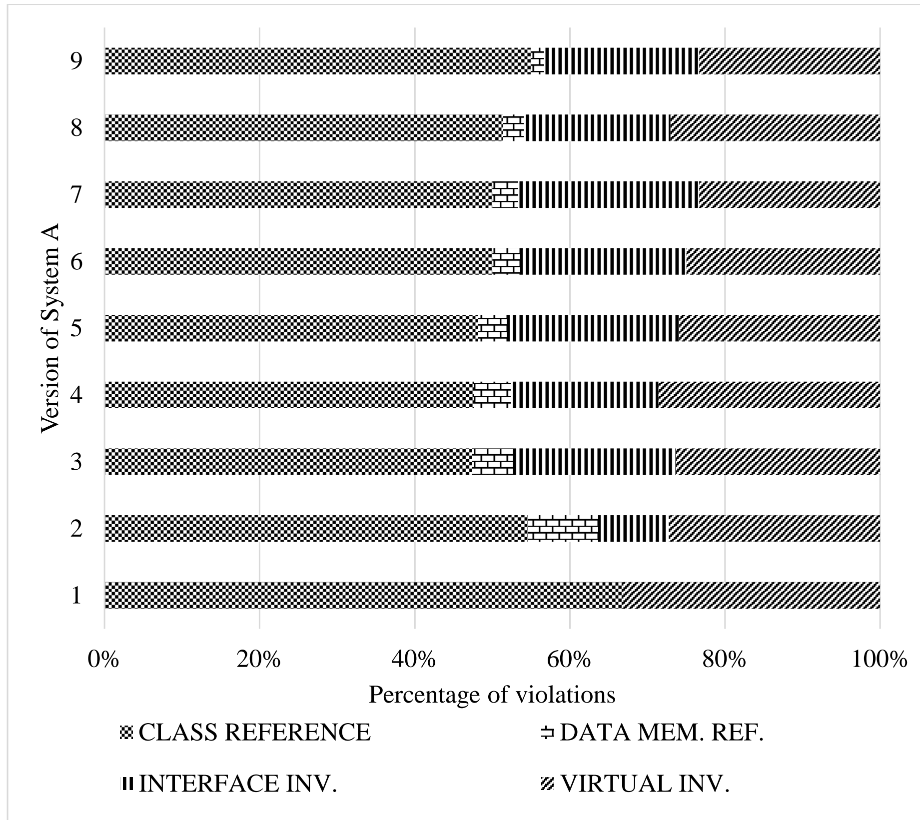

Figure 6.3

Percentage of violations in System A

through 9 were mainly due to to adding functionality due to change in requirements from the client. Figure 6.4 shows the number of net violations increase from one version to another version throughout the life of the system. The dotted lines represents that there was a major change in architecture in the system. This happened because of newly developed features or change in requirements. One such example is adding the `validation` package. The expert explained that increase in the net violations from one version to another version although they had a strong idea of the project from the beginning, but compromises are made when giving priority to achieving client's desired functionality in a given timeframe.

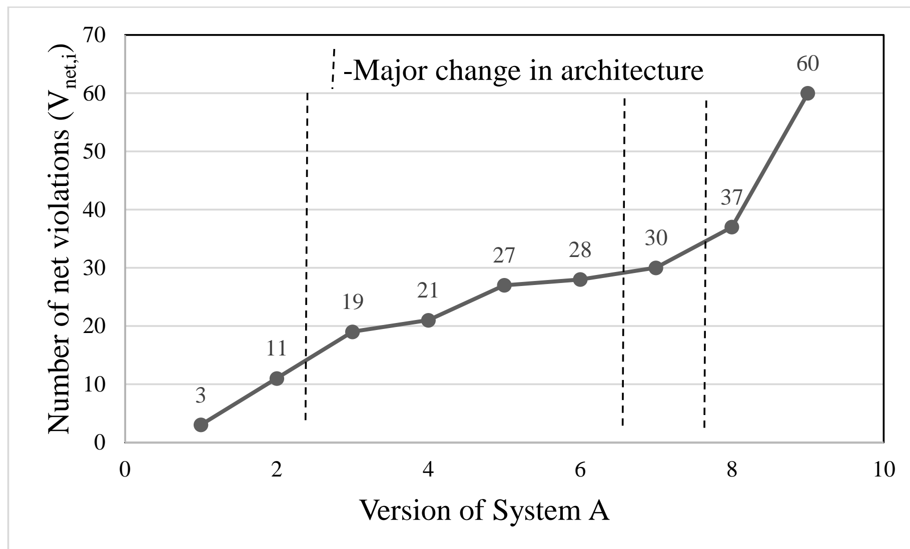

Figure 6.4

Number of net violations in System A

6.3.3 Find new, solved, and reoccurred violations

After discovering the net violations ($V_{net,i}$) in System A, we manually compared these violations from one version to next version and found the new violations ($V_{new,i}$), solved violations ($V_{solved,i}$), and reoccurred violations ($V_{reoccur,i}$). Table 6.2 shows the counts of $V_{new,i}$, $V_{solved,i}$, and $V_{reoccur,i}$ in all the versions of the System A. We assigned an unique ID for each violation to track the life cycle of that violation in different versions. In this study, once the violations were solved in a version, that violation did not reappeared in later versions. Therefore $V_{reoccur,i}$ is zero for all versions in this case study. Figure 6.5 shows the solved violations were 0 except in versions 6 and 9. In the sixth and ninth releases, developers refactored the source code by moving the functionality from one method to a different method. They also renamed and changed the signatures of a few methods. This increase in number of solved violations in version 6 and 9. An example of violation in version 8 is solved in version 9 and is given is given below.

In version 8, one of the architectural constraints is `persistence cannot use entities`. The below given is a method in `GenericJpaRepository`.

```
public List<T> executeLuceneSearch(Map<String,
       Object> requestParams, BaseEntity entity,
        int pageSize){
       // Actual code....
}
```

In version 9, the actual code is moved to a method `executeLuceneSearch(Map-<String,Object> requestParams, BaseEntity entity, int pageSi-ze, HttpServletRequest request)` and this method is called in the above method as shown in below code. This caused an increase in solved violations. Since there was no

change in the actual code and the violation remained same, it was considered as a new violation in version 9.

```
public List<T> executeLuceneSearch(Map<String,
     Object> requestParams, BaseEntity entity,
     int pageSize{
  return executeLuceneSearch(requestParams, entity,
                             pageSize, null);
}
```

Developers actually didn't fix the architectural violations, but the functionality was moved to a new method which caused new violations. This is the reason for increase in new violations in version 9 even though there was an increase in solved violations. They compromised the architecture by giving priority to the client's desired functionality in a given timeframe. Figure 6.5 shows that the new violations occurred in all the versions of System A. The change in architecture in the system didn't solve violations as the developers just concentrated only on the functionality.

6.3.4 Assess code decay over multiple versions

The released dates of the versions of System A were collected from the revision control system. For a given version i, the number of working days since the last release, time (t_i) in work weeks, and the cumulative development time from the beginning of the project for each version (T_i), also in work weeks, were calculated as explained in Section 4.5 in Chapter 4. The values of t_i and T_i for each version is shown in Table 6.3. We then computed the code decay values for each version (cd_i), the net code decay (CD_i) using Equations (4.3) and (4.4) respectively. The values of cd_i and CD_i are given in Table 6.4.

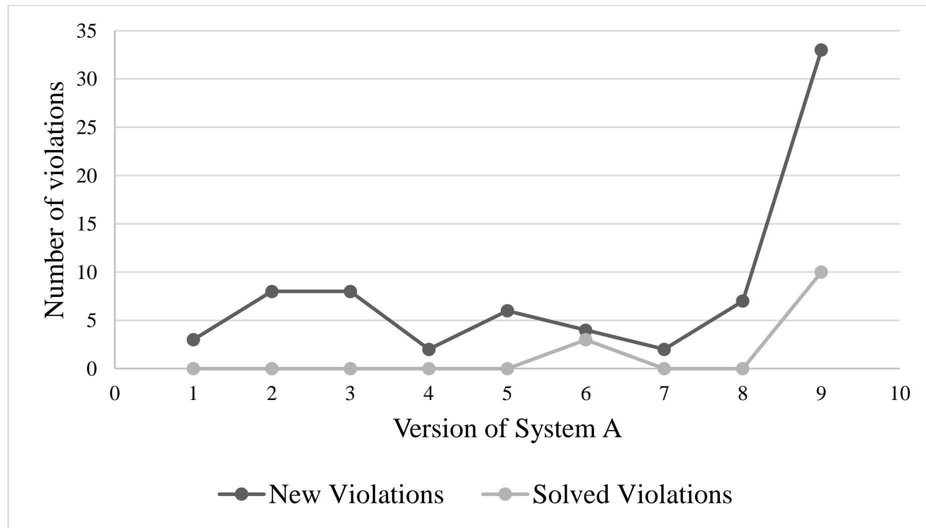

Figure 6.5

New and solved violations in System A

Table 6.3

Values of development time of System A

Version i	1	2	3	4	5	6	7	8	9
Working days	18	29	14	15	15	28	48	85	462
time t_i (weeks)	3.6	5.8	2.8	3.0	3.0	5.6	9.6	17	92.4
Time T_i (weeks)	3.6	9.4	12.2	15.2	18.2	23.8	33.4	50.4	142.8

Table 6.4

Code decay values of System A

Version i	1	2	3	4	5	6	7	8	9
cd_i	0.83	1.3	2.8	0.67	2.0	0.17	0.21	0.41	0.25
CD_i	0.83	1.17	1.55	1.38	1.48	1.17	0.89	0.73	0.42

violations per week

An ideal system follows all the architectural constraints without any architectural violations. Therefore the ideal value of code decay for such system is 0 violations/week.

Figure 6.6 show the code decay values (cd_i) for each release in System A. The cd_i measurements in this graph can give a manager insight into the the process of software development. "Did the development of this version cause further code decay?" The code decay value (cd_i) varied from 0.67 to 2.8 violations/week for the first twenty weeks of the development time of System A. When compared to an ideal system, the code decay values (cd_i) of all the versions were positive because of the increase in number of new violations compared to solved violations. The system A expert explained that the developers focused on the functionality of the software and did not concentrate on the architectural constraints. The code decay values of versions 1 through 6 were fluctuating simply due to activity in the initial development phase.

After the first twenty weeks of development time of the system, the cd_i values of versions 6–9 were mainly reduced due to changes in requirements that were received from the client. The cd_i values varies from 0.17 to 0.25 violations/week. When compared to an ideal system's code decay value of 0 violations/week, the code decay value (cd_i) of versions 7–9 versions in System A decreased compared to the earlier versions. This was because of solved violations of those versions. The changes in the architecture did not cause any decrease in the code decay values.

Figure 6.7 show the net code decay (CD_i) values of each version from the start date of the project. The CD_i measurements in the graph give a manager insight into the status of the software product from its start date. "Is the product's average code decay worse or

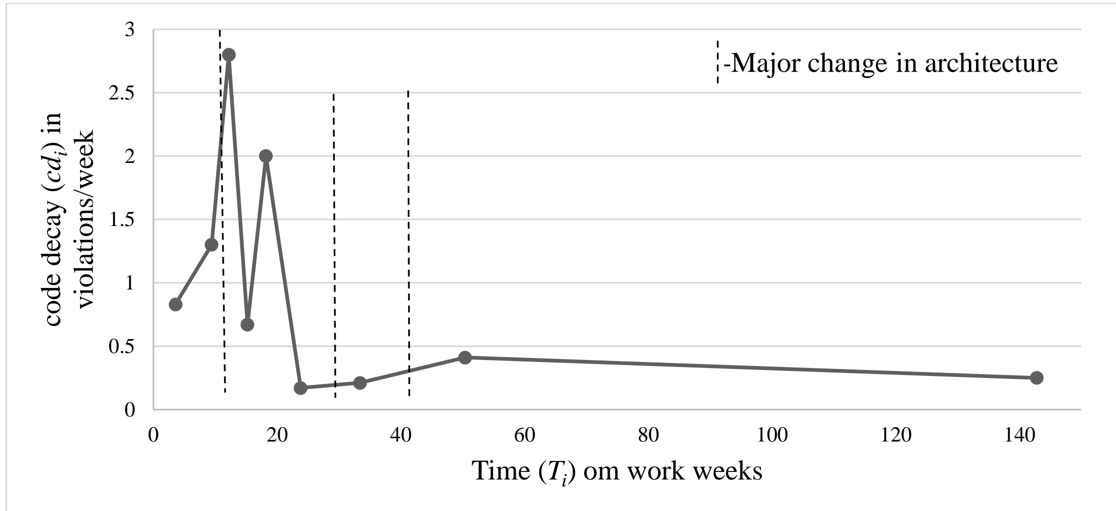

Figure 6.6

Code decay for each release in System A

better than the past version?" The net code decay values are positive because it considers only net violations over the time period since the beginning of the project T_i. The net code decay value of System A decreased at version 6 and thereafter CD_i was less than 1.0 violation/week for later versions. For an ideal system, the net code decay value is 0 violations/week.

The overall code decay (CD_n) of the System A at the end of the study period was 0.42 violations/week. The value of CD_n was calculated using Equation (4.5) and the value is given in Equation (7.1)

$$CD_n = \frac{73 - 13}{142.8} = \frac{60}{142.8} = 0.42 \text{ violations/week.} \tag{6.1}$$

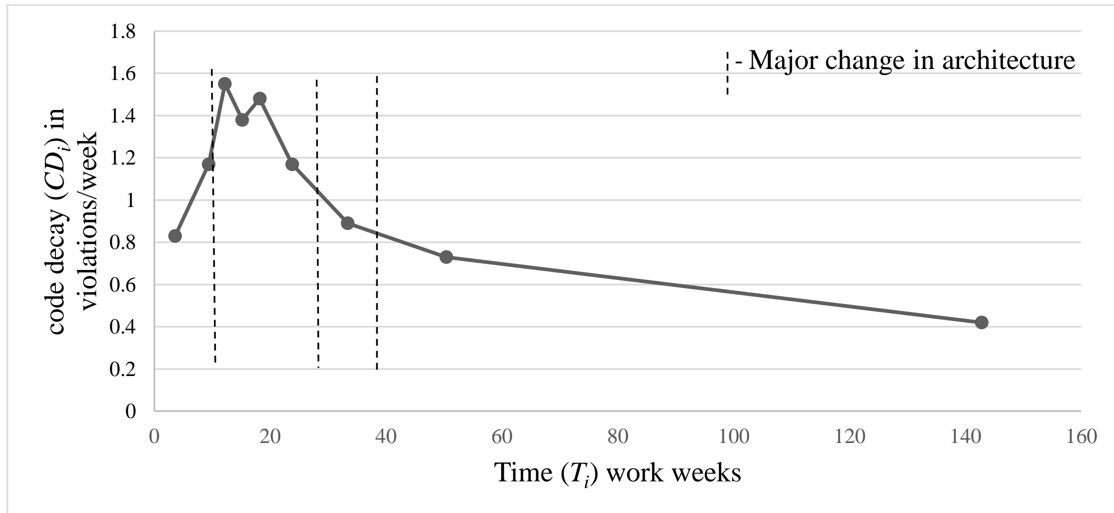

Figure 6.7

Code decay since beginning of the System A

This measure gives a value of code decay for the current system. "Does the current system have unresolved violations and what has been the average rate of violations, namely code decay?" There was on average, approximately one violation for every two weeks. To bring the value of overall code decay value to 0 violations/week, developers could refactor the source code according to the architectural constraints and remove the architectural violations. Another option is to modify the constraints to accommodate any desirable features implemented in the source code. In this case study, the expert did not modify any constraints to remove violations. The expert considered this as a technical debt [11, 12, 36, 49], planning to solve these violations in the future.

6.3.5 Comparison of results

This section presents measurements of coupling metrics ($CBMC$, CBM) to compare with code decay values. We chose coupling metrics to evaluate code decay results because Lindvall et al. claimed that increase in these metrics from one version to another version results in architecture degeneration [39]. In this case study, we excluded all the external library modules because it is expected that several modules use external libraries. We considered a package to be a module. To drill down and conduct deeper analyses we considered both CBM and $CBMC$ metrics. When calculating the CBM and $CBMC$ values for nested packages, we considered all the packages to be at the same level. Table 6.5 presents the values of coupling metrics of different versions of System A. The unit of measure of CBM and $CBMC$ is connections. These values are calculated based on the formulas given in the Section 5.3.3 in Chapter 5. The values of t_i and T_i for each version is shown in Table 6.3. We analyze our results qualitatively, because the versions of the System A are not statistically independent samples and are too few for statistical analysis. Therefore statistical analysis was not appropriate.

The graphs of net coupling between modules ($CBM_{net,i}$) and net coupling between module classes ($CBMC_{net,i}$) over the versions of System A are shown in Figure 6.8 and Figure 6.9 respectively. We observe that, the $CBM_{net,i}$ values increased from one version to the next version. $CBMC_{net,i}$ values of System A also increased from one version to the next version except in version 4. This is because developers modified a module in the system as a part of the deliverable which reduces the the value of $CBMC_{net,i}$. The major changes in the architecture did not help to decrease the coupling values as the developers

Table 6.5

Coupling values of System A

Version i	1	2	3	4	5	6	7	8	9
$CBM_{net,i}$	28	31	36	36	37	39	45	43	52
ΔCBM_i	28	3	5	0	1	2	6	-2	9
$Rate\Delta CBM_i$	7.78	0.52	1.79	0.00	0.33	0.36	0.63	-0.12	0.10
$RateCBM_{net,i}$	7.78	3.30	2.95	2.37	2.03	1.64	1.35	0.85	0.37
$CBMC_{net,i}$	159	267	291	205	413	475	540	594	910
$\Delta CBMC_i$	159	108	24	-86	208	62	65	54	316
$Rate\Delta CBMC_i$	44.17	18.62	8.57	-28.67	69.33	11.07	6.77	3.18	3.42
$RateCBMC_{net,i}$	44.17	28.40	23.85	13.49	22.69	19.96	16.17	11.79	6.37

concentrated on the functionality of the system and not the design or the architecture of the system.

The scatter plots of net coupling metrics $CBM_{net,i}$, $CBMC_{net,i}$ versus the size of the System A are in Figure 6.10 and Figure 6.11 respectively. The size of the system is the sum of the total number of classes and interfaces in a given version of the system. These plots help to compare the size and coupling of System A. The dots in these plots represent version of the System A. The linear trend line in these plots shows that the coupling was related to the size of the system. As the system size increased $CBM_{net,i}$ and $CBMC_{net,i}$ increased. This can be expected, because as the size of the system increases there is more opportunity for coupling connections.

The code decay values (cd_i) for each version is shown in Figure 6.6 above. We compare the cd_i values with the rate of change in coupling metrics ($Rate\Delta CBM_i$ and $Rate\Delta CBMC_i$) over time. Figure 6.12 and Figure 6.13 show the plots of $Rate\Delta CBM_i$ and $Rate\Delta CBMC_i$ over time. From the Figure 6.6 above, the code decay values of all the versions were

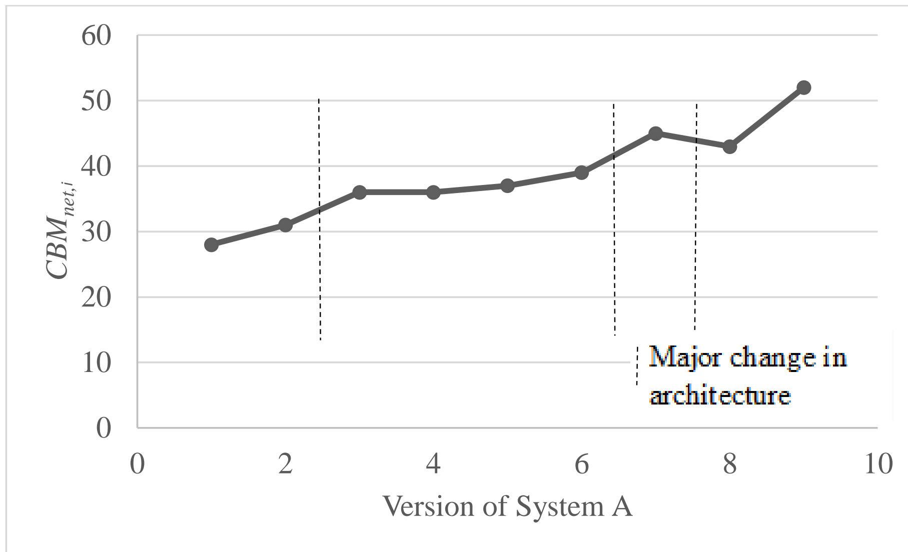

Figure 6.8

$CBM_{net,i}$ values of System A

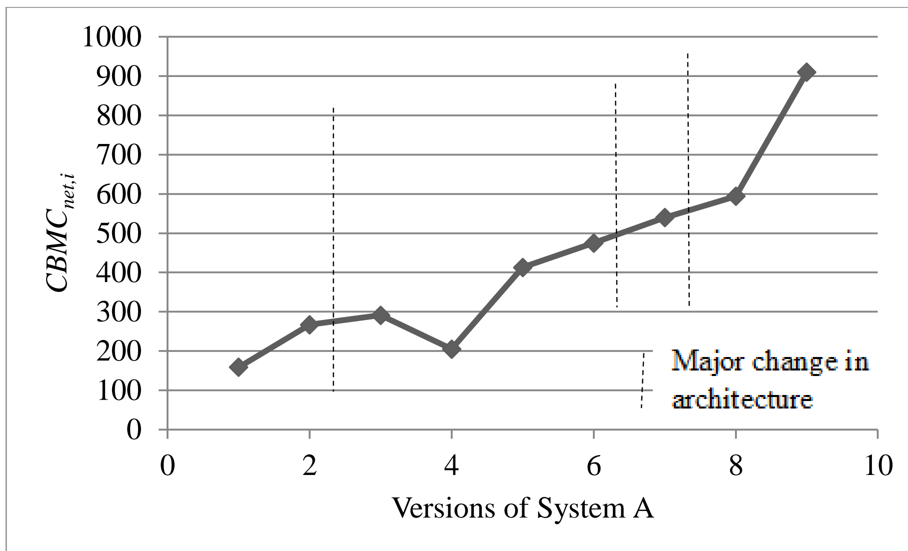

Figure 6.9

$CBMC_{net,i}$ values of System A

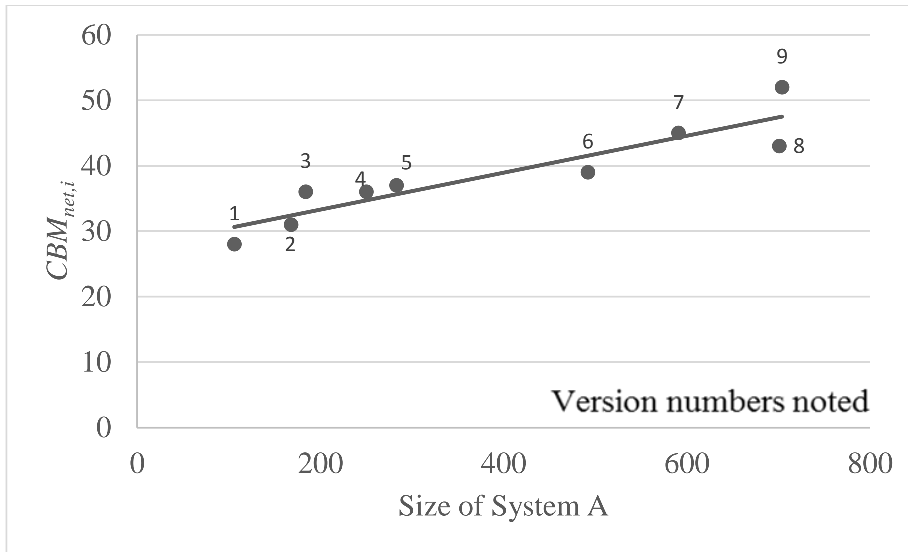

Figure 6.10

$CBM_{net,i}$ vs. size of System A

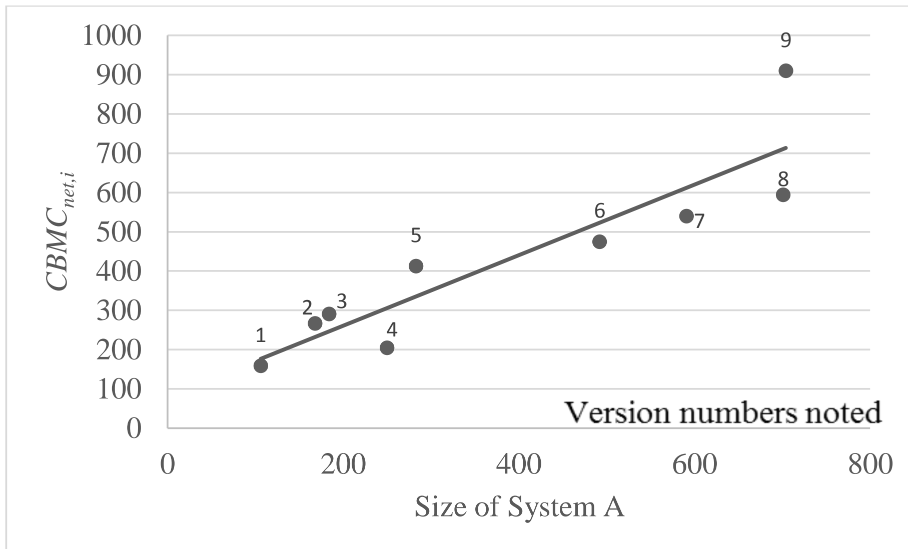

Figure 6.11

$CBMC_{net,i}$ vs. size of System A

positive before and after the changes in architecture. This was because of an increase in new violations ($V_{new,i}$) and decrease in number solved violations ($V_{solved,i}$) which means the system was decaying.

Now, let us compare these code decay values (cd_i) of each version with $Rate\Delta CBM_i$ and $Rate\Delta CBMC_i$ values. From the Figure 6.12, the rate of change in coupling between modules ($Rate\Delta CBM_i$) values for last two versions versions of System A are negative and 0 respectively. This means that coupling was less less in later versions compared to the earlier versions. Also, the cd_i value of the version 4 was reduced because developers modified a module. The values of $Rate\Delta CBM_i$ for version 4 also is less than that of earlier versions. There was an increase in cd_i values before the version 4. But there is fluctuation in the $Rate\Delta CBM_i$ between the earlier versions of the system. This was because developers concentrated on functionality during those releases. Therefore, we observe that the $Rate\Delta CBM_i$ results were not qualitatively correlated with the cd_i results.

From the Figure 6.13 we observe that there were both positive and negative values of $Rate\Delta CBMC_i$ which means the system's coupling between module classes varied between the versions. The $Rate\Delta CBMC_i$ decreases before the fifth version, where as cd_i values increases before version 5. Therefore, before version 5, the cd_i values and $Rate\Delta CBMC_i$ are not qualitatively correlated with each other. After the version 5, from the Figure 6.6 the code decay values reduces and is less than 0.5 violations/week. The $Rate\Delta CBMC_i$ values in Figure 6.13 has a gradual decrease in all the versions after version 5. Therefore, we can say that the rate of change in coupling and the code decay values

were somewhat qualitatively correlated after the version 5. Therefore for the System A, the results of $Rate\Delta CBM_i$ and $Rate\Delta CBMC_i$ are inconsistent with cd_i.

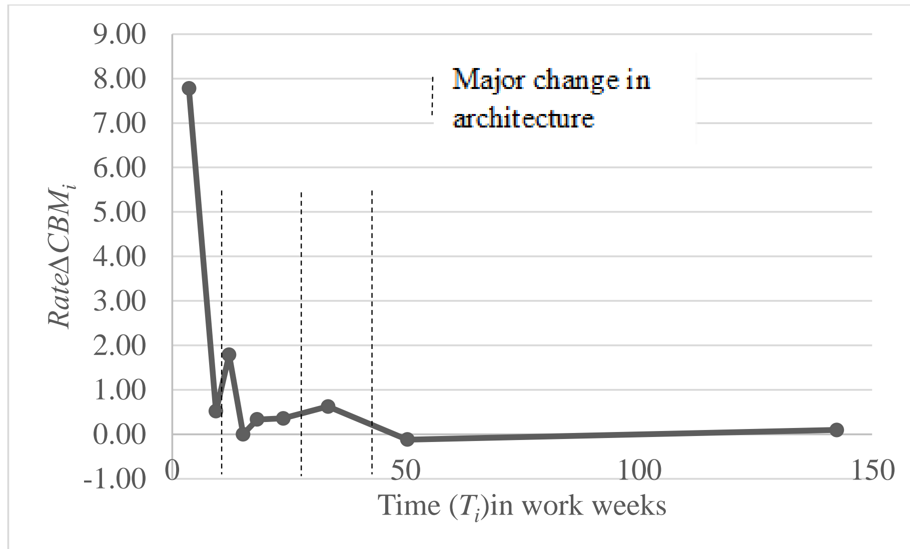

Figure 6.12

$Rate\Delta CBM_i$ for each release in System A

The net code decay (CD_i) values of all the versions of System A in Figure 6.7 were positive before and after the changes in the architecture. We observe that the CD_i values fluctuated before the fifth version and thereafter were reduced. Before the version 4, the net code decay values increases and after the version 5, the net code decay of the system was reduced.

Figure 6.14 and Figure 6.15 show the results of $RateCBM_{net,i}$ and $RateCBMC_{net,i}$ over time. Both the values $RateCBM_{net,i}$ and $RateCBMC_{net,i}$ were positive and were reduced after version 5. From these graphs, we observe that there was a decrease in the rate

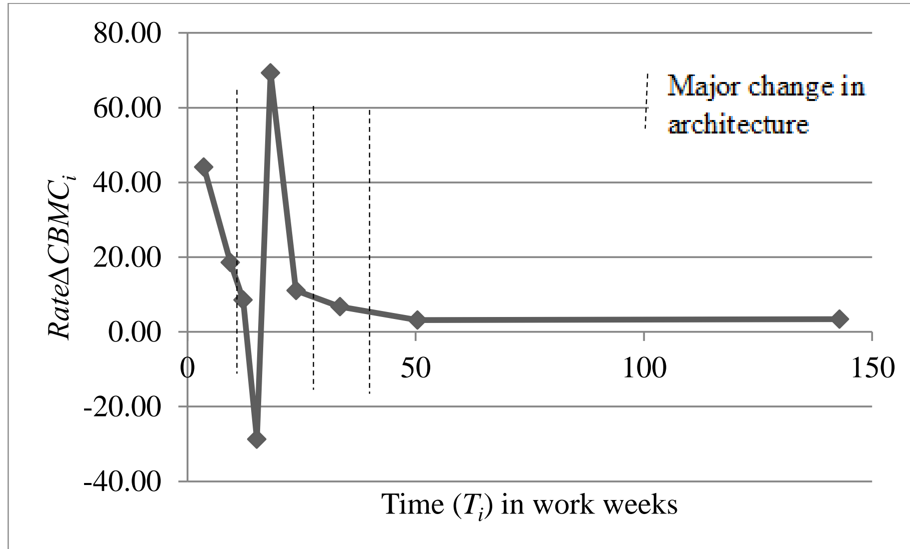

Figure 6.13

$Rate\Delta CBMC_i$ for each release in System A

of net coupling values ($RateCBM_{net,i}$ and $RateCBMC_{net,i}$) and the net code code decay values CD_i after the version 5. Before version 5, there was fluctuation in the values of net code decay (CD_i) which was not observed in the rate of net coupling values. Therefore both the results of $RateCBM_{net,i}$ and $RateCBMC_{net,i}$ were qualitatively correlated with the net code decay (CD_i) results after the version 5 and were not qualitatively correlated before version 5. The reason is developers concentrated only on the functionality of the system during these versions.

Figure 6.16 and Figure 6.17 show the scatter plots of $CBM_{net,i}$ vs. net violations ($V_{net,i}$) and $CBMC_{net,i}$ vs. $V_{net,i}$. These plots help one compare violations of the System A and its coupling. The dots in these plots represent versions of System A. In both the graphs, we observe that as coupling increased, violations also increased. This shows that

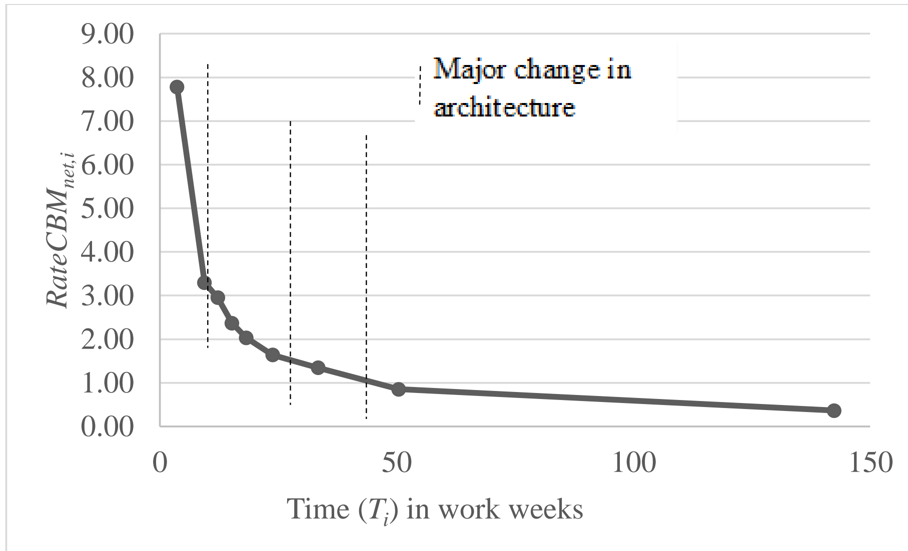

Figure 6.14

$RateCBM_{net,i}$ since the beginning of the System A

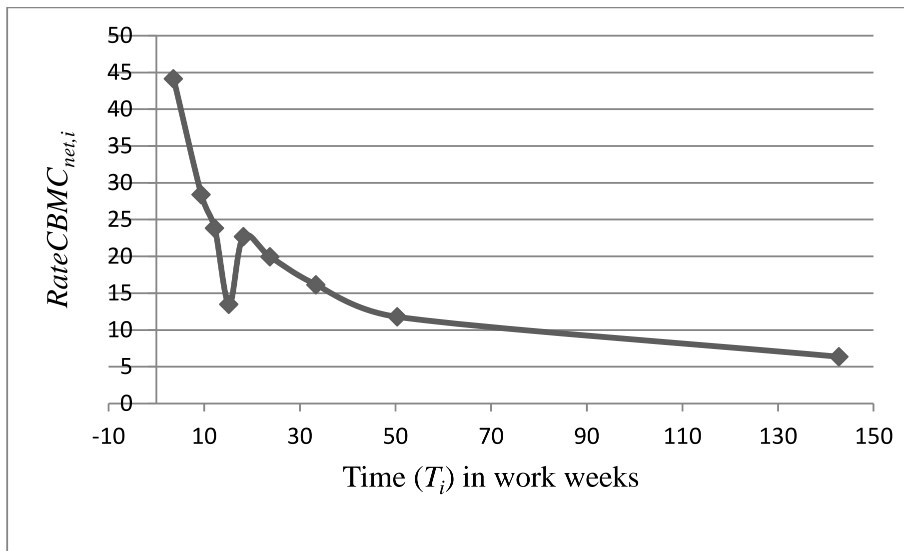

Figure 6.15

$RateCBMC_{net,i}$ since the beginning of the System A

the increase in coupling between modules and coupling between module classes resulted in an opportunity of violations. The trend line in these graphs show coupling and violations are related with each other.

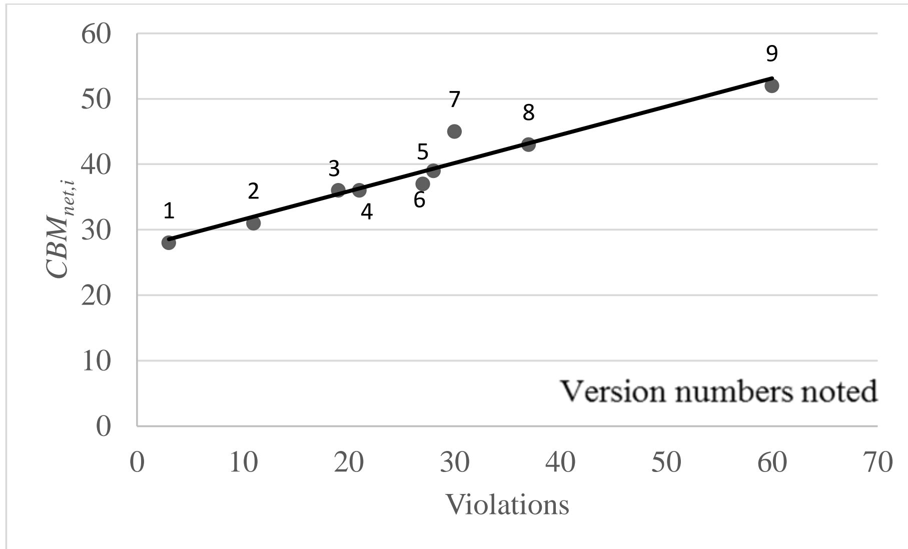

Figure 6.16

$CBM_{net,i}$ vs. net violations in System A

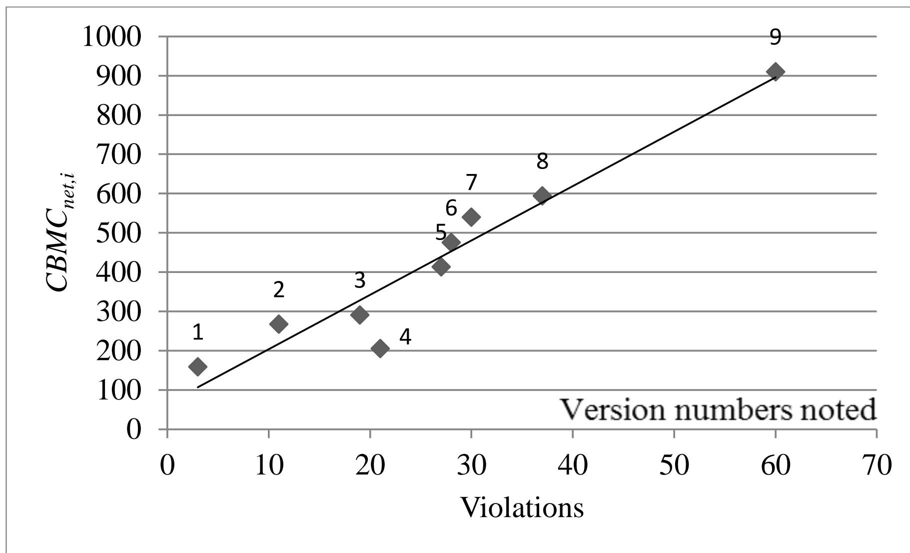

Figure 6.17

$CBMC_{net,i}$ vs. net violations in System A

CHAPTER 7

CASE STUDY OF SYSTEM B

This chapter reports the System B case study. Section 7.1 gives the goals of the case study. Section 7.2 describes the case and subjects who participated in the study. Section 7.3 presents the results and analysis of our research questions.

7.1 Objective

The goal of conducting this case study was to apply our methodology to a larger system, collecting empirical evidence to address the research questions given in Section 1.2.

7.2 Case and subjects

The selection criteria for a case study and participants is given in Chapter 5. The system under study was System B, which is a proprietary system developed by a research center at Mississippi State University. The software component of System B is a mature, online, integrated web solution providing workforce services to job seekers, employers and state employment personnel. System B is the nexus of a dynamic relationship between state workforce staff and the social scientists, data analysts, and software engineers of the research center. System B has grown to include not only reporting and management tools, but also complete web based self-service for employers and job seekers.

We selected 14 released versions of System B to assess code decay. The participants of this study were the software architect and a software developer for expert and evaluator roles respectively. Our selection of the participants and different versions under study was based on the selection criteria given in Section 5.3.1 in Chapter 5. The other details of the System B are given in Table 7.1. The evolution of classes and interfaces over different versions of the System B is shown in Figure 7.1 and Figure 7.2 respectively.

Table 7.1

Details of System B

Attribute	Description
System	System B
Time period	August 2008– May 2014
Versions under study	14
LOC (version 14)	120k

7.3 Results and analysis

This section presents the results of the case study. We executed the methodology given in Chapter 4. The researcher fulfilled the role of analyst.

7.3.1 Derive architectural constraints

This section presents the architectural constraints derived from System B by the participants. At the beginning of the case study, the analyst provided the artifacts of the first release of System B to the participants (expert and evaluator). These artifacts included Java source code in the Eclipse IDE, a conceptual architecture, and a dependency structure

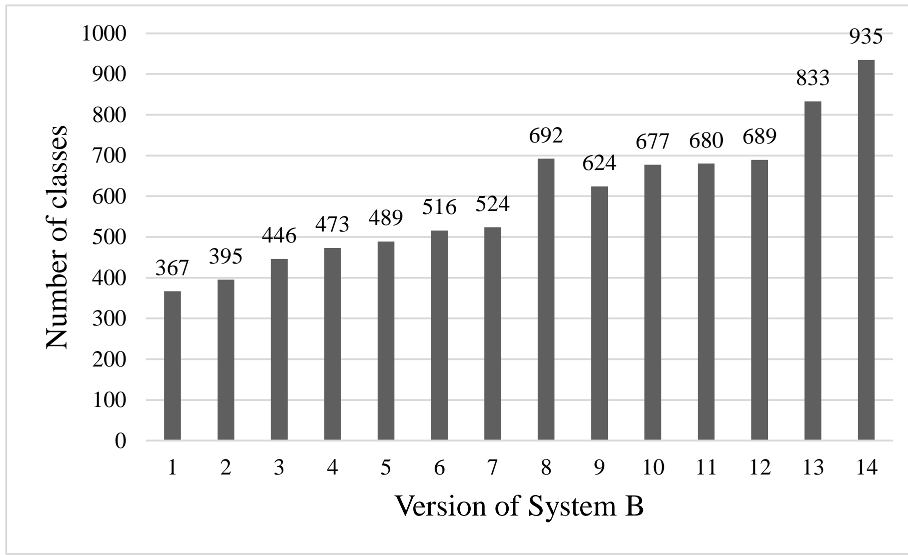

Figure 7.1

The number of classes in System B per version

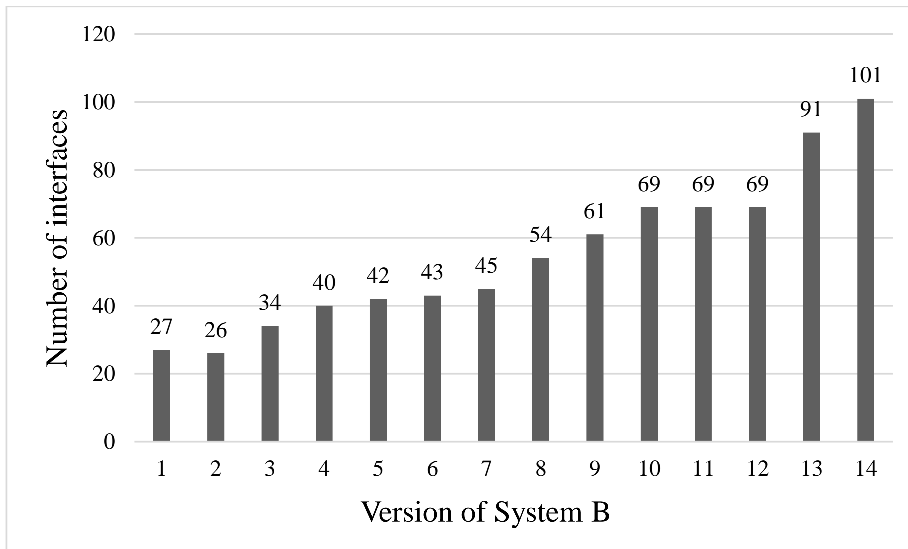

Figure 7.2

The number of interfaces in System B per version

matrix of the first release. The conceptual architecture and the dependency structure matrix were derived from source code by Lattix. These artifacts are shown in Figure D.1 and Figure D.2 in Appendix D.

During the start-up phase of LiSCIA the participants drew the high level architecture of the given version of System B. They referred to the conceptual architecture and the dependency structure matrix provided by the researcher. The high level architecture of the system is shown in Figure D.3 in Appendix D. Then, the participants defined the components of the system. The source code of the software was divided into the logical groups of functionality. The participants divided the system into six components.

- Graphical User Interfaces (GUI)
- Input processing
- Persistence
- Security
- Utilities
- Reporting

For each component, the participants determined source files belong to it by defining a pattern on the directory and file names of the source-files. In general, a single file should only be matched to a single component, but in this case, the architecture was not well defined for the project at the time of development, so the same name-pattern matches three different components. The name-patterns for the components are the following.

- Graphical User Interfaces (GUI) — `*.jsp`
- Input processing — `webapp/*/*/*-*/*.xml`, `*validator.java`, `*validator.xml`

- Persistence
  1. Model — `model/*.java`
  2. Services — `services/*.java`
  3. Lucene — `dao/hibernate/*.java`
- Security — `security/*`
- Utilities — `util.*`
- Reporting — `reporting/*.java`, `reports/*.xml`

The participants listed the technologies they used in developing the system. They used MySQL, Hibernate, Spring, Maven, Lucene, Jackson (JSON), Apache Tiles, Apache Commons, Apache HttpClient, Apache Taglibs, Liquibase, C3PO, JASPR, Castor, OpenLDAP, DWR, JQuery, Ajax, Prototype, Script.aculo.us, DBUnit, Java.x.mail, Log4J, Display tag, SLF4J, Axix, CGLib, WSDL4J, JavaAssist, and DB2.

In the evaluation phase of LiSCIA, the participants used the information from the start-up phase to evaluate the architecture with a goal of deriving the architectural constraints. The participants mostly concentrated on the evaluation of component dependencies. To get the details of the dependencies, they used the dependency structure matrix given by the analyst. The participants discussed circular dependencies, unexpected dependencies, and which component depends on most of the other components. The participants listed the following software architectural constraints represented by can-use or cannot-use phrases.

- `Services can use model`
- `Model cannot use services`
- `Utils can be used statically anywhere`
- `GUI can use taglibs`

- `Taglibs cannot use GUI`
- `taglibs cannot use services`
- `services cannot use converters`
- `framework cannot use services`
- `webservices cannot use reporting`
- `webservices cannot use taglib`
- `webservices cannot use dwr`
- `services can use dao`
- `services can use dao.hibernate`
- `dao cannot use services`
- `dao.hibernate cannot use services`
- `GUI cannot use services`
- `GUI cannot use dao`
- `GUI cannot use dao.hibernate`
- `Persistence cannot use services`
- `dao cannot use services package`
- `GenericHibernate class cannot use AuditService class`

The only major architecture change occurred before release of the eighth version. New architectural constraints were added and no constraints were removed from the earlier architecture. The new constraints are given below.

- `enum cannot use services`
- `enum cannot use dao`
- `enum cannot use listeners`

The above software architectural constraints were derived by the expert and evaluator. Then, they were interviewed by the researcher for their feedback on the session. The questions are listed in Table 5.1 in Chapter 5. The results of the interview session is discussed in Chapter 8. In this study, the roles of evaluator, expert, and analyst were fulfilled by different individuals. An expert role was fulfilled by the architect of the project. The expert of System B had 8 years of experience in developing several software systems in Java. The evaluator, who participated in deriving the architectural constraints, had 9 years of programing experience in Java. An evaluator role was fulfilled by the GUI and business rules developer. Both the expert and the evaluator were very confident when deriving the architectural constraints without missing any constraints. The time taken to derive the constraints was 2 to 2.5 hours including the interview sessions. Once again the constraints were reviewed by the expert to finalize them before discovering violations.

They suggested that a formal example of LiSCIA would be helpful to derive the constraints more quickly.

7.3.2 Discover architectural violations

Using the derived architectural constraints, an XML file was generated by Lattix by manually entering the constraints, as input to Lattix. Lattix discovered candidate architectural violations by comparing the the rules with the usage relationship between the source and target represented by can-use or cannot-use phrases. Then, the expert validated the violations. In this case, the expert didn't change any constraints to accept any desirable

features among those violations. The architectural violations we discovered fall into one of the following five types.

- Class Reference — Reference to a class name
- Method call — The three subcategories in the method call are:
  - Virtual (regular Java method)
  - Static
  - Interface (calling a method on a Java interface)
- Inheritance — The two subcategories in the inheritance category are:
  - Inherits
  - Implements
- Data Member Reference: Reference to a field in class or interface
- Constructs — The two subcategories of the constructor call category are:
  - constructor without arguments
  - constructor with arguments

The net violations ($V_{net,i}$) for all the versions of System B were discovered by Lattix and the values of $V_{net,i}$ are given in Table 7.2. Figure 7.3 shows that class reference type of violations are the most numerous of violation in various versions of System B. Method calls (virtual invocation) is second most numerous type of violation in System B. Null constructor, static, and extends types of violations are in order of highest to lowest percentage of violations.

From the Figure 7.1 above, the growth in the number of classes increased over the first seven releases. There was an increase of 168 classes from version 7 to version 8 and a decrease of 68 classes from version 8 to version 9. Figure 7.4 shows that the number of net violations also increased in the first seven versions of the system. Thereafter there was

Table 7.2

Violations count over multiple versions of System B

Version i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$V_{net,i}$	46	47	50	50	54	54	58	26	28	24	28	29	51	56
$V_{solved,i}$	0	4	3	0	0	0	0	44	0	4	0	0	1	9
$V_{new,i}$	46	5	6	0	4	0	4	12	2	0	4	1	23	14
$V_{reoccur,i}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0

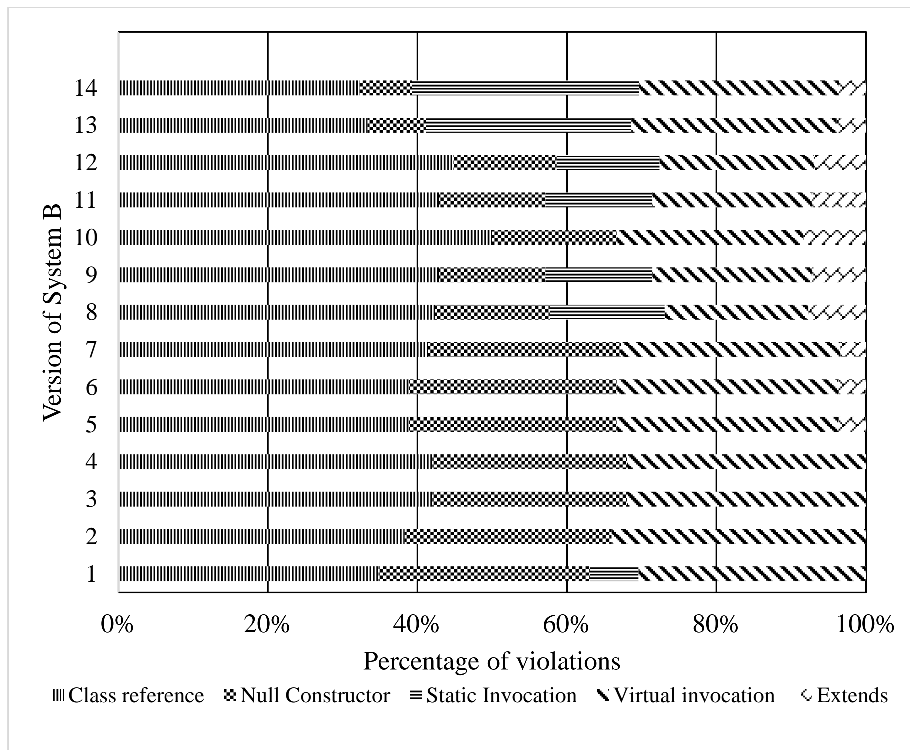

Figure 7.3

Percentage of violations in System B

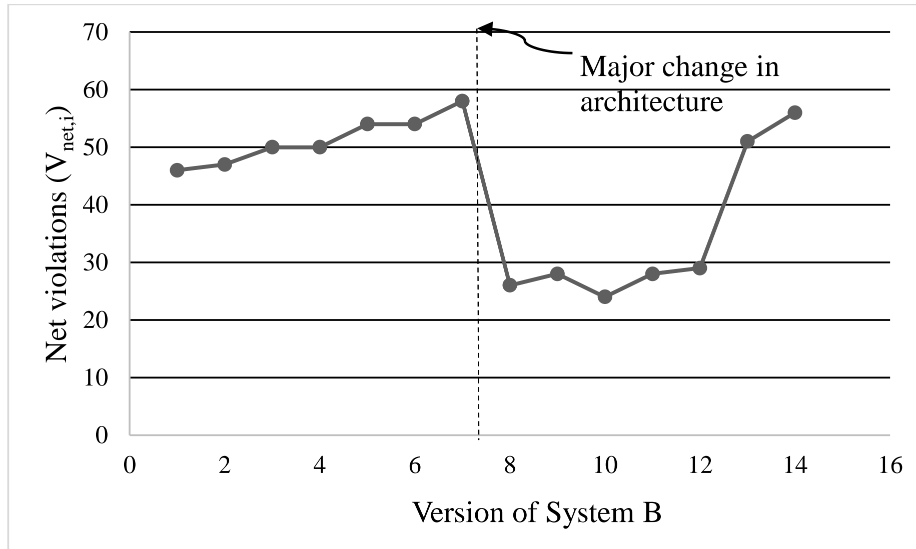

Figure 7.4

Number of net violations in System B

a sudden decrease in number of violations. There was a major architecture change before the eighth version which was created for both the application's framework enhancements and also for the initial development of the new business module called the Trade services module. These framework enhancements were mainly aimed to reduce the number of classes and the number of lines of code by consolidating classes, and re-using classes.

One such example is the Conversion classes which convert the data encapsulated in an entity to a desired format. Initially the system had a `ConversionService` class for each entity, but the developers had to keep on adding classes or additional lines of code to implement different types of conversions for the same entity. So during the enhancements between versions 7 and 8, developers refactored the source code by deleting

the `ConversionService` and opted for JSP tags to conditionally display the desired format for each entity.

In the eighth version, the outcome of the framework enhancements reduced the number of classes. But at the same time, the initial development of the Trade module brought in its own classes. Later on, a spurt in the number of classes was the outcome of this combined effort to enhance the framework and to add the functionality of the trade module.

The expert explained that the increase in the net violations from the twelfth version to the thirteenth version was due to the following.

- Mostly new developers worked on this release.
- Their code review system was not strictly enforced.
- Limited time was allocated to design and analysis of the requirements.

7.3.3 Find new, solved, and reoccurred violations

After discovering the net violations ($V_{net,i}$) in System B, we manually compared these violations from one version to next version and found the new violations ($V_{new,i}$), solved violations ($V_{solved,i}$), and reoccurred violations ($V_{reoccur,i}$). Table 7.2 shows the counts of $V_{new,i}$, $V_{solved,i}$, and $V_{reoccur,i}$ in all the versions of the System B. We assigned a unique ID for each violation to track the life cycle of that violation in different versions. In this study, once the violations were solved in a version, that violation did not reappeared in later versions. Therefore $V_{reoccur,i}$ is zero for all versions in this case study. Figure 7.5 shows the solved violations were very minimal before the eighth release. In the eighth release, developers refactored the source code by deleting those method calls that causes violations.

In the release of eighth version, due to the framework enhancements, which included changes to the architecture, there was an increase in the number of solved violations. Reorganization of several classes solved 44 violations. Figure 7.5 shows that the new violations also increased during the eighth version, due to adding a new business module to the system. In the later versions, the violations were removed in the source code because developers refactored by deleting class references and virtual invocations.

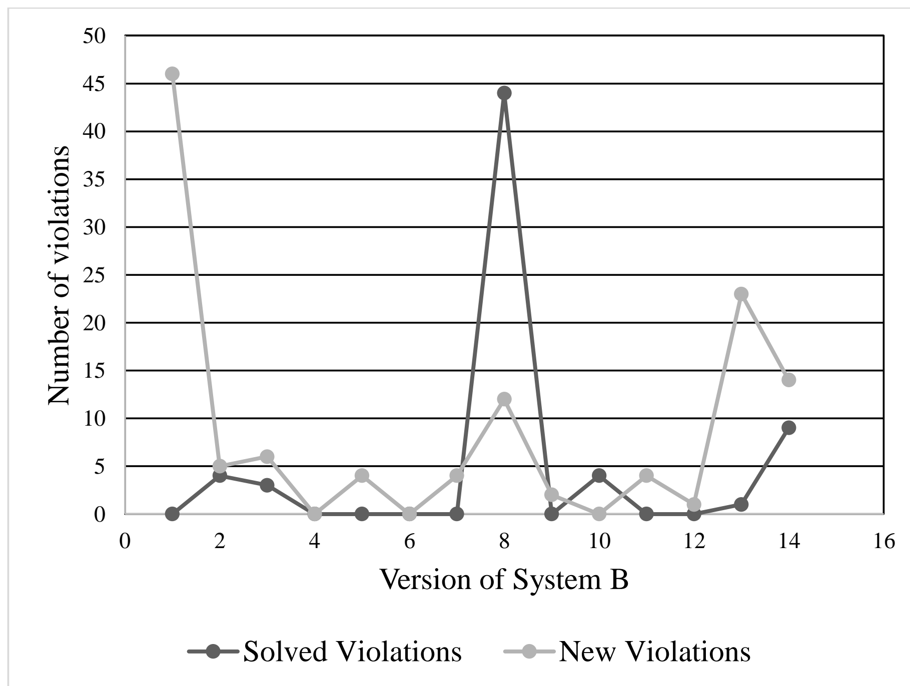

Figure 7.5

New and solved violations in System B

7.3.4 Assess code decay over multiple versions

The released dates of the versions of System B were collected from the revision control system. For a given version i, the number of working days since the last release, time (t_i) in work weeks, and the cumulative development time from the beginning of the project for each version (T_i), also in work weeks, were calculated as explained in Section 4.5 in Chapter 4. The values of t_i and T_i for each version is shown in Table 7.3. We then computed the code decay values for each version (cd_i), the net code decay (CD_i) using Equations (4.3) and (4.4) respectively. The values of cd_i and CD_i are given in Table 7.4.

An ideal system follows all the architectural constraints without any architectural violations. Therefore the value of code decay for such system is 0 violations/week.

Figure 7.6 shows the code decay values (cd_i) for rach release in System B. The cd_i measurements in this graph can give a manager insight into the the process of software development. "Did the development of this version cause further code decay?" Before major change in architecture of the system, the code decay value cd_i varied from 0 to 1.25 violations/week. When compared to an ideal system, the code decay value (cd_i) was positive because of increase in the number of new violations compared to solved violations. The System B expert explained that the developers focused on the functionality of the software and not on the architectural constraints.

After the change in architecture before release 8, the code decay values (cd_i) varies from -0.59 to 1.18 violations/week. When compared to an ideal system's code decay value of 0 violations/week, the code decay value (cd_i) of two versions in the System B after change in architecture has negative values because that there were more solved violations

Table 7.3

Values of development time of System B

Version i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Working days	412	12	12	10	16	36	40	440	36	34	17	28	240	122
t_i (weeks)	82.4	2.4	2.4	2.0	3.2	7.2	8.0	88	7.2	6.8	3.4	5.6	48	24.4
T_i (weeks)	82.4	84.8	87.2	89.2	92.4	99.6	107.6	195.6	202.8	209.6	213.0	218.6	266.6	291

Table 7.4

Code decay values of System B

Version i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
cd_i	0.56	0.42	1.25	0.00	1.25	0.00	0.50	-0.36	0.28	-0.59	1.18	0.18	0.46	0.20
CD_i	0.56	0.55	0.57	0.56	0.58	0.54	0.54	0.13	0.14	0.11	0.13	0.13	0.19	0.19

violations per week

than new violations. This is because the developers concentrated on the framework enhancements.

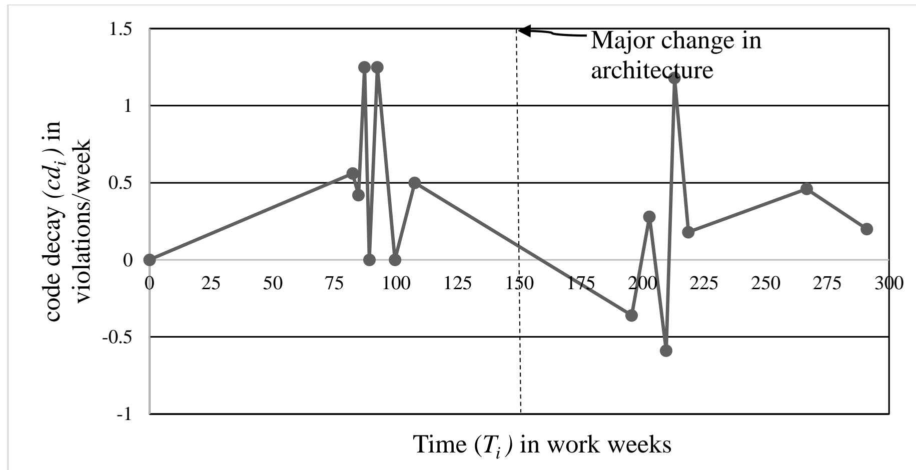

Figure 7.6

Code decay for each release in System B

Figure 7.7 shows the net code decay (CD_i) values of each version from the start date of the project. The CD_i measurements in the graph give a manager insight into the status of the software product from its start date. “Is the product’s average code decay worse better than the past version?” This graph gives the status of the software product from its start date. The net code decay values are positive because it considers only net violations over the time period since the beginning of the project T_i. The net code decay value decreased after the change in architecture and was stable for later versions. For an ideal system, the net code decay value is 0 violations/week.

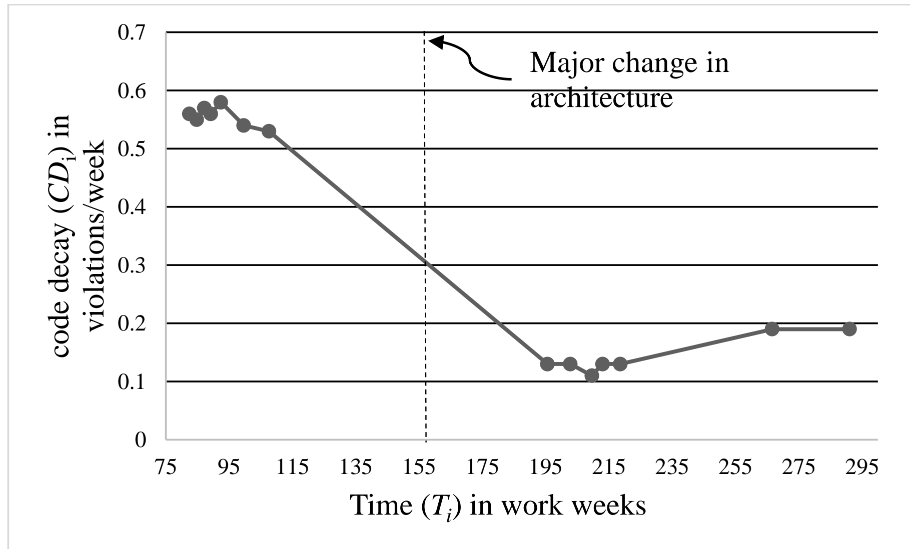

Figure 7.7

Code decay since beginning of the System B

The overall code decay (CD_n) of the System B at the end of the study period was 0.19 violations/week. The value of CD_n was calculated using Equation (4.5) and the value is given in Equation (7.1).

$$CD_n = \frac{121 - 65}{291} = \frac{56}{291} = 0.19 \text{ violations/week.} \tag{7.1}$$

This measure gives a value of code decay for the current system. "Does the current system have unresolved violations and what has been the average rate of violations, namely code decay?" There was on average, approximately one violation for every five weeks. To bring the value of overall code decay value to 0 violations/week, developers could refactor the source code according to the architectural constraints and remove the architectural violations. Another option is to modify the constraints to accommodate any desirable

features implemented in the source code. In this case study, the expert did not modify any constraints to remove violations. The expert considered this as a technical debt [11, 12, 36, 49], planning to solve these violations in the future.

7.3.5 Comparision of results

This section presents measurements of coupling metrics ($CBMC$, CBM) to compare with code decay values. We chose coupling metrics to evaluate code decay results because Lindvall et al. claimed that increase in these metrics from one version to another version results in architecture degeneration [39]. In this case study, we excluded all the external library modules because it is expected that several modules use external libraries. We considered a package to be a module. To drill down and conduct deeper analyses we considered both CBM and $CBMC$ metrics. When calculating the CBM and $CBMC$ values for nested packages, we considered all the packages to be at the same level. Table 7.5 presents the values of coupling metrics of different versions of System B. These values are calculated based on the formulas given in the Section 5.3.3 in Chapter 5. The values of t_i and T_i for each version is shown in Table 7.3. We analyze our results qualitatively, because the versions of the System B are not statistically independent samples and are too few for statistical analysis. Therefore statistical analysis is not appropriate.

The graphs of net coupling between modules ($CBM_{net,i}$) and net coupling between module classes ($CBMC_{net,i}$) over the versions of System B are shown in Figure 7.8 and Figure 7.9 respectively. We observe that, the net CBM and $CBMC$ values increased from one version to the next version. A major change in architecture happened before the eighth

Table 7.5

Coupling values of System B

Version i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$CBM_{net,i}$	57	61	65	66	67	71	71	109	98	99	99	99	106	119
ΔCBM_i	57	-4	-4	-1	-1	-4	0	-38	11	-1	0	0	-7	-13
$Rate\Delta CBM_i$	0.69	-1.67	-1.67	-0.50	-0.31	-0.56	0.00	-0.43	1.53	-0.15	0.00	0.00	-0.15	-0.53
$RateCBM_{net,i}$	0.69	0.72	0.75	0.74	0.73	0.71	0.66	0.56	0.48	0.47	0.46	0.45	0.40	0.41
$CBMC_{net,i}$	934	930	1064	1152	1196	1276	1343	2329	1944	2189	2206	2211	2778	2937
$\Delta CBMC_i$	934	-4	134	88	44	80	67	986	-385	245	17	5	567	159
$Rate\Delta CBMC_i$	11.33	-1.67	55.83	44.00	13.75	11.11	8.38	11.20	-53.47	36.03	5.00	0.89	11.81	6.52
$RateCBMC_{net,i}$	11.33	10.97	12.20	12.91	12.94	12.81	12.48	11.91	9.59	10.44	10.36	10.11	10.42	10.09

release. A new module called the 'trade' module was added to the system which increased the number of modules and classes of the System B. Thus, during the eighth release the net values of CBM and $CBMC$ also increased. The coupling metrics were stable for the next three versions because the size of the system was also stable. During the versions 13 and 14, these values again increased because of the increase in modules and classes.

The scatter plots of net coupling metrics $CBM_{net,i}$, $CBMC_{net,i}$ versus the size of the System B showed in Figure 7.10 and Figure 7.11 respectively. The size of the system is the sum of the total number of classes and interfaces in a given version of the system. These plots help to compare the relation between size of the and coupling of System B. The dots in these plots represent versions of System B. The linear trend line in these plots shows that the coupling is related to the size of the system. As the system size increased $CBM_{net,i}$ and $CBMC_{net,i}$ increased. Therefore as the size of the system increases there is more opportunity for increase in coupling.

The code decay values (cd_i) for each version is shown in Figure 7.6 above. We compare the cd_i values with the rate of change in coupling metrics ($Rate\Delta CBM_i$ and $Rate\Delta$-$CBMC_i$). Figure 7.12 and Figure 7.13 show the plots of $Rate\Delta CBM_i$ and $Rate\Delta$-$CBMC_i$ over time. From the Figure 7.6 above, the code decay values of the versions were positive before there was a change in architecture. This was because of an increase in new violations ($V_{new,i}$) which means the system was decaying. The code decay values of two versions are negative after the change in architecture because of an increase in solved violations $V_{solved,i}$ which indicates that the process in developing the system was getting better.

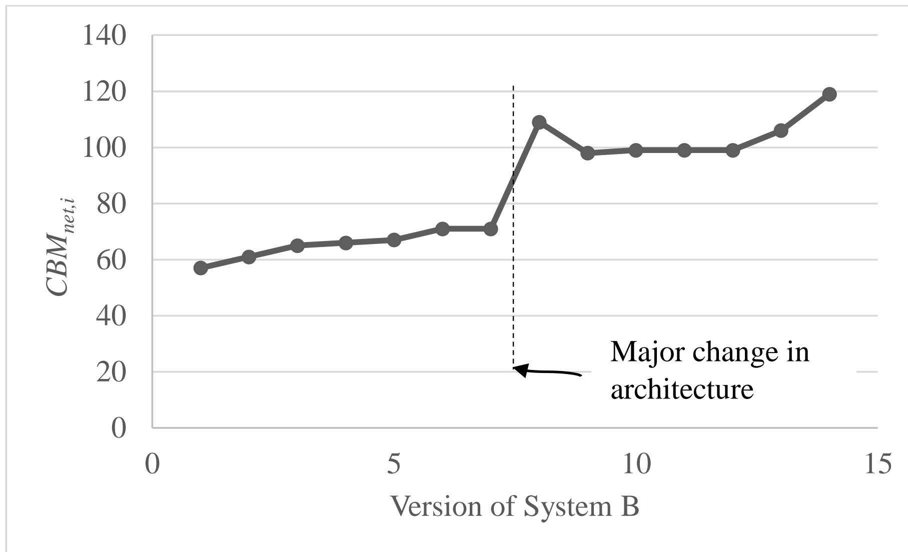

Figure 7.8

$CBM_{net,i}$ values of System B

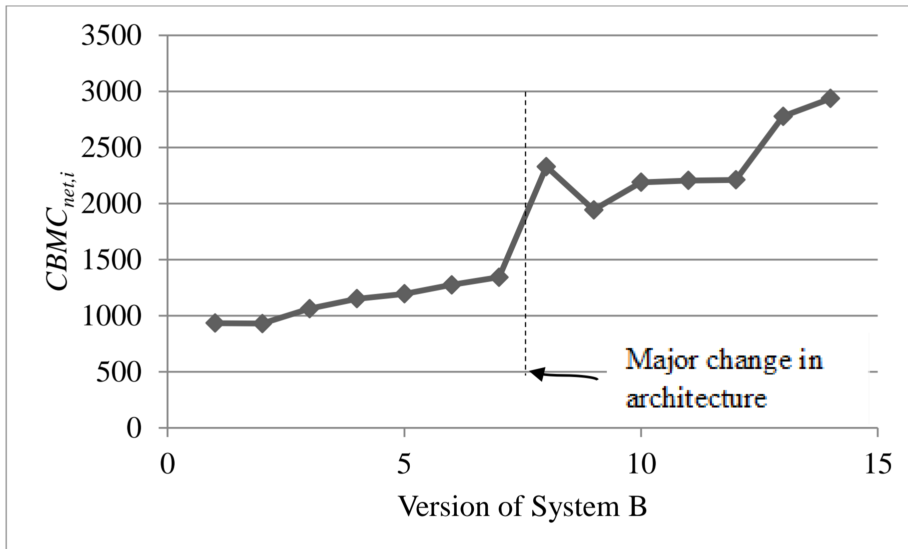

Figure 7.9

$CBMC_{net,i}$ values of System B

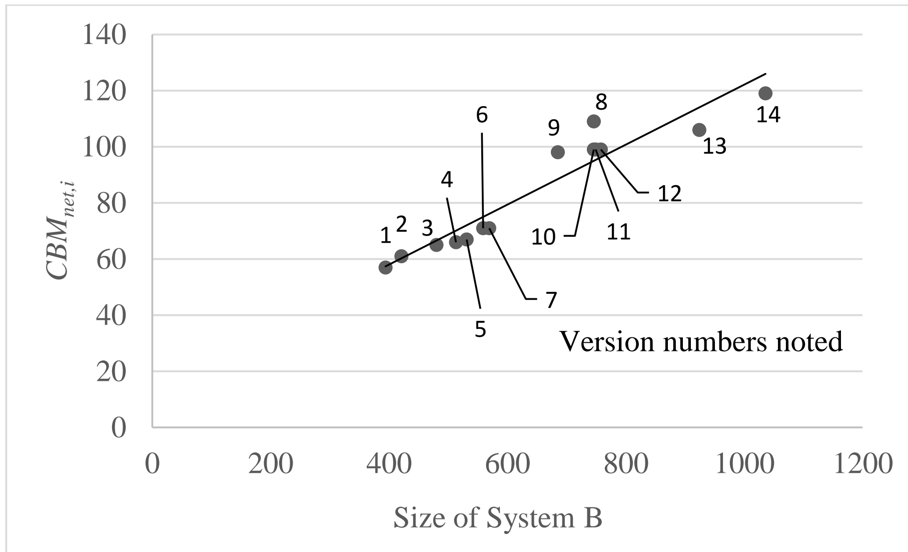

Figure 7.10

$CBM_{net,i}$ vs. size of System B

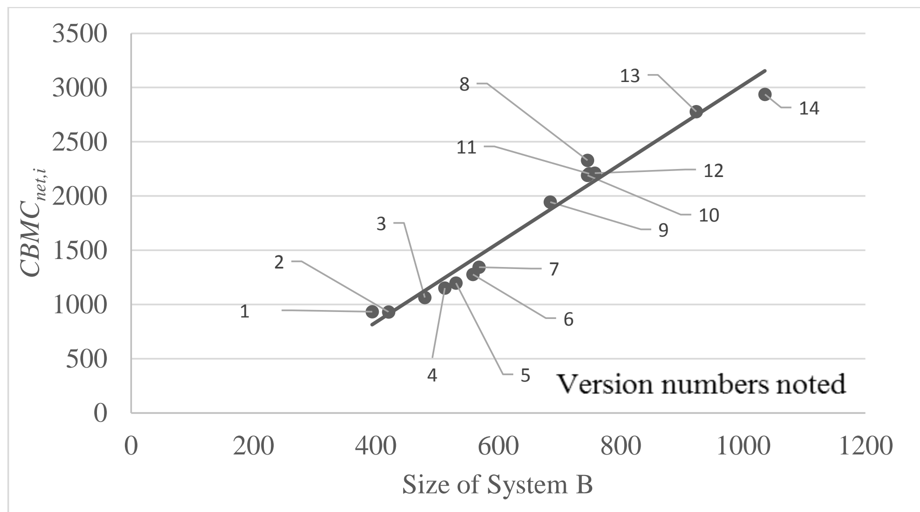

Figure 7.11

$CBMC_{net,i}$ vs. size of System B

Now, let us compare these code decay values (cd_i) with $Rate\Delta CBM_i$ and $Rate\Delta$-$CBMC_i$ values. From the Figure 7.12, the $Rate\Delta CBM_i$ values for different versions of System B are negative, zero, and positive values before and after the changes in the architecture. The decrease in the $Rate\Delta CBM_i$ between the versions was good for the system. But $Rate\Delta CBM_i$ results were not qualitatively correlated with the cd_i results.

From the Figure 7.13, we observe that there were positive values of $Rate\Delta CBMC_i$ before the change in architecture which means the system's coupling between module classes increased between the versions. There was one negative and a zero $Rate\Delta CBMC_i$ value for the versions after the change in architecture which indicates a decrease in coupling between classes. The results of $Rate\Delta CBMC_i$ are not qualitatively correlated with cd_i results. Therefore for the System B, the results of $Rate\Delta CBM_i$ and $Rate\Delta CBMC_i$ are inconsistent with cd_i.

The net code decay (CD_i) values of the System B are shown in Figure 7.7 were positive before and after the change in architecture. This is because we considered only net violations ($V_{net,i}$) at each version. We can observe that the CD_i values were high before the change in architecture and CD_i values were low after the change in architecture. This shows that the code decay of the system was reduced since the beginning of the project. Figure 7.14 and Figure 7.15 show the results of $RateCBM_{net,i}$ and $RateCBMC_{net,i}$ over time. These graphs resembles similar shape in the Figure 7.7. Both the values of $RateCBM_{net,i}$ and $RateCBMC_{net,i}$ were positive and high before the change in architecture then these values were reduced after the change in architecture. Both the results of $RateCBM_{net,i}$ and $RateCBMC_{net,i}$ were somewhat qualitatively correlated with the net

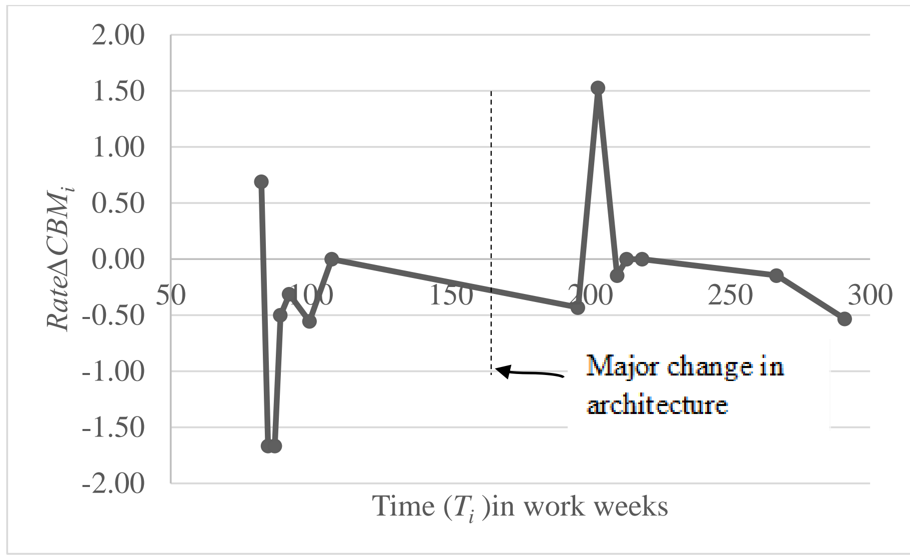

Figure 7.12

$Rate\Delta CBM_i$ for each release in System B

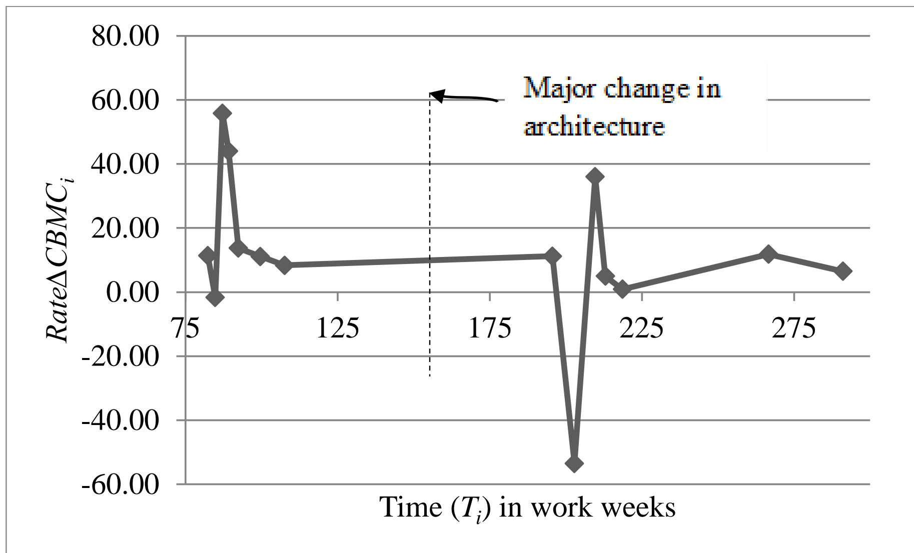

Figure 7.13

$Rate\Delta CBMC_i$ for each release in System B

code decay (CD_i) results. Therefore for the system B, the results of $RateCBM_{net,i}$ and $RateCBMC_{net,i}$ are consistent with CD_i values.

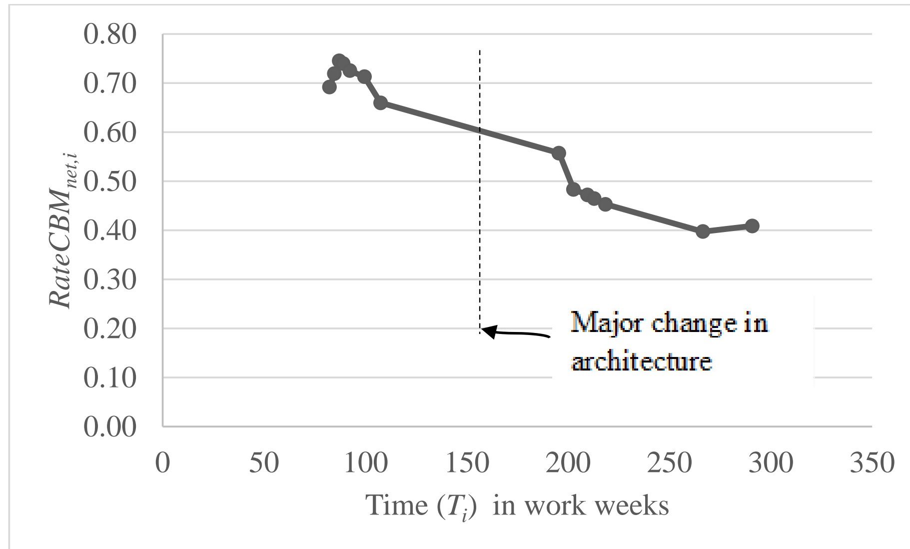

Figure 7.14

$RateCBM_{net,i}$ since the beginning of the System B

Figure 7.16 and Figure 7.17 show the scatter plots of net coupling between modules ($CBM_{net,i}$) vs. net violations ($V_{net,i}$) and net coupling between module classes ($CBMC_{net,i}$) vs. $V_{net,i}$. These plots help to compare the relation between violations and coupling ($CBM_{net,i}$ and $CBMC_{net,i}$) of the System B. The dots in these plots represent version of System B. From both the graphs we observe that as the coupling increases, violations may increase or decrease among the versions of System B. Moreover we see the three clusters (first starting from version number 1 to 7, second from versions 8 to 12, and third with 13 and 14 versions). The reason for the formation of first two clusters was the

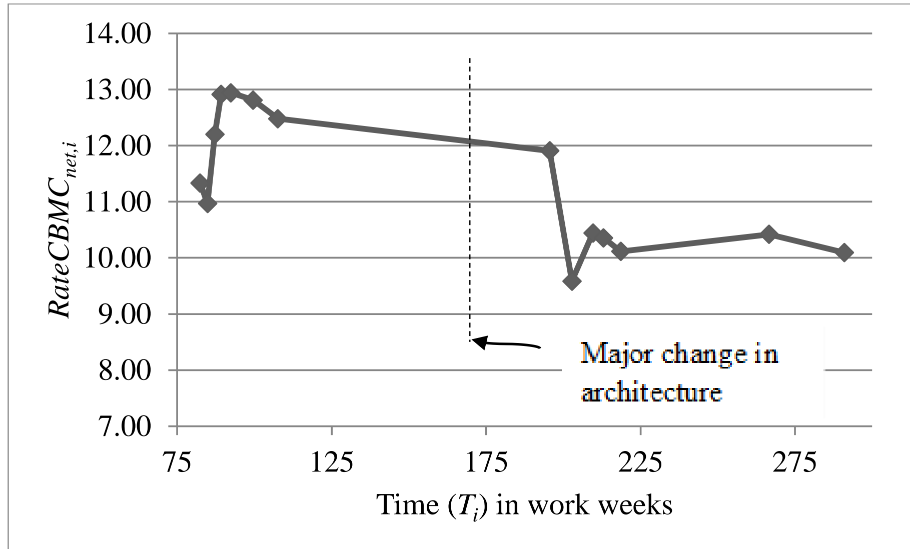

Figure 7.15

$RateCBMC_{net,i}$ since the beginning of the System B

change in architecture reduced the number of net violations. For the third cluster, an increase in functionality and changes in the GUI again increased the number of violations in versions 13 and 14. From the Figure 7.16 and Figure 7.17 it is apparent that coupling and violations are not related.

Let us consider the Figure 7.18 quadrants with architectural violations on the X-axis and coupling on the Y-axis. Figure 7.18 shows that first quadrant has low coupling and few architectural violations, the second quadrant has low coupling and a large number of architectural violations, the third quadrant has high coupling and few architectural violations, and the fourth quadrant has high coupling and a large number of architectural violations. From a software engineering perspective, the quadrants correspond to levels of software quality.

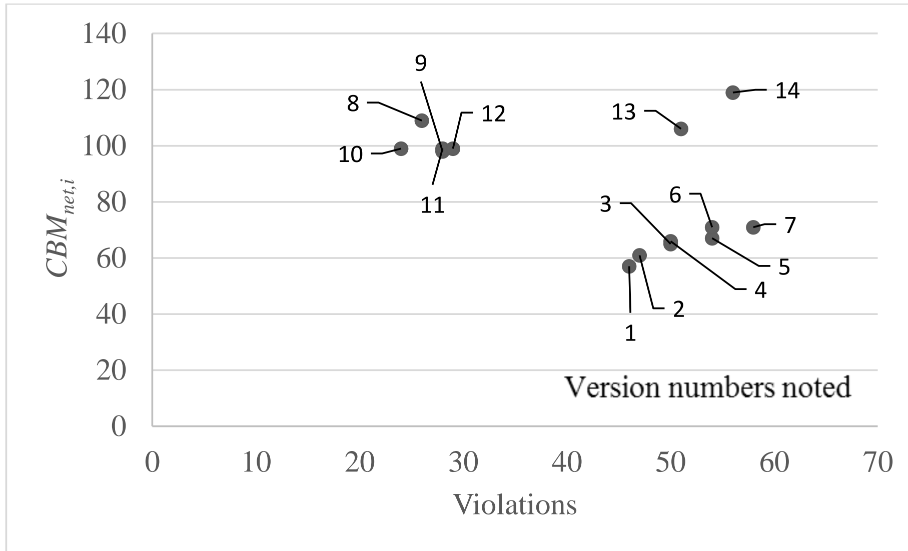

Figure 7.16

$CBM_{net,i}$ vs. net violations in System B

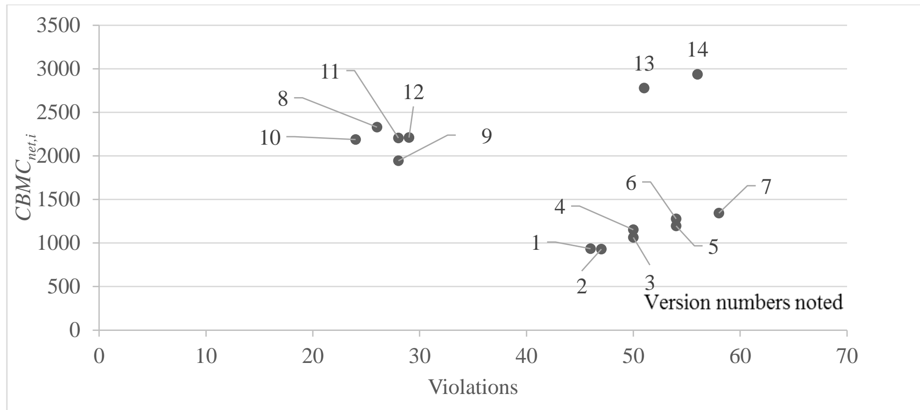

Figure 7.17

$CBMC_{net,i}$ vs. net violations in System B

- I quadrant — Very Good
- II quadrant — Good
- III quadrant — Good
- IV quadrant — Bad

Coupling		
	Quadrant III Coupling - High Architectural violations- Less Rank: Good	Quadrant IV Coupling - High Architectural violations- More Rank: Bad
	Quadrant I Coupling - Low Architectural violations- Less Rank: Very Good	Quadrant II Coupling - low Architectural violations- More Rank: Good
	Architectural violations	

Figure 7.18

Quadrants

From the Figure 7.16 and Figure 7.17, we can observe the clusters fall into different quadrants which are given below.

- II quadrant — versions 1 to 7
- III quadrant — versions 8 to 12
- IV quadrant — versions 13 and 14

This classification can help software engineering practitioners to improve their systems, by recognizing versions with poor quality. To improve the System B after the version 14, the developers could reduce coupling with architectural changes and solve architectural violations. Therefore as the coupling increases there is an opportunity of increase in violations.

CHAPTER 8

DISCUSSION

In this chapter, Section 8.1 presents answers to the research questions given in Section 1.2 in Chapter 1. Section 8.2 presents implications of our research and Section 8.3 presents threats to validity.

8.1 Research questions revisited

This section discusses the answers to our research questions.

8.1.1 Deriving architectural constraints

This section answers our first research question.

> **Research Question 1:** What is an effective method to derive architectural constraints from the source code?

We conducted case studies on two proprietary systems. In each case study, we started our code decay assessment by deriving the architectural constraints. We applied Lattix and LiSCIA as a part of our methodology to derive architectural constraints. These constraints are represented by can-use and cannot-use phrases. The expert of each system validated our derived architectural constraints before discovering the architectural violations and after discovering the architectural violations. The subjects who participated in our studies

had software development experience varying from 3 to 8 years. The participants were confident that they hit all the important constraints of packages or classes of the system. Our participants did not include any constraints regarding extensions of classes from the Spring framework since we were aware that several classes used external libraries. They spent more time in evaluating the components section of the review phase of LiSCIA to derive the constraints. They did not spend much time on other sections in the review phase of LiSCIA. Some of the participants comments from the interview session are listed below.

- One of the participants mentioned that deriving the rules was easier since they used a Model-View-Control (MVC) framework or architectural style in implementing their system.
- The start-up phase of LiSCIA help participants to recall the design decisions that happened in the past on other larger projects.
- Participants suggested having a formal example of LiSCIA to help speed the process of learning our methodology for deriving architectural constraints.
- It was difficult for participants to define the components of the system, which are logical functional units of the implemented system.
- They said that if they had followed the evaluation of components section in the review phase of LiSCIA while developing the system, they would have followed all the architectural constraints resulting in no architectural violations in the system.

Other researchers [6, 54] have analyzed the evolution of architectures by considering the early version's 'uses' relationships as their constraints or considering the architecture style rules as their constraints. However, they did not concentrate on recovering architecture constraints from the implemented system. We propose a methodology that uses LiSCIA and Lattix to derive architectural constraints from the implemented architectures. Our results and qualitative analysis of the case studies showed that the methodology was effective and required a practical level of effort for moderate sized software systems.

8.1.2 Discovering architectural violations

This section answers our second research question.

> **Research Question 2:** What is an effective method to discover the extent of violations in architectural constraints?

Our methodology includes the detection of architectural violations based on constraints derived using LiSCIA. We used Lattix as a part of our methodology to detect architectural violations. We validated the violations with the system experts. Violations may be resolved if the expert modifies the architectural constraints for any implemented desirable features that were flagged as violations. In our case studies, the experts did not choose to modify the constraints while validating the violations.

We categorized the discovered violations and we found that 'class reference' type violations were the highest percentage among all the violations in both of our case studies. Even though there were changes in the architecture of System A, the net violations $V_{net,i}$ increased from one version to other version. However, in System B, the number of net violations $V_{net,i}$ decreased after changes in the architecture.

Other architectural violations detection techniques used reflexion models [48], and heuristics [43, 44]. Reflexion models require successive refinements in the high level architecture, and heuristics use large threshold values and need several iterations to fix the thresholds. However, these methods are time consuming because of several refinements of high level architecture and several iterations to define threshold values. Our proposed methodology has no threshold values and no refinements of high level architecture. Our methodology also provides code decay assessment and is discussed in the next section.

Our results and qualitative analysis showed that our methodology effectively detected and validated the architectural violations for a given list of constraints expressed by can-use and cannot-use phrases.

8.1.3 Assessing code decay

This section answers our third research question.

> **Research Question 3:** What does the existence and repair of architectural violations over time imply about code decay?

We introduced formulas to compute code decay values of the system. These formulas are based on the net violations ($V_{net,i}$) discovered using Lattix. First, find the new violations ($V_{new,i}$), solved violations ($V_{solved,i}$), and reoccurred violations ($V_{reoccur,i}$) by comparing the $V_{net,i}$. The term 'decay' emphasize time, we considered the development time of the systems using the released dates of the versions. Second, we computed the following code decay values in violations per week.

- Code decay for a version (cd_i) — This measure gives a manager insight into the process of software development. "Did the development of this revision cause further code decay?"
- Net code decay (CD_i) — This measure gives a manager insight into the software product from the beginning of the project. "Is the product's average code decay worse or better than the past version?"
- Overall code decay (CD_n) — This measure gives a value of code decay for the current system. "Does the current system have unresolved violations and what has been the average rate of violations, namely code decay?"

For System A and System B, the values of cd_i and CD_i were fluctuating in the beginning of the project and decreased at the end of the final version of the study. The increase

in $V_{new,i}$ and decrease in $V_{solved,i}$ increased the code decay values. This means the existence of violations in the system increased the code decay values and solving or repairing violations decreased the value of code decay. Comparing the CD_n of System A (0.42 violations/week) and System B (0.19 violations/week), we conclude that System A decayed faster than System B. The developer-driven and the process-driven reasons for code decay in both the systems were the following.

- Process driven
  - The code review system was not strictly enforced.
  - Limited time allocation to design and analysis of the requirements.
  - Change in requirements from the client.
  - Project release deadlines.
- Developer driven
  - Inexperienced or novice developers worked on few releases.
  - Developers focused on pure functionality.

8.1.4 Comparison of results

This section answers our fourth research question.

> **Research Question 4:** How does our definition of code decay compare to definitions of code decay in the literature, specifically, is our definition of code decay redundant with coupling metrics?

In their systematic mapping study of code decay, Bandi, Williams, and Allen [1] presented several metrics of code decay. Coupling related metrics were widely used in measuring code decay. We evaluated our code decay results using coupling metrics (CBM and $CBMC$) [39]. Since code decay is violations over time, we compared the rate of coupling

growth to our code decay results. Coupling metrics were qualitatively partially correlated with code decay results and were inconsistent for CBM and $CBMC$.

From the results of System A in Chapter 6, we observe that the $Rate\Delta CBM_i$ results were not qualitatively correlated with the cd_i results. Before version 5, the cd_i values and $Rate\Delta CBMC_i$ were not qualitatively correlated with each other. After the version 5, the code decay cd_i values somewhat qualitatively correlated with the $Rate\Delta CBMC_i$ values. Therefore for System A, the results of $Rate\Delta CBM_i$ and $Rate\Delta CBMC_i$ are inconsistent with cd_i. The results of $RateCBM_{net,i}$ and $RateCBMC_{net,i}$ were qualitatively correlated with the net code decay (CD_i) results after the version 5 and were not qualitatively correlated before version 5.

From the results of System B in chapter 7, we observe that the results of $Rate\Delta CBM_i$ results were not qualitatively correlated with the cd_i results. Also, The results of $Rate\Delta CBMC_i$ are not qualitatively correlated with cd_i results. Therefore for the System B, the results of $Rate\Delta CBM_i$ and $Rate\Delta CBMC_i$ are inconsistent with cd_i. The results of $RateCBM_{net,i}$ and $RateCBMC_{net,i}$ were somewhat qualitatively correlated with the net code decay (CD_i) results. The results of $RateCBM_{net,i}$ and $RateCBMC_{net,i}$ are consistent with CD_i values. Therefore for the system B, the results of $RateCBM_{net,i}$ and $RateCBMC_{net,i}$ are consistent with CD_i values.

We also observed from both of our case studies, as the size of the system increased, coupling increased. This is to be expected because there are more components that need connections. Coupling is an opportunity for violations (i.e. mistakes). Therefore as cou-

pling increases we expect more violations. To improve the system, developers could reduce coupling with architectural changes and solve architectural violations.

8.2 Implications

In general the term 'decay' means a gradual process that goes unnoticed until a crisis occur. In software engineering, a major redesign or reengineering of the whole system is a crisis. To prevent this, developers should concentrate on following the planned architecture. Our methodology can help software engineering practitioners to assess code decay version by version by deriving the constraints and following those constraints while developing versions and thus, prevent code decay. Similar case studies need to conducted to to gather more empirical evidence on the practical aspects of assessing code decay.

8.3 Threats to validity

This section presents the threats to validity of our case studies.

- Construct Validity
  - In our case studies we were limited to can-use and cannot-use architectural relationships. There are other kinds of architectural rules also.
  - We chose calender time to calculate the rate for code decay rather than other measures of time (for example, level of developer's effort).
- Internal validity
  - Case studies do not control factors the way a controlled experiment does. Thus our research questions did not explore cause-effect relationships.
- External validity
  - Both of our case studies we selected were developed in the Java Spring framework in the same organization. Both the systems are database intensive, web browser interfaces and used by government clients. Our case study results (architectural constraints, architectural violations, values of code decay, coupling

measurements, and its graphs) cannot be generalized to all kinds of systems. However, our methodology and code decay measurement techniques can replicate on other systems.

- Statistical conclusion validity
  - This kind of threat is not applicable for our research.

CHAPTER 9

CONCLUSIONS

This chapter presents the conclusions, highlights the contributions of this dissertation and presents some future research directions.

9.1 Conclusions

This section presents the conclusions of our research. The hypothesis of our research is the following.

> *Given source code, a method can be developed to detect changes in the maintainability of a system by identifying the architectural violations over multiple versions.*

Maintainability is the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers [27]. We assume that architectural violations degrade maintainability of software system.

> **Research Question 1:** What is an effective method to derive architectural constraints from the source code?

Using Lattix and LiSCIA, we developed a method to derive architectural constraints. To evaluate the proposed methodology, we conducted two case studies where we validated the derived architectural constraints with experts of the systems. Our empirical results and

qualitative analysis showed that the methodology was effective and required a practical level of effort for moderate sized software systems. This supported our hypothesis that a method can be developed to derive the architectural constraints that define architectural violations. Other researchers [6] used architecture documentation or the uses relationship diagram of the older version of the system for considering the architectural constraints. They didn't use any formal method to derive architectural constraints.

> **Research Question 2:** What is an effective method to discover the extent of violations in architectural constraints?

Using Lattix we developed a method to discover architectural violations. To evaluate the proposed methodology, we conducted two case studies where we validated the discovered architectural violations with experts of the systems. Our results and qualitative analysis showed that the methodology was effective in detecting architectural violations for a given list of constraints that represent can-use and cannot-use rules. This supported our hypothesis that a method can be developed to detect architectural violations.

'Class reference' was the major architectural violation category experienced in the case study software systems. A large number of class reference violations in the software increases undesirable coupling which makes the system hard to maintain.

Researchers used architecture conformance techniques such as reflexion models [48] and heuristics [43, 44]. The disadvantages of reflexion model requires successive refinements or iterations in the high level mental model to discover the absences and divergences in the source code. The expressiveness of the reflexion models is limited to regular expressions but no other types of rules. In addition, these models focus on the conformance of

the design and implementation and do not deal with different architecture styles (e.g., layered architecture). On the other hand, the drawback of the heuristics technique is using many threshold values in heuristics. This architecture conformance process based on the proposed heuristics should follow an iterative approach — running the heuristics several times, starting with rigid thresholds. After each execution, the new warnings should be evaluated by the architect. The selecting of threshold values may takes several iterations. Our methodology overcome these disadvantages.

> **Research Question 3:** What does the existence and repair of architectural violations over time imply about code decay?

New violations in a system increases the code decay values (cd_i, CD_i, and CD_n) and solving or repairing violations decreases the value of code decay. This supported our hypothesis because degradation over multiple versions is key to the concept of "decay." This means that over time, the system becomes harder to change than it should be. Our results showed that System A is decaying faster than System B. This also illustrates that systems have various decay histories over time.

In the systematic mapping study of code decay, Bandi, Williams, and Allen [1] concluded that the coupling related metrics were used to assess different forms of code decay. This dissertation proposed a complementary and alternative approach to assess code decay that uses architecture violations over development time and concluded that coupling is not consistently related to architectural violations.

Research Question 4: How does our definition of code decay compare to definitions of code decay in the literature, specifically, is our definition of code decay redundant with coupling metrics?

We compared our measure of code decay to measures proposed in the literature [39], namely coupling metrics (coupling between modules (CBM) and coupling between module classes ($CBMC$)). The coupling metrics were sometimes qualitatively correlated with code decay results and sometimes not. Trends were inconsistent in between CBM and $CBMC$. In addition, our results showed that coupling metrics were qualitatively correlated with the size of the system as expected. We note that coupling is an opportunity of violations. We concluded that coupling is not consistently related to violations, and thus not consistently related to code decay.

9.2 Contributions

Figure 9.1 shows a chart of our contributions. The contributions of this dissertation are four fold.

1. We performed a literature review on code decay and conducted a systematic mapping study on code decay that gives the classification of the code decay and its related terms, classification of code decay detection techniques (human-based and metric-based approaches), and the metrics used to measure the code decay.
2. We proposed a methodology to derive architectural constraints that uses a reverse engineering tool and LiSCIA. In our case studies we used Lattix as our reverse engineering tool.
3. We proposed a methodology that also uses a reverse engineering tool to discover architectural violations and validate them. In our case studies we used Lattix as our reverse engineering tool.
4. We also proposed an alternative and complementary method to assess code decay which uses code decay indicator measures (cd_i, CD_i, and CD_n). The empirical evidence and qualitative assessment shows our methodology is practical for deriving architectural constraints, discovering architectural violations, and assessing code decay.

5. We qualitatively compared our code decay results with coupling metrics (CBM and $CBMC$) and concluded that coupling is not consistently related to violations, and thus not consistently related to code decay.

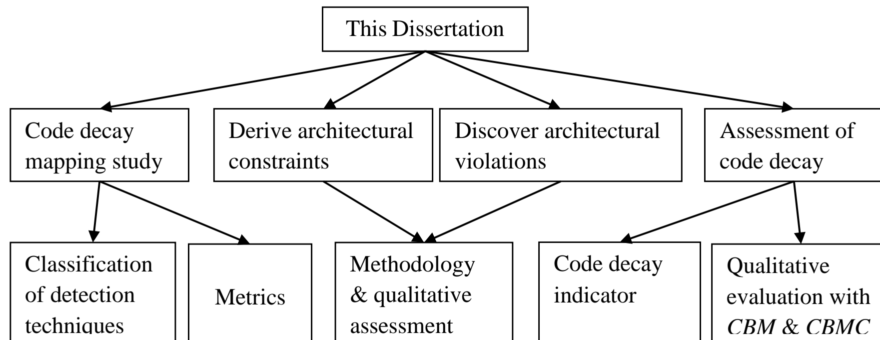

Figure 9.1

Contributions

9.3 Publications

This dissertation resulted in a peer reviewed conference paper which is given below and Table 9.1 shows potential venues for publishing our research.

A. Bandi, B. J. Williams, and E. B. Allen, "Empirical evidence of code decay: A systematic mapping study," *Proceedings: 20th Working Conference on Reverse Engineering*, pp. 341–350.

9.4 Future research directions

We identified the following research dimensions in the code decay area:

- There is need for research on automated detection techniques of code decay. Automated detection means automatic decision-making in identifying violations in architectural rules, design rules, and source code standards. There is a need to build automated classifiers that support developers in locating architecturally-relevant code smells and detecting violations of architectural constraints.

Table 9.1

Publication plan

Publication Venue	Content to publish
Information and Software Technology	Extended version of systematic mapping study
International Conference on Program Comprehension	Case study on deriving architectural constraints
International Conference on Software Analysis, Evolution, and Reengineering	Architecture violation discovery methodology and one case study
International Conference on Software Maintenance	Assessing code decay over multiple versions of software
Empirical Software Engineering Journal	Content of dissertation

- Research should also be conducted to operationalize the various code decay related terms to move toward a consensus in defining the phenomenon of code decay at various levels of abstraction.
- There is a need for systematic literature reviews on architectural violations and design paradigms for identifying the best practice. The research focus must be in identifying and minimizing code decay with respect to procedures, technologies, methods or tools by aggregating information from empirical evidence.
- There is a demand for research and for building tools that support developers in locating architecturally relevant code smells.
- There is a need for research to evaluate techniques to prevent code decay by identifying architecture and design rule violations at the time of check-in to the version control system during the implementation phase of software development life cycle.
- Van Gurp and Bosch [68] indicate expressiveness of representing the architecture is one of the research challenges of the large and complex systems.
- Applying the research on visual analytics to represent the software architectures and to track the violations of the architecture over different versions of the system is another important area of research.
- Research should also be conducted to find techniques for identifying the defect prone source code areas in software using architectural violations. This can help in determine which part of source code can be refactored to reduce code decay.

- An exploratory survey would be helpful to find insights of developers whether they really care about architectural violations in software.
- Researchers should conduct case studies on deriving architectural constraints using other architectural relationships such as composition of modules, aggregation of modules, and database relationships besides can-use and cannot-use relationships.
- Researchers could perform research to analyze the code decay of systems by considering the level of effort time of the developers on the project instead of calender time.

REFERENCES

[1] A. Bandi, B. J. Williams, and E. B. Allen, "Empirical evidence of code decay: A systematic mapping study," *Proceedings: 20th Working Conference on Reverse Engineering*, pp. 341–350.

[2] P. Bengtsson, N. Lassing, J. Bosch, and H. V. Vliet, "Architecture-level modifiability analysis," *Journal of Systems and Software*, vol. 69, 2004, pp. 129–147.

[3] P. O. Bengtsson and J. Bosch, "Scenario-based architecture reengineering," *Proceedings: 5th International Conference on Software Reuse*, 1998, pp. 308–307.

[4] E. Bouwers, *Metric-based Evaluation of Implemented Software Architectures*, doctoral dissertation, University of Delft, Jan. 2013.

[5] E. Bouwers and A. van Deursen, "A lightweight sanity check for implemented architectures," *IEEE Software*, vol. 27, no. 4, 2010, [Primary Study].

[6] J. Brunet, R. A. Bittercourt, D. Serey, and J. Figueiredo, "On the evolutionary nature of architectural violations," *Proceedings: 19th Working Conference on Reverse Engineering*, 2012, pp. 257–266, [Primary Study].

[7] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object-oriented design," *Transactions on Software Engineering*, vol. 20, 1994, pp. 476–493.

[8] O. Ciupke, "Automatic detection of design problems in object-oriented reengineering," *Proceedings: International Conference and Exhibition on Technology of Object-oriented Languages and Systems*, 1999, [Primary Study].

[9] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architecture: Views and Beyond*, Addison-Wesley, Boston, MA, 2003.

[10] P. C. Clements, *Active reviews for intermediate designs*, Tech. Rep. CMU/SEI-2000-TN-009, Carnegie Mellon University, 2000.

[11] Z. Codabux and B. J. Williams, "Managing technical debt: An industrial case study," *Proceedings: 4th International Workshop on Managing Technical Debt*, 2013, pp. 8–15.

[12] Z. Codabux, B. J. Williams, and N. Niu, "A quality assurance approach to technical debt," *Proceedings: International Conference on Software Engineering and Practice*, 2014.

[13] D. Cotroneo, R. Natella, and R. Pietrantuono, "Is software aging related to software metrics?," *Proceedings: 2nd International Workshop on Software Aging and Rejuvenation*, 2011, pp. 1–6, [Primary Study].

[14] L. Dobrica and E. Niemela, "A survey on software architecture analysis methods," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, 2002, pp. 638–653.

[15] T. Dybå and T. Dingsøyr, "Empirical studies of agile software development: A systematic review," *Information and Software Technology*, vol. 50, 2008, pp. 1–27.

[16] T. Dybå and T. Dingsøyr, "Strength of evidence in systematic reviews in software engineering," *Proceedings: Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2008.

[17] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," *Proceedings: 30th International Conference on Software Engineering*, 2008, pp. 391–400.

[18] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? Assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, 2001, pp. 1–12, [Primary Study].

[19] M. Fowler and K. Beck, *Refactoring: Improving the design of existing code*, Addison Wesley, 1999.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, 1994.

[21] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," *Proceedings: European Conference on Software Maintenance and Reengineering*, pp. 255–258.

[22] M. W. Godfrey and E. H. S. Lee, "Secrets from the Monster: Extracting Mozilla's software architecture," *Proceedings: 2nd International Symposium on Constructing Software Engineering Tools*, 2000.

[23] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, 2012, pp. 1276–1304.

[24] S. Hassaine, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "ADvISE: Architectural decay in software evolution," *Proceedings: 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 267–276, [Primary Study].

[25] L. Hochestein and M. Lindvall, "Combating architectural degeneration: A survey," *Information and Software Technology*, vol. 47, no. 10, 2005, pp. 643–656.

[26] L. Hochstein and M. Lindvall, "Diagnosing architectural degeneration," *Proceedings: 28th Annual NASA Goddard Software Engineering Workshop*, 2003, pp. 137–142.

[27] International Organization for Standardization, "ISO/IEC 25010: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," 2011.

[28] C. Izurieta and J. M. Bieman, "Software designs decay: A pilot study of pattern evolution," *Proceedings: 1st International Symposium on Empirical Software Engineering and Measurement*, 2007, pp. 449–451, [Primary Study].

[29] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, *Reporting Experiments in Software Engineering: Guide to Advanced Empirical Software Engineering*, Springer, London, 2008.

[30] E. Johansson and M. Host, "Tracking degradation in software product lines through measurement of design rule violations," *Proceedings: International Conference on Software Engineering and Knowledge Engineering*, 2002, [Primary Study].

[31] R. Kazman, L. Bass, G. Abowd, and M. Webb, "SAAM: A method for analyzing the properties of software architectures," *Proceedings: 16th International Conference on Software Engineering*, 1994.

[32] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," *Proceedings: IEEE International Conference on Engineering of Complex Computer Systems*, 1998, pp. 68–78.

[33] F. Khomh, M. D. Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," *Proceedings: 16th Working Conference on Reverse Engineering*, 2009, pp. 75–84, [Primary Study].

[34] B. A. Kitchenham, D. Budgen, and O. P. Brereton, "Using mapping studies as the basis for further research–A participant-observer case study," *Information and Software Technology*, vol. 53, 2009.

[35] B. A. Kitchenham and S. Charters, *Guidelines for Performing Systematic Literature Reviews in Software Engineering*, Tech. Rep. EBSE, Keele University and University of Durham, 2007.

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

[37] N. Lassing, D. Rijsenbrij, and H. V. Vliet, "On software architecture analysis of flexibility, complexity of changes: Size isn't everything," *Proceedings: 2nd Nordic Software Architecture Workshop*, 1999.

[38] Z. Li and J. Long, "A Case Study of Measuring Degeneration of Software Architectures from a Defect Perspective," *Proceedings: 18th Asia Pacific Software Engineering Conference*, 2011, pp. 242–249, [Primary Study].

[39] M. Lindvall, R. Tesoriero, and P. Costa, "Avoiding architectural degeneration: An evaluation process for software architecture," *Eighth IEEE Symposium on Software Metrics*, 2002, pp. 77–86, [Primary Study].

[40] C.-H. Lung, S. Bot, K. Kalaichelvan, and R. Kazman, "An approach to software architecture analysis for evolution and reusability," *Proceedings: Annual International Conference on Computer Science and Software Engineering*, 1997.

[41] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," *Proceedings: 16th European Conference on Software Maintenance Reengineering*, 2012, pp. 277–286, [Primary Study].

[42] I. Macia, A. Garcia, A. von Staa, J. Garcia, and N. Medvidovic, "On the impact of aspect-oriented code smells on architecture modularity: An exploratory case study," *Proceedings: Fifth Brazilian Symposium on Software Components, Architectures and Reuse*, 2011, pp. 41–50, [Primary Study].

[43] C. Maffort, M. T. Valente, M. Bigonha, N. Anquetily, and A. Hora, "Heuristics for discovering architectural violations," *Proceedings: 20th Working Conference on Reverse Engineering*, 2013, pp. 222–231.

[44] C. Maffort, M. T. Valente, R. Terra, M. Bigonha, N. Anquetil, and A. Hora, *Mining Architectural Violations from Version History*, Tech. Rep.

[45] M. V. Mäntylä, J. Vanhanen, and C. Lassenius, "Bad smells–Humans as code critics," *Proceedings: 20th IEEE International Conference on Software Maintenance*, 2004, pp. 399–408, [Primary Study].

[46] R. Marinescu, "Detecting design flaws via metrics in object-oriented systems," *Proceedings: 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, 2001, [Primary Study].

[47] G. Molter, "Integrating SAAM in domain-centric and reuse-based development processes," *Proceedings: 2nd Nordic Workshop on Software Architecture*, 1999.

[48] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, 2001, pp. 364–380.

[49] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez, "In search of a metric for managing architectural debt," *Joint 10th Working IEEE/IFIP Conference on Software Architecture (WICSA) and 6th European Conference on Software Architecture (ECSA)*, 2012, pp. 91–100.

[50] M. C. Ohlsson, A. von Mayrhauser, B. McGuire, and C. Wohlin, "Code Decay Analysis of Legacy Software through Successive Releases," *IEEE Aerospace Conference*, 1999, pp. 69–81, [Primary Study].

[51] S. M. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," *Proceedings: 3rd International Symposium on Empirical Software Engineering Measurement*, 2009, [Primary Study].

[52] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells are harmful? A study of God classes and Brain classes in the evolution of three open source systems," *Proceedings: 26th IEEE International Conference on Software Maintenance*, 2010, pp. 1–10, [Primary Study].

[53] D. L. Parnas, "Software aging," *Proceedings: 16th International Conference on Software Engineering*, 1994, pp. 279–287.

[54] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. Mendonca, "Static architecture conformance checking: An illustrative overview," *IEEE Software*, vol. 27, no. 5, 2010, pp. 82–89.

[55] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, 1992.

[56] D. Raţiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws," *Proceedings: 8th European Conference on Software Maintenance and Reengineering*, 2004, pp. 223–232, [Primary Study].

[57] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, 2013.

[58] M. Riaz, M. Sulayman, and H. Naqui, "Architectural decay during continuous software evolution and impact of 'design for change' on software architecture," *Communications in Computer and Information Science*, vol. 59, 2009, [Primary Study].

[59] J. Rosik, A. L. Gear, J. Buckley, and M. A. Babar, "Assessing architectural drift in commercial software development: A case study," *Software–Practice and Experience*, vol. 41, no. 1, 2011, [Primary Study].

[60] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, 2009, pp. 131–164.

[61] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Detecting design flaws via metrics in object-oriented systems," *Proceedings: 20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005, pp. 167–176.

[62] S. Sarkar, G. Maskeri, and S. Ramachandran, "Discovery of architectural layers and measurement of layering," *The Journal of Systems and Software*, vol. 82, no. 11, 2009, [Primary Study].

[63] T. Schanz and C. Izurieta, "Object oriented design pattern decay: A taxonomy," *Proceedings: International Symposium on Empirical Software Engineering and Measurement*, 2010, [Primary Study].

[64] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," *Proceedings: International Symposium on Empirical Software Engineering and Measurement*, 2010, [Primary Study].

[65] R. Terra and M. T. Valente, "A dependency constraint language to manage object-oriented software architectures," *Software—Practice and Experience*, vol. 32, no. 12, 2009, pp. 1073–1094.

[66] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems," *Proceedings: 12th Working Conference on Reverse Engineering*, 2005, [Primary Study].

[67] R. T. Tvedt, P. Costa, and M. Lindvall, "Does the code match the design? A process for architecture evaluation," *Proceedings: International Conference on Software Maintenance*, 2002, pp. 393–401, [Primary Study].

[68] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *The Journal of Systems and Software*, vol. 61, 2002, pp. 105–119, [Primary Study].

[69] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of God classes," *Proceedings: 16th Working Conference on Reverse Engineering*, 2009, [Primary Study].

[70] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*, Springer, Berlin, 2012.

[71] S. Wong, M. Kim, and M. Dalton, "Detecting software modularity violations," *Proceedings: 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 411–420, [Primary Study].

[72] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?," *Proceedings: IEEE International Conference on Software Maintenance*, 2012, pp. 306–315, [Primary Study].

APPENDIX A

DETAILS OF MAPPING STUDY

This Appendix presents the details of conducting our mapping study. Our mapping study was published in an article by Bandi, Williams, and Allen [1].

A.1 Search strategy

The search was limited to the automated search techniques and did not performed manual search because most of the conferences and journals related to our topic are included in these electronic databases. In addition, we performed a bibliography check of every primary study to include any additional relevant articles that focused on our research questions. The high-level search string with keywords and their synonyms is shown below.

> (((software OR code OR architecture OR system OR design) AND (erosion OR drift OR degeneration OR decay OR smell OR aging OR grime OR rot OR violation*) AND (detect* OR measure* OR metric* OR assess* OR evaluat*)))

An asterisk (*) means any string. The search strategy is a trade-off between finding all relevant primary studies from the results of the search string and finding too many irrelevant studies. The different stages in our study are:

1. Search electronic databases using the above search string.
2. Eliminate irrelevant studies reviewing the titles and remove duplicate articles using EndNote software.
3. Filter articles based on the abstracts, inclusion and exclusion criteria, and quality assessment criteria.
4. Obtain primary studies and check the bibliography of primary studies to include any additional appropriate studies.

To the best of our knowledge there were no earlier mapping studies or systematic reviews conducted on code decay. Therefore the timeframe for our search was not limited

and articles up to May 2013 were considered. After searching peer-reviewed papers from the data sources and reviewing their titles, we exported only the relevant articles to EndNote to organize our bibliography. We excluded several false positives that are not related to software engineering by limiting the search topic to computer science and engineering. However, search engines are not sophisticated enough to get articles only relevant to software engineering. Based on the titles, we manually excluded papers with a focus on biology, mechanical, chemical and electric engineering topics. EndNote has an advanced feature to remove duplicate articles based on the title, author names and conference titles. After eliminating duplicate articles, there were 205 unique articles remaining.

A.2 Inclusion and exclusion criteria

The basis for selection of primary studies is the inclusion and exclusion criteria. During this stage, we read all the abstracts to determine if the paper focused on identification or empirical evaluation of code decay. We included a paper if it is focused on either on identification or empirical evaluation of code decay. Of the 205 unique studies, 160 papers were selected after a review based on abstracts.

Studies were included if they presented empirical evidence of code decay that includes architecture and design rule violations. The term "decay" refers to the gradual decrease in quality. Therefore, only studies that evaluated decay on more than one version of a system developed over a period of time were included. Time is an important factor of decay. So papers that only took a snapshot of a single version of a system were excluded. Studies that concentrate on code decay detection techniques and analysis of metrics on

proprietary or open source systems are included. Invited talks, expert opinions without empirical evidence were excluded. Table A.1 shows the different criteria for inclusion and exclusion of papers. From the 140 papers after abstract review, we applied our inclusion and exclusion criteria and ended up with 49 papers.

A.3 Quality assessment criteria

Assessing quality criteria of studies important for rigor in selecting primary studies. We defined quality assessment criteria similar to Dybå and Dingsøyr [15, 16] and applied this criteria to the 49 papers resulting from inclusion and exclusion. We established quality criteria based on the types of studies that will be included in the review. There are quality assessment criteria to assess case studies and archival analysis, controlled and quasi-experiments, and peer-reviewed experience reports and surveys from industrial examinations. Based on the guidelines and examples in the literature [15, 16, 29, 60] the quality criteria checklist for different types of research studies are shown in Table A.2. The criteria for the experience report papers are not as rigorous as the other criteria for experiments and case studies. The primary focus for an experience report is peer-review and research value. We applied the quality criteria uniformly for all the papers. The accepted papers pass a specified number questions for the paper to be included in the primary studies. The acceptance criteria is as follows: case studies, 6 of the 8 criteria required; controlled/quasi-experiments 9 of the 11 criteria required; and experience reports, 3 out of 3 criteria required. Applying the quality criteria resulted in 27 primary studies. A bibliography check of the primary studies led to 3 additional primary studies. The results of quality criteria

Table A.1

Inclusion and exclusion criteria

Inclusion	Exclusion
–Papers that detect code decay at any level of abstraction (i.e., class, package, subsystem, architecture, and software system etc.)	–Studies focused on just one version of the system
–Papers that identify procedures and techniques for identifying code decay	–Papers that are based only on expert opinion
–Studies of measurement / metrics analysis of industrial and open-source systems	–Invited talks, introductions to special issues, tutorials, and mini-tracks
–Empirical studies focused on code decay in systems over time	–Studies presented in languages other than English
–Empirical studies focused on architectural violations in systems over time	–Studies without any empirical evidence
–Empirical studies focused on design rule violations in systems over time	
–Studies investigated on the systems with more than one version/certain period of development time.	
–Evaluation of studies focused on different versions of the system	

is shown in Tables A.3, A.4, and A.5. Most of the papers excluded during this stage were idea-based and short papers.

A.4 Data extraction

Once the list of primary studies was decided, the data from the primary studies were extracted. The extracted data from the studies were recorded in a separate Microsoft Excel spreadsheet. Table A.6 provides the details of the data extraction form. This form facilitates collecting the data items specifically related to our mapping study research questions. If the same study appeared in more than one publication, we included the most recent or the most comprehensive version (i.e. the journal article). After applying the quality assessment criteria, we divided the papers into two sets (70% and 30%). Two researchers studied those papers in detail and independently extracted the data from those sets respectively and then independently reviewed a sample of each other's data extraction forms for consistency. We used a third researcher's opinion to resolve any inconsistencies. As a result there were no disagreements on extracted data. During this process, after extracting the data from sample of papers, new keywords are included in the search string if necessary.

A.5 Data mapping

The extracted data from the primary studies was mapped by identifying similarities in the detection techniques and the metrics used in the detection process.

Table A.2

Quality assessment criteria

Criteria	Description if warranted	Controlled/ quasi-experiments	Case studies or Archival studies	Human experience/ Surveys
Is the paper based on research (or is it merely a position statement based on expert opinion)?	The paper provides empirical evidence of claims made	Yes	Yes	Yes
Is there a clear statement of the aims of the research?	–The objectives of the research are clearly defined –The research questions and/or hypotheses are clearly stated	Yes	Yes	Yes
Is there an adequate description of the context in which the research was carried out?	–The authors provide a description of the system, organization and context of the study e.g.,: Type of system (e.g., proprietary or open-source) Description of system (e.g., size, programming language) –Description of development environment (e.g., distributed, global, co-located, team size) and study setting (e.g., classroom, industry)	Yes	Yes	Yes
Was the research design appropriate to address the aims of the research?	–The experiment design is described in detail –Type of study lends itself to addressing the research questions/hypothesis.	Yes	Yes	

Table A.2

(continued)

Criteria	Description if warranted	Controlled/ quasi-experiments	Case studies or Archival studies	Human experience/ Surveys
Was the recruitment strategy appropriate to the aims of the research? (human subjects)		Yes		
Was there a control group with which to compare treatments?	The authors clearly identify treatment and control variables	Yes		
Was the data collected in a way that addressed the research issue?	–Described data collection procedure –Clear description and presentation of metrics –Explicit statement of data collection methods –Validation of results using quality control methods	Yes	Yes	
Was the data analysis sufficiently rigorous?	–In-depth description of the analysis process –Clear description of statistical methods used to evaluate metrics identified –Appropriate use of statistical methods –Does the data support the findings –Deviations from hypotheses reported	Yes	Yes	
Has the relationship between researcher and participants been adequately considered?		Yes		

Table A.2

(continued)

Criteria	Description if warranted	Controlled/ quasi- experiments	Case studies or Archival stud- ies	Human experi- ence/ Surveys
Is there a clear statement of findings?	–Limitations and threats to validity –Outcomes are explicitly quantified –Research questions / hypothesis discussed in relation to findings –Conclusions are justified by the results –An evaluation and analysis of results provided	Yes	Yes	
Is the study of value for research or practice?	–Contributions and implications of research described –Describe application to SE practitioner and research community –Open research questions identified –Future work stated	Yes	Yes	Yes

Table A.3

Quality assessment criteria results for case studies/archival studies

Study	Research	Objective/ Hypothesis	Study context	Study design	Data collection	Data analysis	Findings	Value	Count
[6]	1	1	1	1	1	1	1	1	8
[13]	1	1	1	1	1	1	1	1	8
[18]	1	1	1	1	1	1	1	1	8
[24]	1	1	1	1	1	1	1	1	8
[28]	1	1	1	0	0	1	1	1	6
[30]	1	1	1	1	1	1	1	1	8
[33]	1	1	1	1	1	1	1	1	8
[38]	1	1	1	1	1	1	1	1	8
[39]	1	1	1	1	1	1	1	1	8
[46]	1	1	1	1	1	1	1	1	8
[41]	1	1	1	1	1	1	1	1	8
[42]	1	1	1	1	1	1	1	1	8
[45]	1	1	1	1	1	1	1	1	8
[51]	1	1	1	1	1	1	1	1	8
[52]	1	1	1	1	1	1	1	1	8
[50]	1	1	1	1	1	1	1	1	8
[56]	1	1	1	1	1	1	1	1	8
[59]	1	1	1	1	1	1	1	1	8
[63]	1	1	1	1	1	1	1	1	8
[64]	1	1	1	1	1	1	1	1	8
[67]	1	1	1	1	1	1	1	1	8
[66]	1	1	1	1	1	1	1	1	8
[68]	1	1	1	1	1	1	1	1	8
[69]	1	1	1	1	1	1	1	1	8
[71]	1	1	1	1	1	1	1	1	8
[72]	1	1	1	1	1	1	1	1	8

Table A.4

Quality assessment criteria results for controlled/quasi-experiments

Study	Research	Objective/ Hypothesis	Study context	Experiment design	Control group	Recruitment strategy	Data collection	Data analysis	Relationship between researcher and participants	Findings	Value	Count
[62]	1	1	1	1	1	0	1	1	0	1	1	9
[58]	1	1	1	1	1	0	1	1	0	1	1	9

Table A.5

Quality assessment criteria results for experiences/surveys

Study	Research questions	Study context	Value	Count
[5]	1	1	1	3
[8]	1	1	1	3

Table A.6

Data extraction form

S.No	Data item	Description
Study citation		
1	Bib code	Unique identifier for the study
2	Date of data extraction	
3	Bibliography information	Author, conference/journal, tile of the paper and year
Study details		
1	Objective	What is the objective / goal of the study?
2	Design of study	Describe the study design and type of study (e.g., controlled experiment, survey, case study, etc.)
3	Research hypothesis	What are the research hypotheses and/or research questions?
4	Definition of code decay	Verbatim description from the study
5	Context of study	Description of context (e.g., industrial/ proprietary/ open source, system size, programming language, distributed/global/co-located development teams, team size, etc.)
6	Metrics of code decay	What measures are used to assess code decay and at what level of abstraction (e.g., architecture, class-level)?
7	Software tools for code decay analysis	What software tools are used (e.g., Coverity, FindBugs)?
8	Techniques to detect code decay	What detection methods are applied?
9	Data collection	How the data is acquired?
10	Data analysis	What statistical methods are used to analyze data?
Study findings		
1	Findings and conclusions	What are the findings and conclusions of the research?
2	Limitations	Threats to validity
3	Significance	Research and practice

APPENDIX B

DETAILS OF LISCIA

Lightweight Sanity Check for Implemented Architectures (LiSCIA) overview and questionnaire is quoted with minor modifications from the dissertation of Eric Bouwers [4].

General overview. LiSCIA consist of two phases, a start-up phase and a review phase. The start-up phase only needs to be conducted once during the first evaluation of a system. The review phase should be conducted during every evaluation.

Start-up Phase. The code of a system is divided into source files that encode some part of the functionality of the system. In order to have a good grasp of the system we need to divide the code into logical groups of functionality. Such a group of functionality is called a component.

A component can either represent some business functionality, such as Accounting and Stocks, or a more technical functionality, such as GUI and XML-processing. The evaluation can be applied to both decompositions. Also, the evaluation can be applied to the same version of a system using different decompositions. In this way, different views on the architecture can be explored, which can lead to more insight and a better understanding of the implemented architecture.

Defining components. Make a list of logical groups of functionality that should be in the system. This list of functionalities should contain about 5 to 10 different core-functionalities. Usual functionalities for a typical application can be things such as User Interface, Input processing, Administration or Utilities. Ideally, each core-functionality is a component within the system. If there are more than 15 components try to group some of the components together. For example, the components GUI for administrators and GUI for users can be grouped into a component GUI. If there is already a list of components in the documentation this list can be used.

Defining name-patterns. For each component, try to determine which source files belong to it by defining a pattern on the directory-/file-names of the source-files. For example, all files that implement the GUI are in a subdirectory called GUI, the name-pattern for this component then becomes: Component GUI, name-pattern = .*/GUI/.* Note that the name-patterns for the components should be exclusive. In other words, a single file should only be matched to a single component.

Inventory of Technologies. Determine the technologies used within the system by:

- Listing all different file-extensions used in the system
- Mapping each file-extension to a technology

Review Phase.

Evaluation of Source Groups. Determine whether all source-files in the system belong to a component by applying the name-patterns to the sources in

the system. All source files that cannot be placed under a component fall into one of the following two categories:

- Code that can be removed because it does not implement any functionality
- Code that should be put under a, possibly new, component

When code should be put into an existing component answer the following questions:

1. Should the name-pattern be expanded or should the code be relocated on the file-system?
2. Why does the code fall outside of its desired component?

   When code should be put into a new component, answer the following questions:

3. Why has this component only surfaced now?
4. Is it likely that more components will emerge?

Evaluation of Component Functionality. Answer the following questions about the way the sources are grouped into the components:

1. Are the name-patterns defined for the components straightforward or complicated? In other words, are the sources located in the file-system according to the components or according to a different type of decomposition?
2. Is all functionality that is needed from the system available in the components?
3. Can the functionality of each component be described in a single sentence? If not, why?
4. Do multiple components implement the same functionality? (For example, do two components parse the same messages?)
5. Does any component contain functionality that is also available in a library/framework? If so, why is this library/framework not used?

Evaluation of Component Size. Determine the size of each component by counting the lines of code for each file, and then summing up the lines of code of all files in a component. Answer the following questions about the size of the components:

1. Are the sizes of the components evenly distributed?
2. If not, what is the reason for this uneven distribution?
3. Is the reason in-line with the expectations about the functionality?

   When previous results are available:

4. Which component has grown the most? Is this to be expected?
5. Which component has been reduced the most? Is this to be expected?
6. Is the ordering of components on size heavily changed? Is this to be expected?

Evaluation of Component Dependencies. Determine the dependencies between components by determining the dependencies on file-level (or lower). After this, for each dependency between two files, determine the components of the files. If no dependency between the components existed, add this dependency, otherwise add an extra weight to the dependency. Answer the following questions about the dependencies between components:

1. Are there any circular dependencies between the components? If so, why? List those constraints.
2. Are there any unexpected dependencies between components? List those constraints
3. Which component depends on most other components, is this to be expected?
4. Which component is the most depended on (which component has the most incoming dependencies)? Is this to be expected?

   When previous results are available:
5. Are there any new dependencies? Is this to be expected?
6. Are there any dependencies that were removed? Is this to be expected?

Evaluation of Technologies. Answer the following questions about the technologies:

1. Is each technology needed in the system? Can the functionality be encoded in a different technology that is used in the system?
2. Is each technology being used for the purpose it was designed for?
3. Is the combination of technologies common? Does the official documentation of the technologies mention the other technologies?
4. Is each technology still supported by an active community or vendor?
5. Are the latest versions for each technology used? If not, why?

   When previous results are available:
6. Are there any new technologies added? If so, why?
7. Are there any technologies that were discarded? If so, why?

APPENDIX C

SYSTEM A ARTIFACTS

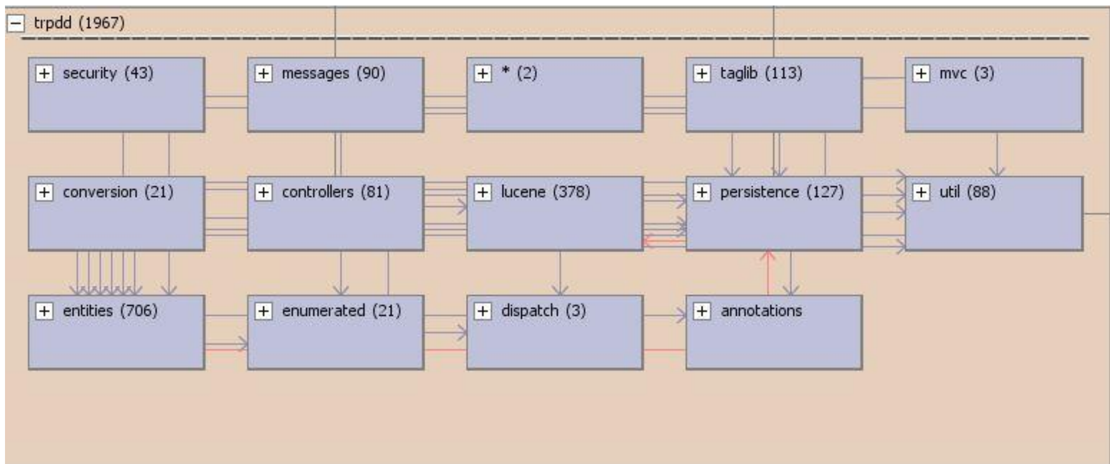

Figure C.1

System A conceptual architecture

oot		..1	..2	..3	..4	..5	..6	..7	..8	..9	10	11	12	13	14	15	16
*	1	82															
spring.util	2		12							1		1	2				
security	3			43													
messages	4				90												
*	5	8				2											
taglib	6						113										
mvc	7							3									
conversion	8								21								
controllers	9									81							
lucene	10									9	378	6					
persistence	11						3	3	3	36	14	127		26			
util	12		1				6	1		6	1	3	88				
entities	13						14	1	6	81	10	38	3	706			
enumerated	14						1							4	21		
dispatch	15									1	5					3	
annotations	16											4		6			

Figure C.2

System A dependency structure matrix

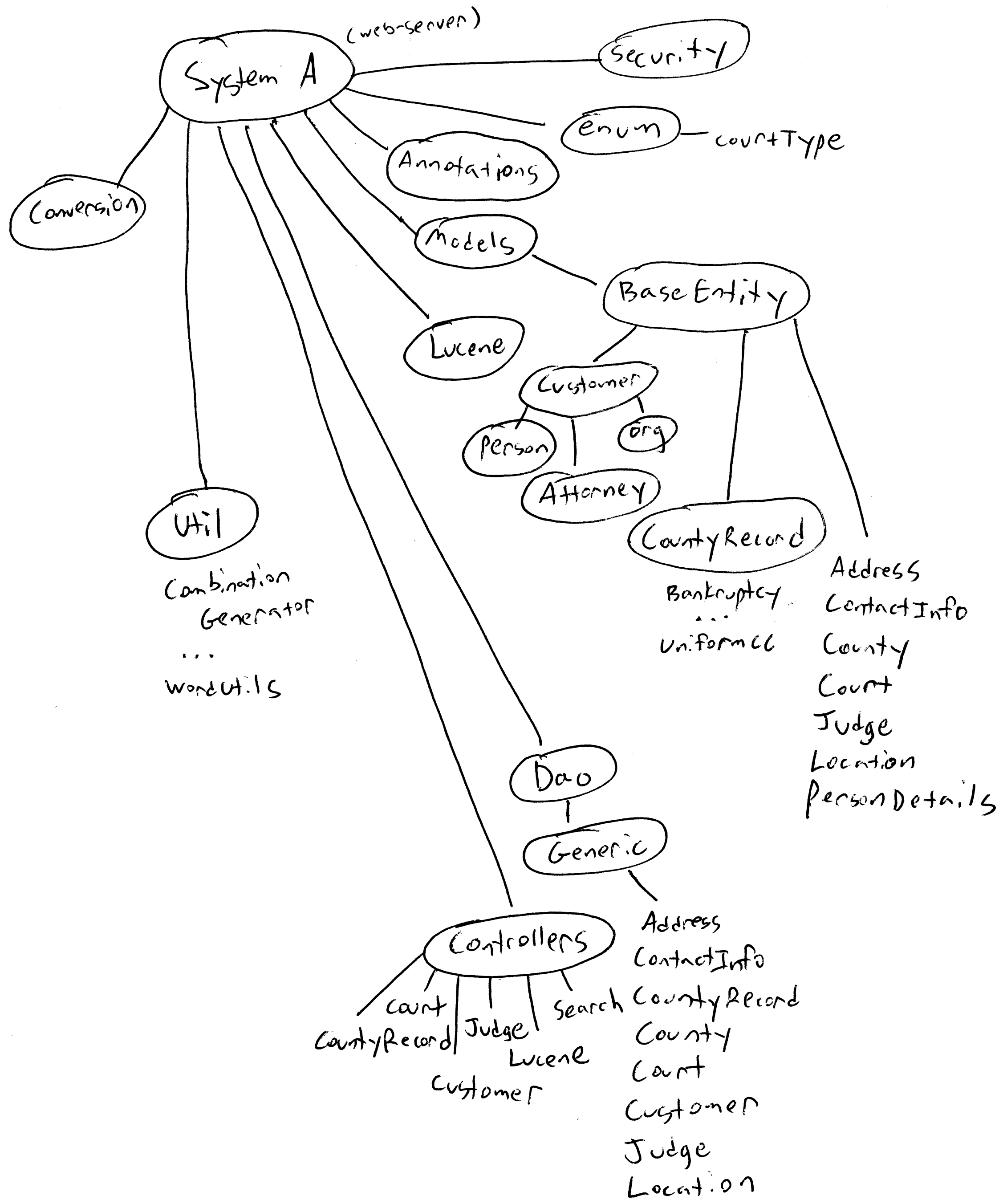

Figure C.3

System A high level architecture diagram by participants

APPENDIX D

SYSTEM B ARTIFACTS

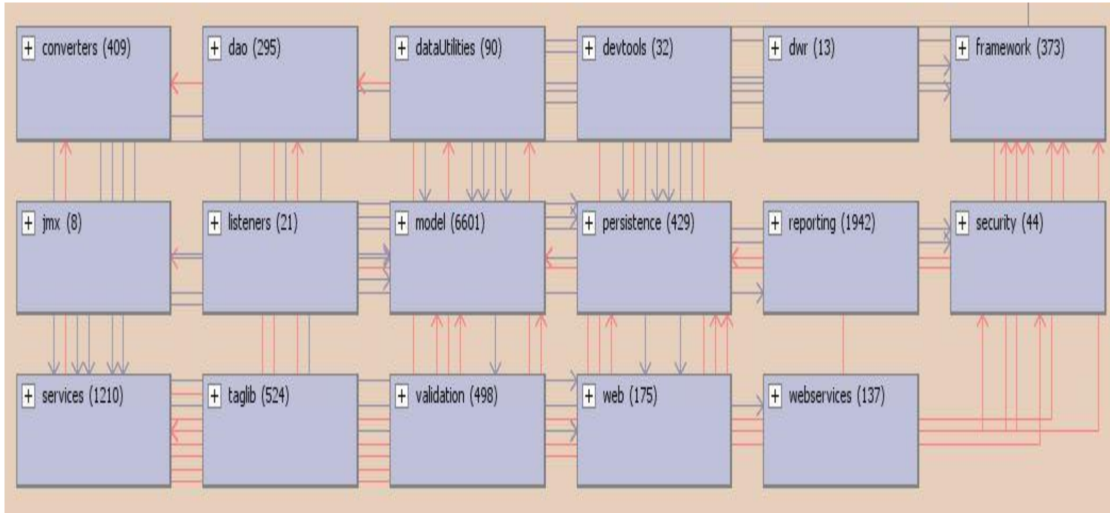

Figure D.1

System B conceptual architecture

		..1	..2	..3	..4	..5	..6	..7	..8	..9	10	11	12	13	14	15	16	17	18	19	20
spring	1	18																			
converters	2		409												57	13		186			
dao	3			295								8		11	523		30	17			
dataUtilities	4				90										37		8				
devtools	5					32															
dwr	6						13														
framework	7			5			1	373		3		27			48	2	10	4			
jmx	8								8							1					
listeners	9									21											
model	10		751	961	10	25	5	12		1	6601	48	631	45	2895	61	437	31	11		
persistence	11			609		6	12	6		5		429	399	3	134		3	15			2
reporting	12		17										1942								
security	13			4		4								44	10		4				
services	14		3	1		6				3		6			1210		27	4	16		
taglib	15															524				3	
validation	16							6									498				
web	17						3	2		5					7	4		175			
webservices	18														4				137		
springbyexample.web.servlet.view.tiles2	19							6												147	
springframework.webflow.conversatio...	20																				31

Figure D.2

System B dependency structure matrix

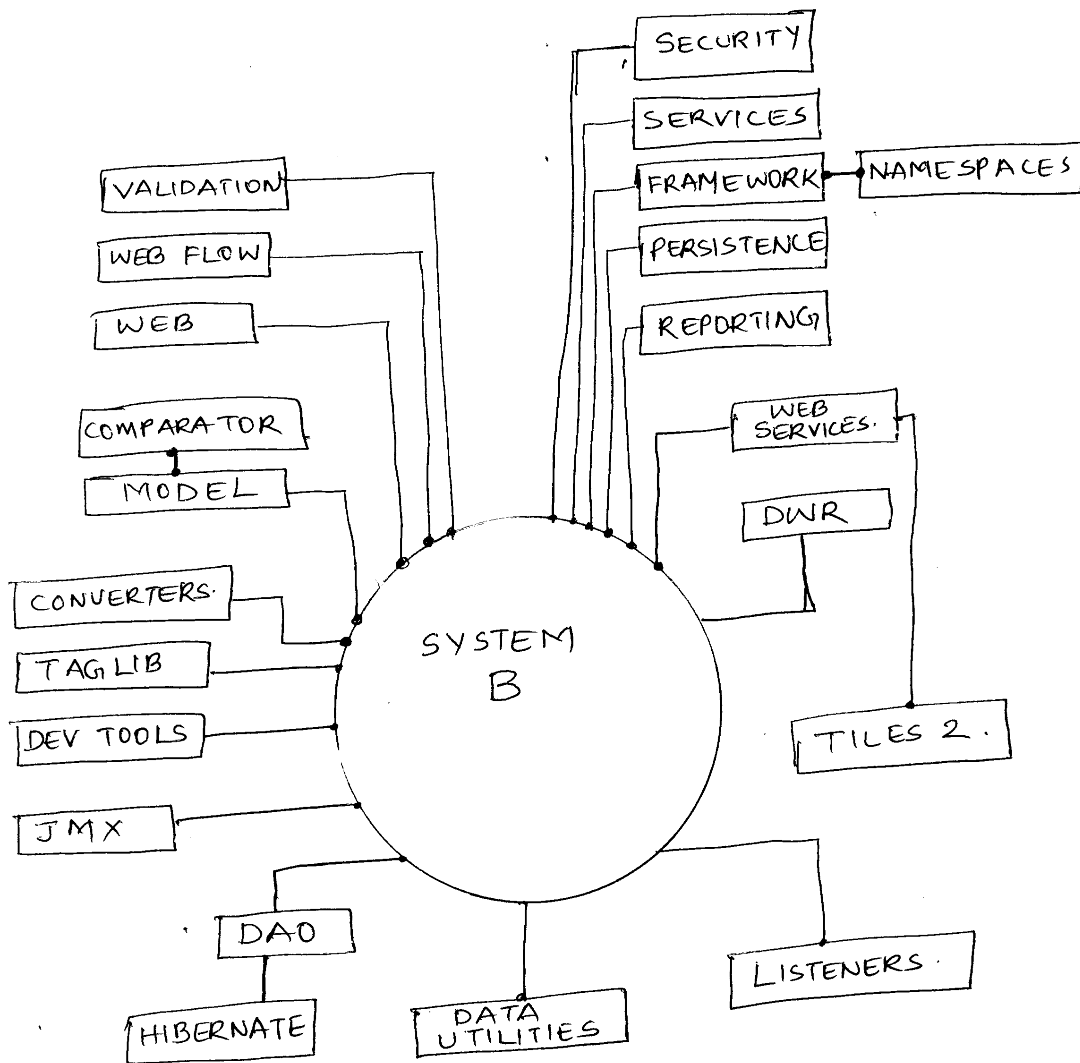

Figure D.3

System B high level architecture diagram by participants

APPENDIX E

SOURCE CODE

E.1 Source code for CBMCCalculator

This section presents the source to calculate $CBMC$ values.

```
package coupling;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

public class CBMCCalculator {
/**
 * @param args
 * @throws FileNotFoundException
 */
public static void main(String[] args) throws
              FileNotFoundException {

ArrayList<String> sourceList = new ArrayList<String>();
ArrayList<String> targetList = new ArrayList<String>();
ArrayList<String> sourcePackages = new ArrayList<String>();
ArrayList<String> targetPackages = new ArrayList<String>();

// Input from text file
Scanner myScanner = new Scanner(new File("exam"));

// Read from the text file
String line;

//Populate the source and target lists
while(myScanner.hasNext()){
line = myScanner.next();
String source = line.substring(0, line.indexOf(',')));
String target = line.substring(line.indexOf(',')+1);
sourceList.add(source);
targetList.add(target);
}//end while
myScanner.close();

//Remove bidirectional uses.
removeBidirectionalUses(sourceList, targetList);
//remove intra coupling
```

```
ArrayList<String> list = removeIntraUses(sourceList, targetList,
sourcePackages, targetPackages);

System.out.println("********Printing CBMC values********");
int tcbmc = computeCBMC(list);
int totalCBMC = tcbmc/2 ;
System.out.println("Total CBM: "+ totalCBMC);
}//end main

/**
 *
 * @param sourcePackages
 */
private static int computeCBMC(ArrayList<String> sourcePackages) {
Set<String> uniqueSet = new HashSet<String>(sourcePackages);
int cbmc = 0;
for (String temp : uniqueSet) {
System.out.println(temp + ": " +
                   Collections.frequency(sourcePackages, temp));
cbmc = cbmc +  Collections.frequency(sourcePackages, temp);
}
return cbmc;
}

/**
 *
 * @param sourceListIn
 * @param targetListIn
 * @param sourcePackagesIn
 * @param targetPackagesIn
 * @returnList with all the source and targets after removing
 *intra uses (duplicates and crisscross)
 */
private static ArrayList<String> removeIntraUses
(ArrayList<String> sourceListIn, ArrayList<String> targetListIn,
ArrayList<String> sourcePackagesIn,
ArrayList<String> targetPackagesIn ){

ArrayList<String> finalList = new ArrayList<String>();
ArrayList<String> newSourceList = new
ArrayList<String>(sourceListIn);
ArrayList<String> newTargetList = new
ArrayList<String>(targetListIn);
ArrayList<Integer> flagList = new ArrayList<Integer>();
//System.out.println("Old source list Size: "+
sourceListIn.size());
```

```
//System.out.println("New source list Size: "+
newSourceList.size());
for (int i = 0; i < newSourceList.size(); i++) {
flagList.add(0);
}
for(int i=0; i<newSourceList.size(); i++){
String sourcePackage = getPackageName(newSourceList.get(i));
String targetPackage = getPackageName(newTargetList.get(i));
sourcePackagesIn.add(sourcePackage);
targetPackagesIn.add(targetPackage);
if(sourcePackage.contains(targetPackage) ||
targetPackage.contains(sourcePackage)){
//this means remove the element from the arraylist.
flagList.set(i, -1);
}
}

for(int i=0; i<flagList.size();i++){
System.out.println(flagList.get(i));
}

for(int i=0; i<flagList.size();i++){
int flag = flagList.get(i);
if (flag == -1){
System.out.println("source: "+newSourceList.get(i));
System.out.println("target: "+newTargetList.get(i));
}
else{
//add package names to source lists.
finalList.add(sourcePackagesIn.get(i));
finalList.add(targetPackagesIn.get(i));
}
}
return finalList;
}

/**
 * getUnidirectional values.
 * @param sourceList
 * @param targetList
 */
private static void removeBidirectionalUses(ArrayList<String>
 sourceList,
ArrayList<String> targetList) {
//Checking for criss cross elements
for(int i=0; i<targetList.size();i++){
```

```
String target = targetList.get(i);
for(int j=i+1; j<=sourceList.size()-1;j++){
String source = sourceList.get(j);
//Check whether the target is in source
if(compareNodes(target,source)){
//Now check whether the source is in target
if(compareNodes(sourceList.get(i), targetList.get(j))){
sourceList.remove(j);
targetList.remove(j);
}
}
}//end for:j
}//end for:i

//Checking for same elements.
for(int i=0; i<sourceList.size(); i++){
String source1 = sourceList.get(i);
String target1 = targetList.get(i);
for(int j=i+1; j<=sourceList.size()-1; j++){
String source2 = sourceList.get(j);
String target2 = targetList.get(j);
if(source1.equals(source2) && target1.equals(target2)){
sourceList.remove(j);
targetList.remove(j);
}//end if
}//end for:j
}//end for:i
}//end method: removeBidirectionalUses

/**
 *
 * @param source
 * @param target
 * @return
 */
public static boolean compareNodes(String source, String target){
if(source.equals(target)) return true;
else return false;
}

/**
 *
 * @param list
 */
public static void print(ArrayList<String> list){
for(int i=0; i<list.size(); i++){
```

```
System.out.println(list.get(i));
}
}

/**
 *
 * @param element
 * @return
 */
public static String getPackageName(String element){
String sub = element.substring(0,element.lastIndexOf("."));
String packageName = sub.substring(sub.lastIndexOf(".")+1);
return packageName;
}
}
```

E.2 Source code for CBMCalculator

This section presents the source to calculate CBM values.

```
package coupling;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.Scanner;
import java.util.Set;

public class CBMCalculator {
public static void main(String[] args) throws
 FileNotFoundException {
ArrayList<String> sourceList = new ArrayList<String>()
ArrayList<String> targetList = new ArrayList<String>();

// Input from text file
Scanner myScanner = new Scanner(new File("exam"));
// Read from the text file
String line;

//Populate the source and target lists
while(myScanner.hasNext()){
line = myScanner.next();
String source = line.substring(0, line.indexOf(','));
```

```
String target = line.substring(line.indexOf(',')+1);
sourceList.add(source);
targetList.add(target);

}//end while
myScanner.close();

removeBidirectionalUses(sourceList, targetList);
int sCBM= computeCBM(sourceList);
int tCBM = computeCBM(targetList);
int totalCBM = (sCBM+tCBM)/2 ;
System.out.println("Total CBM: "+ totalCBM);
}

private static int computeCBM(ArrayList<String>
sourceListIn) {
Set<String> uniqueSet = new HashSet<String>(sourceListIn);
int cbm = 0;
for (String temp : uniqueSet) {
System.out.println(temp + ": " +
 Collections.frequency(sourceListIn, temp));
cbm = cbm +  Collections.frequency(sourceListIn, temp);
}

return cbm;
}

/**
 * getUnidirectional values.
 * @param sourceList
 * @param targetList
 */
private static void removeBidirectionalUses(ArrayList<String>
 sourceList,
ArrayList<String> targetList) {
//Checking for criss cross elements
for(int i=0; i<targetList.size();i++){
String target = targetList.get(i);
for(int j=i+1; j<=sourceList.size()-1;j++){
String source = sourceList.get(j);
//Check whether the target is in source
if(compareNodes(target,source)){
//Now check whether the source is in target
if(compareNodes(sourceList.get(i), targetList.get(j))){
//System.out.println(i+"    "+j);
sourceList.remove(j);
```

```
targetList.remove(j);
}
}
}
}
//Checking for same elements.
for(int i=0; i<sourceList.size(); i++){
String source1 = sourceList.get(i);
String target1 = targetList.get(i);
for(int j=i+1; j<=sourceList.size()-1; j++){
String source2 = sourceList.get(j);
String target2 = targetList.get(j);
if(source1.equals(source2) && target1.equals(target2)){
sourceList.remove(j);
targetList.remove(j);
}
}
}
}

public static boolean compareNodes(String source,
 String target){
if(source.equals(target)) return true;
else return false;
}

public static void print(ArrayList<String> list){
for(int i=0; i<list.size(); i++){
System.out.println(list.get(i));
}
}
}
```